LeakWatch: Estimating Information Leakage from Java Programs

Tom Chothia[1], Yusuke Kawamoto[2]*, and Chris Novakovic[1]

[1] School of Computer Science, University of Birmingham, UK
[2] INRIA Saclay & LIX, École Polytechnique, France

Abstract. Programs that process secret data may inadvertently reveal information about those secrets in their publicly-observable output. This paper presents LeakWatch, a quantitative information leakage analysis tool for the Java programming language; it is based on a flexible "point-to-point" information leakage model, where secret and publicly-observable data may occur at any time during a program's execution. LeakWatch repeatedly executes a Java program containing both secret and publicly-observable data and uses robust statistical techniques to provide estimates, with confidence intervals, for min-entropy leakage (using a new theoretical result presented in this paper) and mutual information. We demonstrate how LeakWatch can be used to estimate the size of information leaks in a range of real-world Java programs.

Keywords: Quantitative information flow, statistical estimation, Java, mutual information, min-entropy leakage

1 Introduction

An information leak occurs when a passive observer learns something about a system's secret data by observing its public outputs. Information leaks may be a side effect of a correctly-functioning system, and pose no real threat to security (e.g., a rejected guess of a secret, high-entropy password leaks some information: that this value is not the correct password). Larger information leaks, on the other hand, may lead to a complete breakdown of security (e.g., a flawed random number generator may give an observer all the information they need to guess important secret values). It is therefore important for a designer or analyst of a system to know exactly where information leaks occur and to be able to quantify them.

Information theory is a useful mechanism for providing quantitative bounds on what an attacker can learn. The attacker's uncertainty about a system's secret data is usually represented as Shannon entropy [1], and the reduction in uncertainty about the secret data is represented using a measure such as the mutual information of the secret data and publicly-observable data [2], or the

* Work by Yusuke Kawamoto was supported by a postdoc grant funded by the IDEX Digital Society project.

min-entropy leakage from the secret data to the publicly-observable data [3]. Although the bounds provided by these measures are meaningful, it is tedious to manually compute them; there is therefore a need for tools that automatically and robustly detect information leakage vulnerabilities in software.

This paper presents LeakWatch, a quantitative information leakage analysis tool for the Java programming language. It is based on a "point-to-point" information leakage model in which secret and publicly-observable data may occur at any time during the program's execution, including inside complex code structures such as branches and nested loops. This model, which we developed previously using a semantics based on discrete-time Markov chains [4], is particularly well-suited to analysing complex programs where secret and publicly-observable data may occur at any point: if secret and publicly-observable values are "tagged" using the secret and observe commands respectively, it measures how much information a passive attacker with knowledge of the program's source code learns about the secret values by examining the observable values.

Given a Java program whose source code has been annotated with the positions where its secret and publicly-observable data occurs, LeakWatch repeatedly executes it, recording the occurrences of secret and public data, and then performs robust statistical tests to detect whether an information leak is present and, if so, estimate the size of the leak. We note that this relies on the analyst correctly identifying which values in their program should be kept secret and which other values might be observable to an attacker, but, assuming this is done correctly (a reasonable assumption for most programs), LeakWatch can be used to verify whether the program is secure, or whether it contains an information leak that could lead to an attack.

LeakWatch uses previous techniques [5,6] for estimating mutual information and a new technique for estimating min-entropy leakage. These are brute-force approaches for probabilistic systems; i.e., we must run the program enough times to collect sampled data for every possible secret value. If the systems we target were deterministic, we could compute the information leakage precisely; however, since they are probabilistic, we use these statistical estimation techniques to distinguish a real information leak from noise in the measurements and to place bounds on the possible leakage.

We note that this estimation technique is quite different from those that estimate mutual information or min-entropy leakage with sampled data for only *some* of the possible secrets (e.g., [7]). These results often require additional assumptions about the distribution of the secret values, which we do not make, and may only work for non-probabilistic systems. Practically, we can analyse systems containing tens (or, in some cases, hundreds) of thousands of secret and observable values that occur with a non-negligible probability, and in which each trial run of the system is independent and identically distributed. We show that this provides interesting results for complex systems.

We present new results for calculating when enough samples have been collected for our estimates to be accurate, and handling user (but not attacker) input to a system. We provide a full Java-based implementation of our analysis method, and illustrate its power with three realistic security-themed examples.

Other tools, such as QUAIL [8], QIF [9] and our earlier CH-IMP implementation [4], compute the leakage from small formal models of programs. A key difference in this work is that we target full Java programs, at the cost of estimating leakage instead of computing it precisely.

There are other information leakage tools built on model checkers for C and Java [10,11,12]. They require the secret values to be inputs to the program and the observable values to be the program's final outputs; they are also restricted to the subset of the language's syntax supported by the model checker. LeakWatch has neither of these constraints.

In summary, our main contributions are the following:

a) a new result for estimating min-entropy leakage from trial runs of systems, as well as providing confidence intervals for those estimates using χ^2 tests;

b) a technique that, given certain assumptions, ensures we have enough samples to estimate information leakage from trial runs of a system;

c) LeakWatch, a robust information leakage analysis tool that can estimate mutual information and min-entropy leakage in Java programs; LeakWatch is freely available at [13], with full documentation and a range of sample Java programs.

The rest of the paper is organised as follows. In Section 2 we introduce relevant theoretical background information. In Section 3 we show a new theoretical result for estimating min-entropy leakage and its confidence interval from trial runs of a system. In Sections 4 and 5 we discuss LeakWatch's design and implementation respectively. In Section 6 we show three examples of LeakWatch being used to uncover information leakage vulnerabilities in real-world Java programs.

2 Background

2.1 Leakage Measures and Estimating Mutual Information

Our approach assumes that trial runs of the system are independent and identically distributed; the analyst must verify that this is the case. We also assume the system has probabilistic behaviour: each run results in some secret values $x \in \mathcal{X}$ occurring from some probability distribution X, and some observable behaviour $y \in \mathcal{Y}$ occurring from some probability distribution Y. Then, for each run of the system, the probability of the secrets x occurring and the attacker observing y is given by the joint probability distribution $p(x, y)$. The question we wish to answer is "how much does an attacker learn about the value of the secret from the observable behaviour of the system?".

We use two popular measures of information leakage: mutual information and min-entropy leakage. Mutual information is given by the equation

$$I(X;Y) = \sum_{x\in\mathcal{X},y\in\mathcal{Y}} p(x,y)\log_2\left(\frac{p(x,y)}{p(x)p(y)}\right) \quad (1)$$

and tells us how much information, in bits, we learn about X by observing Y.

