# LEAKWATCH: MEASURING POINT-TO-POINT INFORMATION LEAKAGE IN SMALL PROBABILISTIC JAVA PROGRAMS.

Tom Chothia

# My Co-Authors

Yusuke Kawamoto

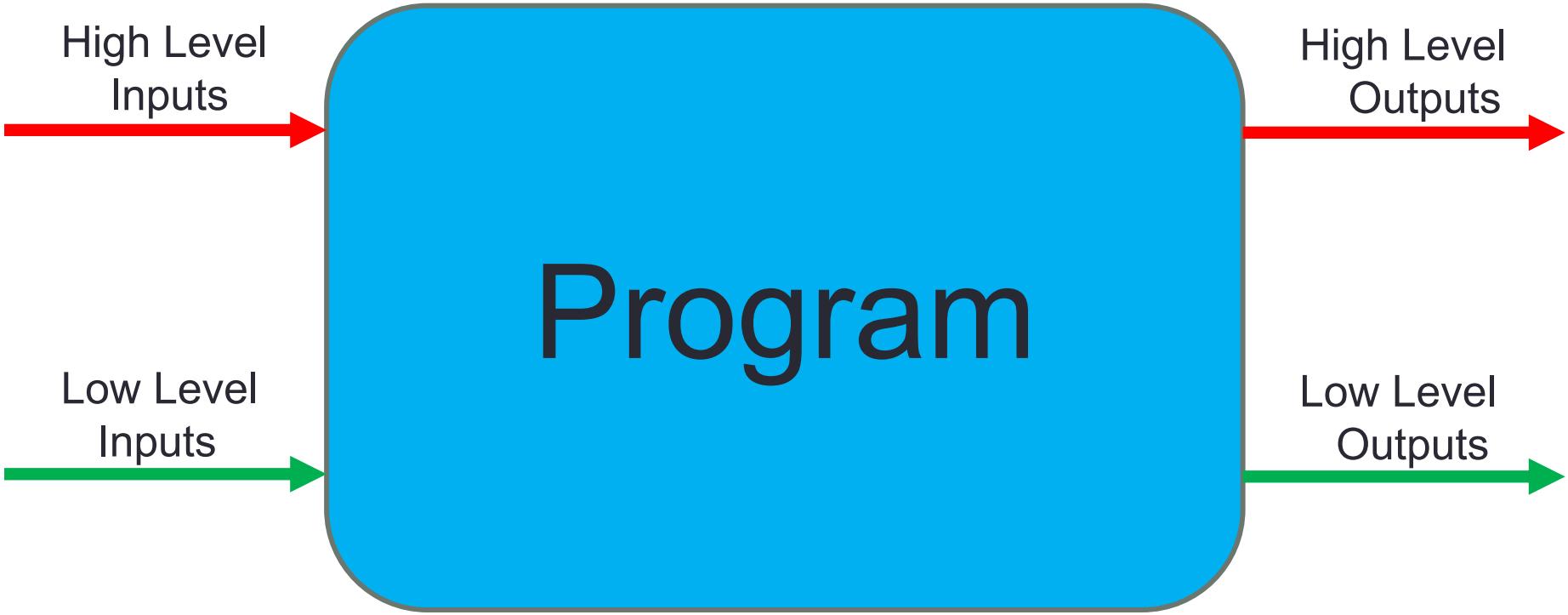
Apratim Guha

Chris Novakovic

# Introduction

- We adapt the classic information leakage model to allow us to tag any variables in a program as “observable” or “secret”.
  - Correlation, Mutual information, Min-entropy leakage, g-leakage
- We build a formal model that measures what the observable values tell us about the secrets.
- We discuss leakage from non-terminating programs.
- LeakWatch: a Framework for statistical measuring leakage from Java programs.