Min-entropy leakage is given by the equation

$$\mathcal{L}(X;Y) = \log_2 \sum_{y \in \mathcal{Y}} \max_{x \in \mathcal{X}} p(x,y) - \log_2 \max_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x,y) \tag{2}$$

and tells us how difficult it is for the attacker to guess the secret values in one attempt, given the observable behaviour. We refer the reader to [2] for a more in-depth evaluation of mutual information and min-entropy leakage as information leakage measures.

From trial runs of a system we can estimate the joint distribution $\hat{p}(x,y)$ and the distributions $\hat{p}(x)$ and $\hat{p}(y)$; we can use these distributions in Equation 1 to estimate the mutual information $\hat{I}(X;Y)$. To find bounds on the true mutual information of the secret and observable values in a system, we need to know how $\hat{I}(X;Y)$ relates to $I(X;Y)$. We have shown previously [14] how these values are related when the distribution on secrets is known and we estimated $\hat{p}(y|x)$; however, this case is different, in that we are also estimating $\hat{p}(x)$. This case has been studied by Moddemeijer [5] and Brillinger [6], who found that:

Theorem 1. *When $I(X;Y) = 0$, for a large number of samples n, $2n\hat{I}(X;Y)$ will be drawn from a χ^2 distribution with $(\#\mathcal{X}-1)(\#\mathcal{Y}-1)$ degrees of freedom; i.e., $\hat{I}(X;Y)$ has an average value of $(\#\mathcal{X}-1)(\#\mathcal{Y}-1)/2n$ and variance $(\#\mathcal{X}-1)(\#\mathcal{Y}-1)/2n^2$.*

Theorem 2. *When $I(X;Y) > 0$, for a large number of samples n, the estimates $\hat{I}(X;Y)$ will be drawn from a distribution with mean $I(X;Y) + (\#\mathcal{X}-1)(\#\mathcal{Y}-1)/2n + O\left(\frac{1}{n^2}\right)$ and variance*

$$\frac{1}{n}\left(\sum_{x,y} p(x,y)\log^2\left(\frac{p(x,y)}{p(x)p(y)}\right) - \left(\sum_{x,y} p(x,y)\log\left(\frac{p(x,y)}{p(x)p(y)}\right)\right)^2\right) + O\left(\frac{1}{n^2}\right).$$

Using these results, we first test a $\hat{I}(X;Y)$ value against the χ^2 distribution from Theorem 1; if it is consistent with the 95% confidence interval for this distribution, we conclude that there is no evidence of an information leak in our sampled data. If $\hat{I}(X;Y)$ is inconsistent with the distribution from Theorem 1, we conclude that there is evidence of an information leak in our sampled data and use Theorem 2 to calculate a confidence interval for the leakage.

In both cases "a large number of samples" means enough samples to ensure that every $\hat{p}(x,y)$ is close to $p(x,y)$, so it is important to note that this is a brute-force approach that requires many more samples than the product of the number of secret and observable values. The contribution of the estimation results is to allow us to analyse systems that behave probabilistically.

The $O(n^{-2})$ term in Theorem 2 is an infinite sum on descending powers of n. This term is a result of using the Taylor expansion of entropy and conditional entropy. To make use of Theorem 2 we require enough samples for the $O(n^{-2})$ term to be small. This will always be the case for a sufficiently large n, and we address how to tell when n is large enough in Section 4.1. In practice, these results let us analyse systems with tens of thousands of unique secrets and observables.

2.2 Our Information Leakage Model

Our tool uses a "point-to-point" information leakage model, which tells us how much an attacker learns about a secret value at a particular point in a program from the program's observable outputs. This model is particularly well-suited to analysing entire programs (rather than code fragments), and it is a generalisation of the more common model of information flow that measures the leakage from high-level secret inputs of a function to its low-level public outputs [15]. It is important to note that our information flow model measures the information leakage from the value of variable at a particular point in the program. This differs from (e.g.) the Jif [16] information flow model, which ensures that no value stored in a high-level variable ever affects the value stored in a low-level variable. We have previously [4] developed a formal model of this information leakage, showed that it can be computed precisely for a simple probabilistic language using discrete-time Markov chains and that it can be estimated from trial runs of a program.

The LeakWatch API provides the command `secret(v1)` to denote that the current value of the variable `v1` should be kept secret, and `observe(v2)` to denote that `v2` is a value the attacker can observe; it is up to the analyst to decide where to place these commands. We then measure the information leakage from occurrences of `v1` to occurrences of `v2`.

For example, consider a Java card game program in which a `Card` object (`theirCard`) is drawn from a deck and sent over an insecure socket to an opposing player. Another `Card` object (`ourCard`) is drawn from the deck and stored locally; the opponent is then given the opportunity to make a bet based on the value of `theirCard`. If an analyst wanted to estimate how much information a remote attacker learns about `ourCard` from `theirCard`, the code could be annotated as:

```
Card theirCard = deck.drawCard();
LeakWatchAPI.observe(theirCard);
opponent.writeToSocket(theirCard);
Card ours = deck.drawCard();
LeakWatchAPI.secret(ours);
if (opponent.placedBet()) determineWinner();
```

This would, for example, alert the analyst to a badly-implemented random number generator in the deck-shuffling algorithm that allows the opposing player to predict the value of the next card dealt from the deck, giving them an unfair advantage when deciding whether to bet.

We note that our measure of information leakage only tells us what a passive attacker learns about the secret values by examining the observable values; it does not measure how easy the secret value is to guess (e.g., because it has low entropy). Therefore, the leakage measurement is only useful when there is uncertainty about the secret values. This could be due to secret values being randomly-generated numbers, or programs exhibiting unpredictable behaviour such as process scheduling or network timing. In cases where the secret is an input to the system, code can be added to generate a truly random value for the secret and then measure the leakage to the observable values. We give examples illustrating all of these cases in Section 6 and on the LeakWatch web site [13].

3 Estimating Min-Entropy Leakage

Our mutual information estimation result calculates the exact distribution of the estimates and so lets us calculate exact confidence intervals; obtaining a similar result for min-entropy leakage is difficult because of the maximum in its definition. So, to allow us to calculate this popular leakage measure, we find upper and lower bounds for a (more than) 95% confidence interval.

The estimation gives a point estimate of the leakage $\mathcal{L}(X;Y)$ from a distribution X on secret values to a distribution Y on observable values, and its (more than) 95% confidence interval. We do not know the exact distribution X, so we estimate it from the trial run data. Since we do not know the exact joint probability distribution $p(x,y)$ for the system, we first calculate the empirical joint distribution from the trial runs. Let L be the total number of trial runs, and $\hat{s}(x,y)$ be the frequency of (i.e., the number of trial runs with) a secret $x \in \mathcal{X}$ and an observable $y \in \mathcal{Y}$; then the empirical probability of having a secret x and an observable y is defined by $\frac{\hat{s}(x,y)}{L}$. Also, let $\hat{u}(x)$ be the frequency of a secret $x \in \mathcal{X}$; i.e., $\hat{u}(x) = \sum_{y\in\mathcal{Y}} \hat{s}(x,y)$; then we calculate the empirical probability of seeing x as $\frac{\hat{u}(x)}{L}$. Using these empirical distributions we obtain a point estimate $\widehat{\mathcal{L}}(X;Y)$ of the min-entropy leakage:

$$\widehat{\mathcal{L}}(X;Y) = -\log_2 \max_{x\in\mathcal{X}} \frac{\hat{u}(x)}{L} + \log_2 \sum_{y\in\mathcal{Y}} \max_{x\in\mathcal{X}} \frac{\hat{s}(x,y)}{L}.$$

Given L independent and identically distributed trial runs, the frequency $\hat{s}(x,y)$ follows the binomial distribution $B(L, p(x,y))$, where $p(x,y)$ is the *true* joint probability of a secret x and an observable y occurring. We note that we cannot treat each of the empirical joint probabilities as independently sampled from a binomial distribution, because together they must sum to 1. Instead, we perform Pearson's χ^2 tests [17,18] for a large number L of trial runs.

The estimation of a confidence interval is based on the fact that, with a high probability, each observed frequency $\hat{s}(x,y)$ is close to the "expected frequency" $p(x,y)L$, where $p(x,y)$ is the true probability we want to estimate. By applying χ^2 tests, we evaluate the probability that the observed frequencies $\hat{s}(x,y)$ come from the joint probability distributions $p(x,y)$. Given the observed frequencies $\hat{s}(x,y)$ and the expected frequencies $p(x,y)L$, the χ^2 test statistics is defined by:

$$\chi^2 = \sum_{x\in\mathcal{X},\, y\in\mathcal{Y}} \frac{(\hat{s}(x,y) - p(x,y)L)^2}{p(x,y)L}.$$

Since the joint probability distribution is regarded as a one-way table in this setting, the χ^2 test statistics follows the χ^2 distribution with $(\#\mathcal{X} \cdot \#\mathcal{Y}) - 1$ degrees of freedom. We denote by $\chi^2_{(0.05,k)}$ the test statistics with upper tail area 0.05 and k degrees of freedom.

The goal of our new method is to obtain a (more than) 95% confidence interval of the min-entropy leakage $\mathcal{L}(X;Y)$ between the secret and observable distributions X, Y. To obtain this, we estimate the 95% confidence intervals of the min-entropy $H_\infty(X) = -\log_2 \max_{x\in\mathcal{X}} p(x)$ and the conditional min-entropy $H_\infty(Y|X) = -\log_2 \sum_{y\in\mathcal{Y}} \max_{x\in\mathcal{X}} p(x,y)$ respectively.

We first present a method for obtaining the confidence interval of the conditional min-entropy $H_\infty(Y|X)$. Given L independent and identically distributed trial runs of the system, we obtain the observed frequencies $\hat{s}$. Then we construct expected frequencies $s_{\max}$ that give the largest *a posteriori* vulnerability $\sum_{y\in\mathcal{Y}} \max_{x\in\mathcal{X}} p(x,y)$ among all expected frequencies that satisfy: $\chi^2_{(0.05,\#\mathcal{X}\#\mathcal{Y}-1)} = \sum_{x\in\mathcal{X},\, y\in\mathcal{Y}} \frac{(\hat{s}(x,y)-s_{\max}(x,y))^2}{s_{\max}(x,y)}$. More specifically, $s_{\max}$ is constructed from $\hat{s}$ by increasing only the maximum expected frequencies $\max_{x\in\mathcal{X}} \hat{s}(x,y)$ and by decreasing others, while keeping the total number of frequencies as L; i.e., $\sum_{x\in\mathcal{X},y\in\mathcal{Y}} s_{\max}(x,y) = L$. From $s_{\max}$ we calculate the empirical distribution $P^{\text{post}}_{\max}[x,y] = \frac{s_{\max}(x,y)}{L}$. Next, we construct expected frequencies $s_{\min}$ that give the smallest *a posteriori* vulnerability. Keeping the total number of frequencies as L, we repeatedly decrease the current maximum expected frequency and increase the smallest frequencies until we obtain $\chi^2_{(0.05,\#\mathcal{X}\#\mathcal{Y}-1)} = \sum_{x\in\mathcal{X},\, y\in\mathcal{Y}} \frac{(\hat{s}(x,y)-s_{\min}(x,y))^2}{s_{\min}(x,y)}$. Then we calculate the corresponding distribution $P^{\text{post}}_{\min}$. From $P^{\text{post}}_{\max}$ and $P^{\text{post}}_{\min}$ we obtain the following confidence interval of the conditional min-entropy:

Lemma 1. *The lower and upper bounds for the 95% confidence interval of the conditional min-entropy $H_\infty(Y|X)$ are respectively given by:*

$$H^{\text{low}}_\infty(Y|X) = -\log_2 \sum_{y\in\mathcal{Y}} \max_{x\in\mathcal{X}} P^{\text{post}}_{\max}[x,y],\ H^{\text{up}}_\infty(Y|X) = -\log_2 \sum_{y\in\mathcal{Y}} \max_{x\in\mathcal{X}} P^{\text{post}}_{\min}[x,y].$$

Next, we compute the confidence interval of the min-entropy $H_\infty(X)$. Given the observed frequencies $\hat{u}$, we construct expected frequencies $u_{\max}$ that give the largest vulnerability $\max_{x\in\mathcal{X}} p(x)$ such that $\chi^2_{(0.05,\#\mathcal{X}-1)} = \sum_{x\in\mathcal{X}} \frac{(\hat{u}(x)-u_{\max}(x))^2}{u_{\max}(x)}$. We calculate the empirical distribution $P^{\text{prior}}_{\max}[x] = \frac{u_{\max}(x)}{L}$. Similarly, we construct expected frequencies $u_{\min}$ giving the smallest vulnerability, and calculate the corresponding distribution $P^{\text{prior}}_{\min}$. Then we obtain the following:

Lemma 2. *The lower and upper bounds for the 95% confidence interval of the min-entropy $H_\infty(X)$ are respectively given by:*

$$H^{\text{low}}_\infty(X) = -\log_2 \max_{x\in\mathcal{X}} P^{\text{prior}}_{\max}[x], \quad H^{\text{up}}_\infty(X) = -\log_2 \max_{x\in\mathcal{X}} P^{\text{prior}}_{\min}[x].$$

Finally, we obtain a confidence interval for the min-entropy leakage:

Theorem 3. *The lower and upper bounds for a more than 95% confidence interval of the min-entropy leakage $\mathcal{L}(X;Y)$ are respectively given by:*

$$\mathcal{L}^{\text{low}}(X;Y) = H^{\text{low}}_\infty(X) - H^{\text{up}}_\infty(Y|X),\ \mathcal{L}^{\text{up}}(X;Y) = H^{\text{up}}_\infty(X) - H^{\text{low}}_\infty(Y|X).$$

Note that our estimation technique for min-entropy leakage requires a large number of trial runs (usually many more than that required to estimate mutual information) to ensure that no more than 20% of the non-zero expected frequencies are below 5, which is a prerequisite for χ^2 tests.

Fig. 1 shows an example of mutual information and min-entropy leakage estimation; the graph is generated with 1,000 estimates of leakage from the 4-diner DC-net described in depth in Section 6.1) in which the random bits are biased towards 0 with probability 0.75. The graph shows the true leakage result, the estimated results from 10,000 runs of LeakWatch, and the bounds for min-entropy leakage. Our mutual information result gives the exact distribution of the estimates and so LeakWatch calculates the exact 95% confidence interval.

In this case, we observe that the estimation of min-entropy leakage is slightly biased and that it demonstrates more variation in the range of results than mutual information, although other examples (especially examples with unique maximum probabilities) demonstrate more variation in the mutual information estimate. In all cases, our bounds for the mutual information estimate are better than our bounds for min-entropy leakage; this is because we find the exact distribution of the estimates for mutual information and so can find a more accurate confidence interval.

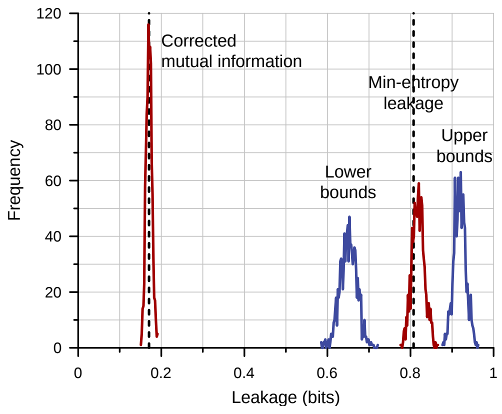

Fig. 1. The sampling distributions of mutual information and min-entropy leakage (and the lower/upper bounds for min-entropy leakage's confidence interval).

4 The Design of LeakWatch

LeakWatch is a robust tool intended for testing general-purpose, real-world Java programs where the presence of information leakage vulnerabilities may be a concern. It repeatedly executes a target program, recording the secret and observable values it encounters, and then uses the estimation results described in Sections 2 and 3 to find bounds on the information leakage from the program.

We target Java because of its enterprise popularity and common usage in large software projects. LeakWatch requires minimal modifications to be made to the target program's source code: the analyst simply inserts calls to the LeakWatch API methods, `secret()` and `observe()` (indicating the occurrence of a secret and observable value respectively), identifies the name of the *main class* containing the target program's main method and instructs LeakWatch to estimate the leakage from that program. The repeated execution of the program in a manner that guarantees the correctness of the leakage estimation —

including handling idiosyncrasies of the Java Virtual Machine (JVM), such as class-loading — and computation of the leakage estimation are then performed by LeakWatch. As the target program is repeatedly executed, LeakWatch automatically determines whether the amount of secret and observable information gathered is sufficient to produce a reliable leakage estimation; no further user interaction is required.

An important requirement of our statistical tests is that the sampled data is independent and identically distributed. This requirement is easily fulfilled if the target program does not rely on any external state (e.g., a value read from a file) that may change between executions. Programs that rely on some external state that does not have a statistically significant effect on the observable behaviour of the program can also be analysed without modification. It may be possible to modify programs that do rely on external state (e.g., by replacing a network read with a randomly generated value) and still obtain useful results.

4.1 Collecting Sufficient Program Execution Data

For our estimation results to be meaningful, we must collect enough samples to ensure that the estimated joint distribution of the secret and observable information that occurs in the program is a close approximation of the true distribution. Recall from Theorem 2 that the mean and variance of the distribution from which mutual information estimates are drawn are both defined as Taylor series. For efficiency reasons, LeakWatch only evaluates the first-order Taylor polynomial (e.g., $\hat{I}(X;Y) - (\#\mathcal{X}-1)(\#\mathcal{Y}-1)/2n$ for the point estimate of mutual information). Evaluating more terms in the Taylor series is computationally expensive because the joint distribution of X and Y must be enumerated differently for each term. For a small number of samples n, it is likely that the $O(n^{-2})$ and higher-order terms are too large for the first-order Taylor polynomial to be a good approximation of the sum of the series, and so $\hat{I}(X;Y) - (\#\mathcal{X}-1)(\#\mathcal{Y}-1)/2n$ will change as n increases. However, for a large enough number of samples, the higher-order terms quickly become small enough to have no meaningful effect on the result, and $\hat{I}(X;Y) - (\#\mathcal{X}-1)(\#\mathcal{Y}-1)/2n$ will no longer change as n increases.

Based on this observation, LeakWatch uses the following heuristic to automatically determine when a sufficient amount of trial run data has been collected to minimise the higher-order terms in the Taylor series and therefore provide an accurate mutual information estimate. It computes an estimate $\hat{I}(X;Y)$ after $\max(\#\mathcal{X} \times \#\mathcal{Y}, 100)$ samples have been collected, and stops collecting samples if all of the following conditions are met:

1) $\#\mathcal{X}$, $\#\mathcal{Y} > 1$ (otherwise a leakage measure cannot be computed from the joint probability distribution);
2) $\#\mathcal{X}$ and $\#\mathcal{Y}$ have remained constant since these conditions were last checked (otherwise the values in condition 4 cannot be compared meaningfully);
3) the minimum number of samples ($4 \times \#\mathcal{X} \times \#\mathcal{Y}$) has been collected;
4) the value $\hat{I}(X;Y) - (\#\mathcal{X}-1)(\#\mathcal{Y}-1)/2n$ has not changed beyond a certain amount, configurable by the analyst, since these conditions were last checked

(otherwise the higher-order Taylor series terms are non-negligible).