# Standard Leakage Model



# Correlation

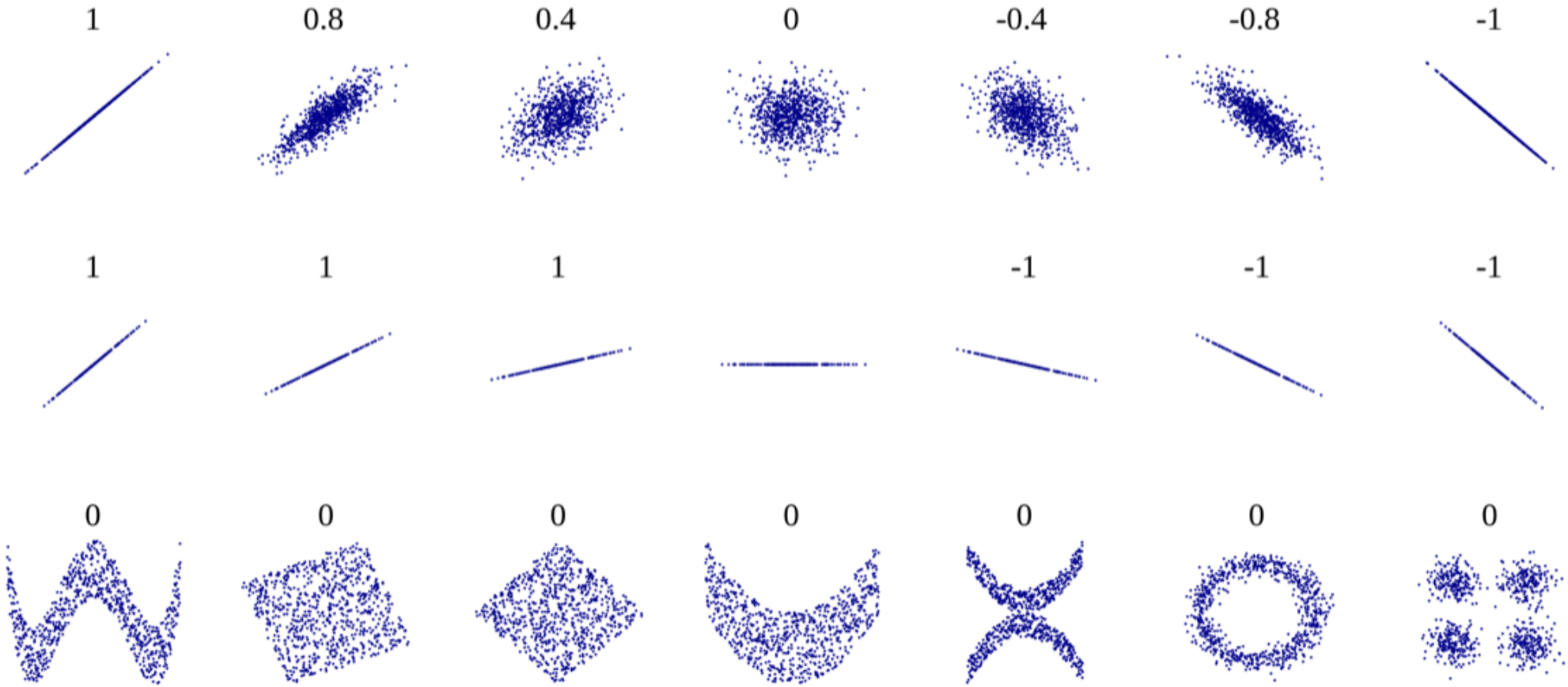



Image from wikipedia

# Mutual Information

- Given X and Y we can ask how much does one tell us about another? How much information “leaks” from X to Y?

- Mutual Information I(X;Y) is the reduction of uncertainty you get in X if you know Y:

$$I(X;Y) = H(X) - H(X \mid Y)$$

$$= \sum_{x \in X, y \in Y} p(x,y) \log \left( \frac{p(x,y)}{p(x)p(y)} \right)$$

# Guessing Entropy

- For a distribution X order the probabilities p1,...,pn such that that p1 ≥ p2 ≥ · · · ≥ pn.
- Guessing entropy is defined as:

$$E[G(X)] = \sum_{i=1}^{n} ip_i$$

- Number of guesses the attacker needs to guess X.
- Ordering requirements means it's hard to generalise and estimate.

# Min-entropy

- Min-entropy of X is the uncertainty in guessing X in one try:

$$H_\infty(X) = -\log(\max_{x \in X}(p(x)))$$

- The uncertainty in guessing X after having observed Y is:

$$H_\infty(X \mid Y) = -\log\left(\sum_{y \in Y} p(y) \max_{x \in X}(p(x \mid y))\right)$$

- Min-entropy leakage tells you how easier a system makes it to guess X in one try after seeing Y.

$$L(X;Y) = H_\infty(X) - H_\infty(X \mid Y)$$

# g-leakage

- g-leakage uses a “gain function” the measure how good a guess is for the attacks

$$V_g(\pi) = max_{w \in W} \sum_{y \in Y} \pi(x) g(w, x)$$

$$V_g(\pi, C) = \sum_{y \in Y} p(y)\, V_g(p_{X|y})$$

- g-leakage equals:

$$L_g(\pi, C) = \log V_g(\pi, C) - \log V_g(\pi)$$

or

$$L_g^+(\pi, C) = \log V_g(\pi, C) - \log V_g(\pi)$$

# Example 1

```
new rand :={0→0.5,1→0.5};
observe rand;
new sec :={0→0.5,1→0.5};
secret sec;
new out := sec xor rand;
observe out;
```

- We need to measure the information leakage to all observables, before and after the secret, together.