If these conditions are not met, the process is repeated again when $\#\mathcal{X} \times \#\mathcal{Y}$ more samples have been collected.

This heuristic allows LeakWatch to produce accurate mutual information estimates and confidence intervals for target programs containing both small and large numbers of unique secret and observable values that occur with a non-negligible probability. The initial number of samples $\max(\#\mathcal{X} \times \#\mathcal{Y}, 100)$ prevents LeakWatch from terminating too early when analysing very small systems (e.g., those containing fewer than ten unique secret and observable values in total) due to the low value of $\#\mathcal{X} \times \#\mathcal{Y}$ in these systems.

To estimate min-entropy leakage, we need many more trial runs than we need to estimate mutual information. As mentioned in Section 3, we require that no more than 20% of the non-zero expected frequencies are below 5; LeakWatch therefore collects samples until this condition as well as the first three conditions in the above procedure for mutual information estimation are met.

5 Implementing LeakWatch

To test a program for information leakage, an analyst simply imports the LeakWatch API class into their Java program and tags the secret and publicly-observable data with the API's `secret()` and `observe()` methods. Execution and sandboxing of the target program are achieved using the core Java libraries, and leakage estimates are calculated by leakiEst [19,20], our information leakage estimation library for Java, which now implements our min-entropy leakage result from Section 3; this ensures that LeakWatch is not tethered to a particular version or implementation of the JVM specification.

We now discuss two implementation issues: ensuring the independence of target program executions, and automatically providing programs with simulated user input.

5.1 Ensuring the Independence of Target Program Executions

When a Java program is executed, a new JVM is created; this involves loading and uncompressing many megabytes of Java class files. To generate test data efficiently, LeakWatch uses the same JVM to execute each trial run of the program. However, ensuring that programs sharing the same JVM execute independently (a requirement for our statistical estimation method) is not trivial to guarantee.

Java programs consist of one or more classes that are loaded into memory on-demand; the task of locating the bytecode that defines a class, parsing the bytecode, and returning a reference to the new class to the caller is performed by a *classloader*, itself a Java class. A class with a given name may be loaded only once by a given classloader. The JVM contains three classloaders by default, arranged in a hierarchy: the *bootstrap classloader* loads the core Java classes (e.g., those whose names begin with `java.`), the *extensions classloader* loads the Java extension classes (e.g., those that perform cryptographic operations), and the *system classloader* loads other classes. This hierarchy is strictly enforced; e.g., the system classloader delegates the loading of `java.lang.String` to the extensions classloader, which in turn delegates it to the bootstrap classloader. By default, this means that both LeakWatch's classes and the target program's classes are loaded by the system classloader.

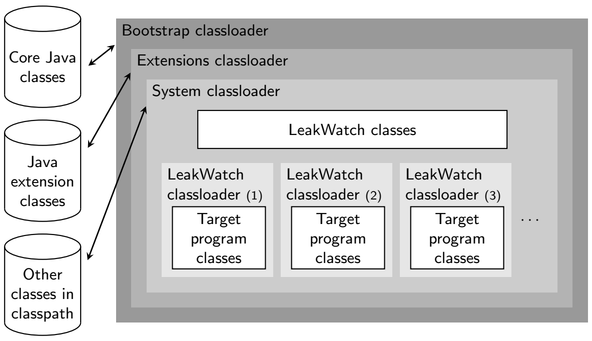

Fig. 2. The LeakWatch classloader hierarchy. Multiple copies of the target program can be executed simultaneously and in isolation using separate instances of the LeakWatch classloader.

LeakWatch runs the target program by invoking the main method in the target program's main class and waiting for it to return. Because a class may contain static member variables, having the system classloader load the target program's classes would be problematic: LeakWatch would not be able to "reset" the value of static member variables present in the target program's classes before invoking the main class's main method again, so some state may be preserved between executions of the target program. This potentially violates the independence of trial runs.

LeakWatch solves this problem by loading target program classes with its own classloader, positioned in the hierarchy between the system classloader and any classloaders used by the target program (see Fig. 2). Before each invocation of the main class's main method, a new LeakWatch classloader is initialised; it contains only the definition of the main class. As the main method executes, the LeakWatch classloader creates a new instance of any class required by the target program; subsequent requests for the class with that name will return this instance of the class, rather than the instance that would usually be returned by the system classloader. When the main method returns, LeakWatch destroys this classloader (and therefore any class loaded by it), ensuring that earlier invocations of the main method cannot interfere with future invocations. This guarantee even holds when multiple instances of the LeakWatch classloader exist concurrently, allowing LeakWatch to perform multiple isolated invocations of the main method at the same time using multithreading.

If a class is usually loaded by either the bootstrap or extensions classloaders, the LeakWatch classloader must delegate the request to them, so all executions of the target program "see" the same copies of the Java API classes. Although information could be shared between executions in this way, it is only possible with methods in a handful of classes (e.g., `java.lang.System`'s `setProperty()` method) and it is easy for the analyst to verify whether they are used.

5.2 Automatically Providing User Input to Target Programs

So that programs that rely on user input can be tested for information leakage, LeakWatch allows the analyst to specify an *input strategy* that provides input values to the program based on observable values that have previously occurred. To ensure independence of trial runs of the target program, we place two restrictions on LeakWatch's behaviour: only one input strategy may be defined for all trial runs, and the input values provided by the input strategy must depend only on the observable values that have occurred in the current trial run, and not in other trial runs. In previous work [4], we proved that if a program leaks information for any input strategy then it will also leak a non-zero amount of information for the input strategy that selects all possible inputs uniformly, so the uniform input strategy is a good default strategy.

Input can be provided to Java programs in many ways. We focus on input provided via the *standard input stream*, a universal method of supplying data to software; in Java, this stream is accessed with the static member variable `System.in` of type `java.io.InputStream`, whose `read()` method returns the next byte from the input buffer.

Operating systems provide a single standard input stream to a process; this means that all classes loaded by a particular JVM read from the same `System.in` stream. This is problematic because LeakWatch's classes and the target program's classes all execute within the same JVM; even though LeakWatch sandboxes each execution of the target program using its own classloader, all instances of the target program will share (and therefore read from) the same input stream. This is most noticeable when using multithreading to perform multiple isolated executions of the target program concurrently: two instances of a program reading 20 bytes of input from `System.in` will conflict, each using the `read()` method to read approximately 10 bytes from the same stream. This leaves both instances of the program with meaningless input, and violates the requirement that trial runs be independent and identically distributed.

We solve both problems by transforming every target class that reads from the standard input stream to instead read from an *input driver*, a mock object that mimics `System.in`. When using LeakWatch to analyse a program that reads from the standard input stream, the analyst must also write an appropriate input driver to supply input when it is required. The purpose of the input driver is to implement the input strategy described above: like `System.in`, it is a subclass of `java.io.InputStream`, but its `read()` method may consult the list of observable values that have been encountered so far during execution and return a stream of bytes comprising the selected input to the target program. When classes are loaded by LeakWatch's classloader, their bytecode is dynamically transformed (using the ASM [21] library) so that all references to `System.in` are replaced with references to the analyst's input driver. The loading of the input driver class is also performed by LeakWatch's classloader, so each execution of the target program believes it alone is reading from the standard input stream; this means that concurrent executions of a target program that reads from the standard input stream can progress without interfering with each other.

6 Practical Applications

We now present three examples demonstrating how LeakWatch can be applied to real-world situations to detect the presence of information leaks, quantify their size, and remove or mitigate them; they were benchmarked on a desktop computer with a quad-core CPU and 4GB of RAM. The source code for these examples is available for download from [13].

6.1 A Poorly-Implemented Multi-Party Computation Protocol

The *dining cryptographers problem* [22] investigates how anonymity can be guaranteed during secure multi-party computation. Informally: a group of cryptographers dine at a restaurant, and the waiter informs them that the bill is to be paid anonymously; it may be paid by any of the diners, or (e.g.) by the national security agent sitting at an adjacent table. After the bill has been paid, how do the diners collectively discover whether one of them paid the bill, while respecting their fellow diners' right to anonymity?

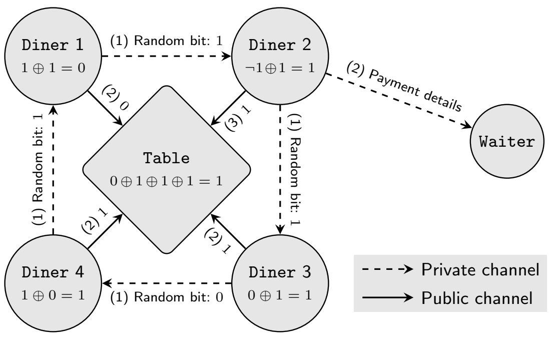

Fig. 3. An overview of the Java DC-net implementation. `Diner` 2 pays the bill; the final result is 1, indicating that one of the `Diner`s paid.

The *DC-net* is a solution to the dining cryptographers problem; it provides unconditional sender and recipient untraceability. Briefly, each diner generates a random bit visible only to them and the diner to their left, giving each diner sight of two separate randomly-generated bits; each diner computes the XOR of these two bits and announces the result publicly to the rest of the table — except for the payer, who announces the inverse of their XOR computation. The XOR of the announcements themselves allow each diner to verify whether one of them paid: if this value is 1, one of the diners claimed to have paid; if it is 0, nobody claimed to have paid. This protocol preserves the anonymity of the payer; however, care must be taken when implementing the protocol to ensure that side-channels do not leak information about the payer's identity.

This example implements a multithreaded, object-oriented DC-net in Java (see Fig. 3). Four `Diner`s are seated at a `Table` and — for the purposes of verifying whether the code contains an information leak — one of them is randomly selected to be the payer; the identity of the payer is marked as a secret with `secret()`. The `Diner`s then concurrently execute the protocol described above: they privately exchange their randomly-generated bits with each other using the socket libraries in the Java API, and the payer sends their payment details to the `Waiter` over another private socket. The `Diner`s then announce the results of their XOR computation to the rest of the `Table`; the messages sent to the `Table`'s socket are publicly visible, and are marked as observable with `observe()`. LeakWatch then answers the question "what does a passive attacker learn about the payer's identity by watching the messages sent to the `Table`?".

After 2,600 trial runs (taking 9 minutes to perform the required number of trial runs and 300ms to calculate the leakage estimate), LeakWatch estimates that there are ca. 1.15 of a possible 2 bits of mutual information between the identity of the payer and the messages the `Diner`s broadcast to the `Table`. The min-entropy leakage is found to be 0.47 bits after 6,080 trial runs; 11 minutes were spent performing the trial runs, and 221ms were spent calculating the estimate. Although the messages themselves reveal no information about the identity of the payer, the order in which they are sent does: the additional time taken by the payer to send their payment details to the `Waiter` means that, more often than not, they are one of the last `Diner`s to announce the result of their XOR computation to the `Table`. This leakage can be eliminated in several ways; e.g., modifying the implementation so that each `Diner` waits 100ms before sending their announcement to the `Table`'s socket makes it more likely that messages will arrive at the `Table`'s socket in any order. After 5,700 trial runs of this modified DC-net, LeakWatch confirms that there is no leakage of the payer's identity.

By increasing the number of `Diner`s participating in the DC-net, the communication protocol becomes more complex, and the amount of observable information increases exponentially. Fig. 5 (3 pages below) shows the amount of time LeakWatch takes to estimate the leakage from a simplified DC-net, with all socket-based communication removed and a new leak caused by biased random bit generation inserted (a 0 is generated with probability 0.75, rather than 0.5). The graph shows that, as more `Diner`s are added to this simplified DC-net, the amount of time LeakWatch takes to perform a number of trial runs that is sufficient to compute an accurate leakage estimate increases exponentially (ca. 2 hours when 17 `Diner`s participate). The amount of time required to estimate the size of the leak also increases exponentially, but remains comparatively very small (ca. 12 seconds when 17 `Diner`s participate), indicating that the vast majority of LeakWatch's time is spent collecting sufficient trial run data, and not computing leakage estimates.