# Example 2

```
new sec1 :={0→0.5,1→0.5};
secret sec1;
new sec2 :={0→0.5,1→0.5};
secret sec1;
new out := sec1 xor sec2;
observe out;
```

- We need to measure the leakage from all secrets together.

# Example 3

```
new result := 0,i := 0;
while (i < 4) {
      observe result;
      new sec := {1 → 0.5, 2 → 0.5 };
      secret sec;
      if(i==2) { result := sec; }
      i :=i +1;
}
```

- We must measure the leakage form all data produced inside a loop to anywhere in the program (could be infinite).

# Example 4

```
new coin := {1 → 0.5, 2 →0.5 };
observe(coin);
if(coin==1) {
      new creditCard:= {1 → 0.5, 2 → 0.5 };
      secret(creditCard);
} else {
      new cash:= {1 → 0.5, 2 → 0.5 };
      secret(cash);
}
```

We decided to say this is a leak of 1 bit.

You would be free to disagree.

# Formalising this model (with Dave Parker)

## Syntax

```
C ::= new V:=ρ
|   V:=ρ
|   if(B) {C} else {C}
|   while( B ) { C }
|   C;C
...  int & expressions
|   secret (V)
|   observe (V)
```

## Semantics

- Discrete-time Markov chain semantics

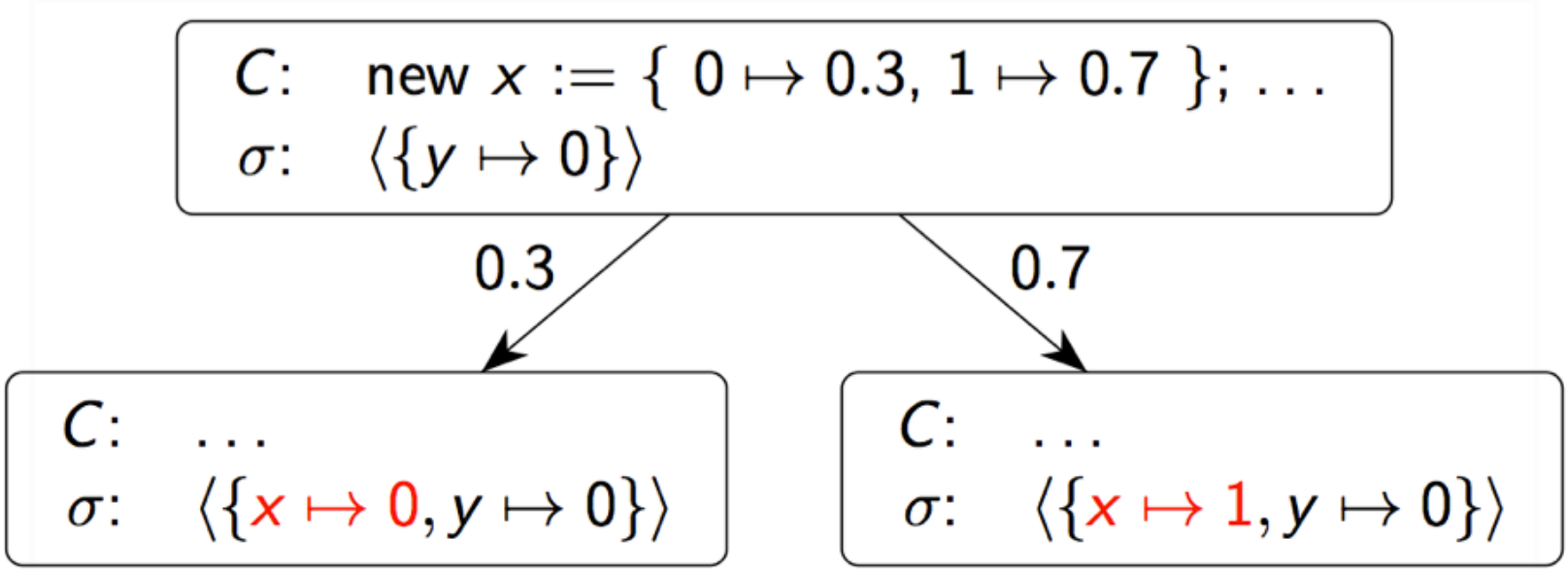
$$(C,\sigma,S,O) \rightarrow (C',\sigma',S',O')$$

Where:

- C: commands to be run
- σ: list of variable scopes
- S: previous secret mappings
- O: previous observed values