6.2 Analysing the Design of Stream Ciphers

Crypto-1 is a stream cipher used to encrypt transmissions in commercial RFID tags. The design of this cipher was kept secret, but careful analysis (e.g., [23]) revealed that it is based on a 48-bit linear feedback shift register (LFSR). Each keystream bit is generated by applying two functions (f_a, f_b) to 20 bits of the state, and then applying a third function (f_c) to the outputs of these functions. In this example we use LeakWatch to show that the mutual information between the state bits and the keystream bits reveals much of this structure.

The initial state of the LFSR is derived from the key; to simplify the example, we assume that we can set the initial state directly. Information about the structure of Crypto-1 is revealed by loading different initial states into the

LFSR and observing the output from the final Boolean function f_c. Using a Java implementation of Crypto-1, an LFSR is created with a randomly-generated initial state, and the first output bit from f_c is computed. The value of this bit is marked as observable with `observe()`. Another LFSR is created with the same initial state as before, but with the value of the bit at index i flipped with probability $\frac{1}{2}$ — the decision about whether or not to flip this bit is marked as secret information with `secret()`. The output from f_c in this second cipher is then computed, and its value is also marked as observable with `observe()`. The question being asked here is "what is the correlation between the LFSR bit at index i being flipped and the output of f_c changing?"; informally, this can be seen as the influence of the bit at index i on the cipher's keystream.

By running LeakWatch 48 times, each time using a different value of i between 0 and 47 (taking a total of 19 seconds to perform the trial runs and a total of 425ms to produce the 48 leakage estimates), LeakWatch reveals which indices of the LFSR are tapped and passed as input to the Boolean functions; each execution of LeakWatch performs approximately 220 trial runs to determine the influence that that particular bit has on the keystream. Fig. 4 graphs the influence of each LFSR bit on the output of the cipher; points that fall above the dashed line near the x axis indicate a statistically significant correlation between flipping the LFSR bit at the index on the x axis and the first bit of the keystream changing. By reading off these indices on the x axis, we see which bits are tapped to produce the keystream. Moreover, the relative vertical distances between the points for each group of four indices reveal two distinctive patterns: these are the Boolean functions f_a and f_b (i.e., the pattern for groups $\{9, 11, 13, 15\}$ and $\{33, 35, 37, 39\}$ represents f_a, and the pattern for groups $\{17, 19, 21, 23\}$, $\{25, 27, 29, 31\}$ and $\{41, 43, 45, 47\}$ represents f_b); it is therefore possible to "see" which indices are tapped by each of these functions, as indicated by the dashed lines between the points in each group. The slight variation in the groups' distances from the x axis is accounted for by the third Boolean function f_c, into which the output from the other Boolean functions is fed.

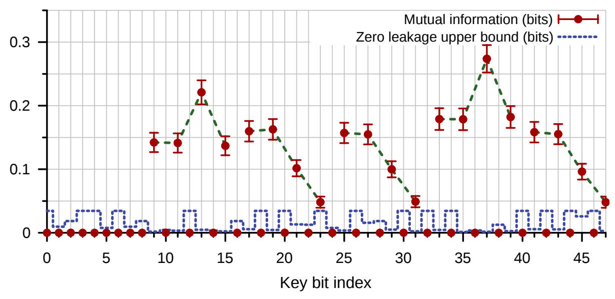

Fig. 4. The influence over the first bit of the keystream of each bit in a 48-bit secret initial state for Crypto-1.

This analysis shows that the output of this popular but flawed cipher reveals a lot of information about its internal design; it is therefore unsurprising that the cipher's design was fully reverse-engineered. The same technique can also be used to analyse other LFSR-based stream ciphers, such as Hitag-2 [24].

6.3 Recipient Disclosure in OpenPGP Encrypted Messages

OpenPGP [25] is a data encryption standard. In a typical usage scenario, encrypted OpenPGP messages contain two *packets*: the first contains a randomly-generated symmetric session key encrypted with the recipient's public key, and the second contains the sender's message, encrypted under the session key. Although OpenPGP provides message confidentiality and integrity, it does not necessarily provide recipient confidentiality because the first packet contains the recipient's *key ID* — the low 64 bits of the SHA-1 hash of their public key — which may be used to corroborate the recipient's identity with a resource mapping public keys to identities, such as an OpenPGP key server.

To demonstrate this, we present an example where two principals attempt to communicate securely using OpenPGP while concealing their identities from a passive attacker with the ability to read messages sent over the communication medium; the OpenPGP API is provided by the BCPG Java library [26]. In the program, a sender is chosen randomly from a pool of six principals, and a recipient is chosen from the remaining five; their identities are both marked as secret (with `secret()`). The sender greets the recipient by name, encrypts the greeting with the recipient's public key, and sends the encrypted message over an insecure medium, where it is monitored by the attacker. Two pieces of information are marked as observable by the attacker using separate calls to `observe()`: the header of the first packet in the encrypted OpenPGP message, and the length (in bytes) of the entire encrypted message. Thus, LeakWatch answers the question "how much information does an attacker learn about the principals' identities by observing these two features of the encrypted traffic?".

Assuming the two principals are selected uniformly, there are ca. 4.9 bits of secret information in this scenario (ca. 2.6 bits from the sender's identity and ca. 2.3 bits from the recipient's identity). After 550 trial runs (taking 17 seconds to perform the required number of trial runs and 300ms to produce the leakage estimate), LeakWatch reveals that, because BCPG includes the recipient's key ID in the first packet, there is a leakage of ca. 2.52 bits about the secret information: there is complete leakage of the recipient's identity, and a further leakage of ca. 0.2 bits of the sender's identity, because the attacker also knows that the sender is not the recipient.

Some OpenPGP implementations mitigate this leakage of the recipient's identity; e.g., GnuPG features a `-R` option that replaces the key ID in the first packet with a string of null bytes. By patching BCPG to do the same, the leakage decreases to ca. 1.86 bits after 350 trial runs: the recipient's identity is no longer leaked completely via the first packet, but because the attacker knows the format of the unencrypted message being sent, the length of the second packet still reveals some information about the recipient's identity (because the sender's encrypted message will be longer when greeting a recipient with a longer name).

Fig. 5 shows how increasing the number of bits in the first packet that are observable by the attacker affects LeakWatch's execution time. It reveals a result similar to that in Section 6.1: as the number of bits in the observable output increases, the amount of time required for LeakWatch to perform the number of trial runs required to estimate leakage increases exponentially (ca. 6 hours when the observation size reaches 132 bits); this is because of the exponentially-increasing number of trial runs required to verify whether parts of the randomly-generated encrypted session key leak information about the principals' identities. The amount of time required to estimate the size of the information leak from the trial run data, however, remains comparatively very small (ca. 1.5 seconds when the observation size reaches 132 bits), as it does in Section 6.1.

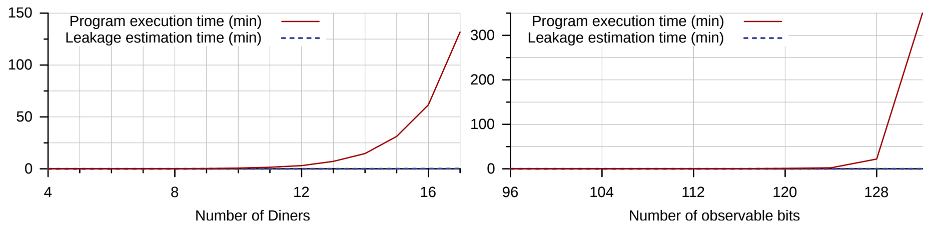

Fig. 5. The effect on LeakWatch's execution time of increasing the amount of secret or observable information in the examples in Sections 6.1 (the number of `Diners`, left) and 6.3 (the number of observable bits in the encrypted OpenPGP message, right).

7 Conclusion

We have presented new theoretical results and practical techniques for the statistical estimation of information leakage from real-world Java programs, based on trial runs. In particular, we have described a new method for estimating min-entropy leakage and its confidence interval, and a technique for ensuring the collection of a sufficient number of samples. We have also presented a mechanism that ensures the independence of trial runs of Java programs, and applied our information leakage model and estimation techniques to input-consuming Java programs. Using three examples, we have demonstrated that our robust information leakage analysis tool LeakWatch can uncover information leakage vulnerabilities in Java programs.

References

1. Shannon, C.E.: A Mathematical Theory of Communication. Bell System Technical Journal **27**(3) (July 1948) 379–423
2. Smith, G.: On the Foundations of Quantitative Information Flow. In: Proc. FOSSACS. (2009) 288–302
3. Smith, G.: Quantifying Information Flow Using Min-Entropy. In: Proc. of the 8th Conference on Quantitative Evaluation of Systems (QEST 2011). (2011) 159–167
4. Chothia, T., Kawamoto, Y., Novakovic, C., Parker, D.: Probabilistic Point-to-Point Information Leakage. In: Proc. of the 26th IEEE Computer Security Foundations Symposium (CSF 2013), IEEE Computer Society (June 2013) 193–205
5. Moddemeijer, R.: On estimation of entropy and mutual information of continuous distributions. Signal Processing **16** (1989) 233–248
6. Brillinger, D.R.: Some data analysis using mutual information. Brazilian Journal of Probability and Statistics **18**(6) (2004) 163–183

7. Boreale, M., Paolini, M.: On formally bounding information leakage by statistical estimation. Unpublished Manuscript `http://rap.dsi.unifi.it/~boreale/papers/StatQIF.pdf` (2014)
8. Biondi, F., Legay, A., Traonouez, L., Wasowski, A.: QUAIL: A Quantitative Security Analyzer for Imperative Code. In: Proc. of the 25th International Conference on Computer Aided Verification (CAV 2013). (2013)
9. Mu, C., Clark, D.: A tool: quantitative analyser for programs. In: Proc.of the 8th Conference on Quantitative Evaluation of Systems (QEST 2011). (2011) 145–146
10. McCamant, S., Ernst, M.D.: Quantitative Information Flow as Network Flow Capacity. In: Proc. of the Conference on Programming Language Design and Implementation (PLDI 2008). (2008) 193–205
11. Heusser, J., Malacaria, P.: Quantifying Information Leaks in Software. In: Proc. of the 2010 Annual Computer Security Applications Conference (ACSAC'10), Austin, Texas, USA, ACM Press (December 2010) 261–269
12. Phan, Q.S., Malacaria, P., Tkachuk, O., Păsăreanu, C.S.: Symbolic quantitative information flow. ACM SIGSOFT Software Engineering Notes **37**(6) (2012) 1–5
13. Chothia, T., Kawamoto, Y., Novakovic, C.: LeakWatch `http://www.cs.bham.ac.uk/research/projects/infotools/leakwatch/`.
14. Chatzikokolakis, K., Chothia, T., Guha, A.: Statistical Measurement of Information Leakage. In: Proc. of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010). Volume 6015 of Lecture Notes in Computer Science., Springer (March 2010) 390–404
15. Denning, D.E.: Cryptography and Data Security. Addison-Wesley (May 1982)
16. Myers, A.C., Liskov, B.: Complete, Safe Information Flow with Decentralized Labels. In: Proc. of the 1998 IEEE Symposium on Security and Privacy, Oakland, California, USA, IEEE Computer Society (May 1998) 186–197
17. Pearson, K.: X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. Philosophical Magazine Series 5 **50**(302) (1900) 157–175
18. Diez, D.M., Barr, C.D., Cetinkaya-Rundel, M.: OpenIntro Statistics. CreateSpace (2012)
19. Chothia, T., Kawamoto, Y., Novakovic, C.: A Tool for Estimating Information Leakage. In: Proc. of the 25th Conference on Computer Aided Verification (CAV 2013). Volume 8044 of Lecture Notes in Computer Science. (2013) 690–695
20. Kawamoto, Y., Chatzikokolakis, K., Palamidessi, C.: Compositionality Results for Quantitative Information Flow. In: Proc. of the 11th International Conference on Quantitative Evaluation of Systems (QEST 2014). (September 2014) To appear.
21. OW2 Consortium: ASM `http://asm.ow2.org`.
22. Chaum, D.: The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. In: Journal of Cryptology. (1988) 65–75
23. Garcia, F.D., van Rossum, P., Verdult, R., Schreur, R.W.: Wirelessly pickpocketing a Mifare Classic card. In: IEEE Symposium on Security and Privacy (S&P 2009), IEEE (2009) 3–15
24. Verdult, R., Garcia, F.D., Balasch, J.: Gone in 360 seconds: Hijacking with Hitag2. In: 21st USENIX Security Symposium (USENIX Security 2012), USENIX Association (2012) 237–252
25. Callas, J., Donnerhacke, L., Finney, H., Shaw, D., Thayer, R.: OpenPGP Message Format `http://tools.ietf.org/html/rfc4880`.
26. Legion of the Bouncy Castle Inc.: The Legion of the Bouncy Castle Java Cryptography APIs `https://www.bouncycastle.org/java.html`.