# Rule for random new variables

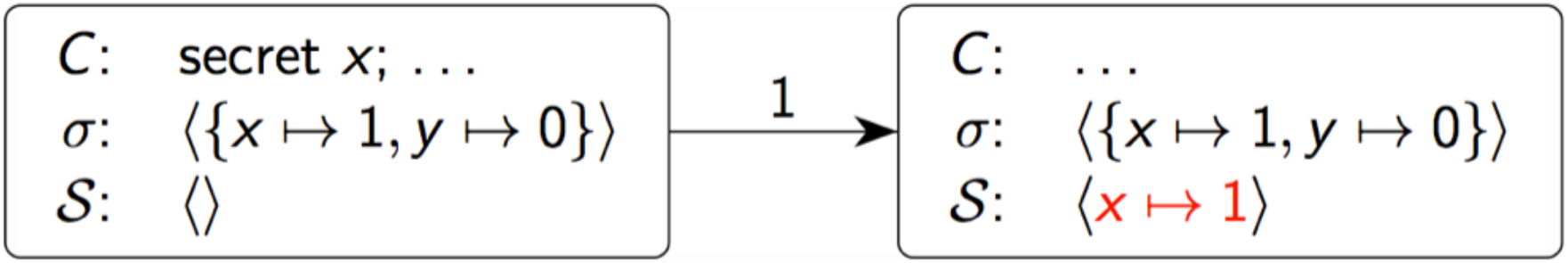
$$\frac{}{(\text{new } V := \rho;\ C, o :: \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{\rho(n)} (C, (\{V \mapsto n\} \cup o) :: \sigma, \mathcal{S}, \mathcal{O})}$$



# Secret Rules

N.B. A list of mapping
This reflects the
examples above.

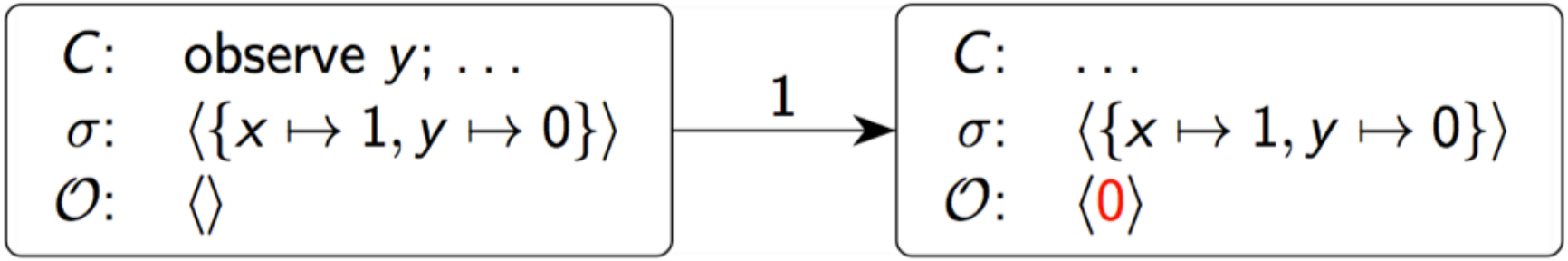
$$(\text{secret } V;\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S} :: (V \mapsto [\![V]\!]\sigma), \mathcal{O})$$



# Observation Rule:

**N.B. A list values**
**The attack only see the value, nothing else.**

$$\frac{}{(\text{observe } V;\ C, \sigma, \mathcal{S}, \mathcal{O}) \xrightarrow{1} (C, \sigma, \mathcal{S}, \mathcal{O} :: [\![V]\!]\sigma)}$$



# Leakage

- The semantics gives us a distribution on (S,O).
- We can then measure the leakage from S to O, i.e. what was an attacker learn about S from observing O?
- Popular measure of leakage include mutual information I and min-entropy leakage L:

$$I(X;Y) = \sum p(x,y) \log \left( \frac{p(x,y)}{p(x)p(y)} \right)$$

$$L(X;Y) = \log \sum_{Y} \max_{X} p(x,y) - \log \max_{X} \sum_{Y} p(x,y)$$

# Do we actually want to measure?

The "***worst possible program***" is one in which every "secret" is replaced with "print to name and value to attacker".

- i.e. attacker sees the exact values we are trying to hide.

**Theorem**: Our semantics measures how much an attacker learns about this "worst possible program" by observing the real program.

**Proof**: Define semantics for this "worst possible program" & induction on the semantic rules.

# An Infinite Number of Outputs (Finite Secrets)

- We have now defined leakage for finite inputs and outputs, but not infinite.

- Does it make sense to measure leakage to a infinite number of outputs?

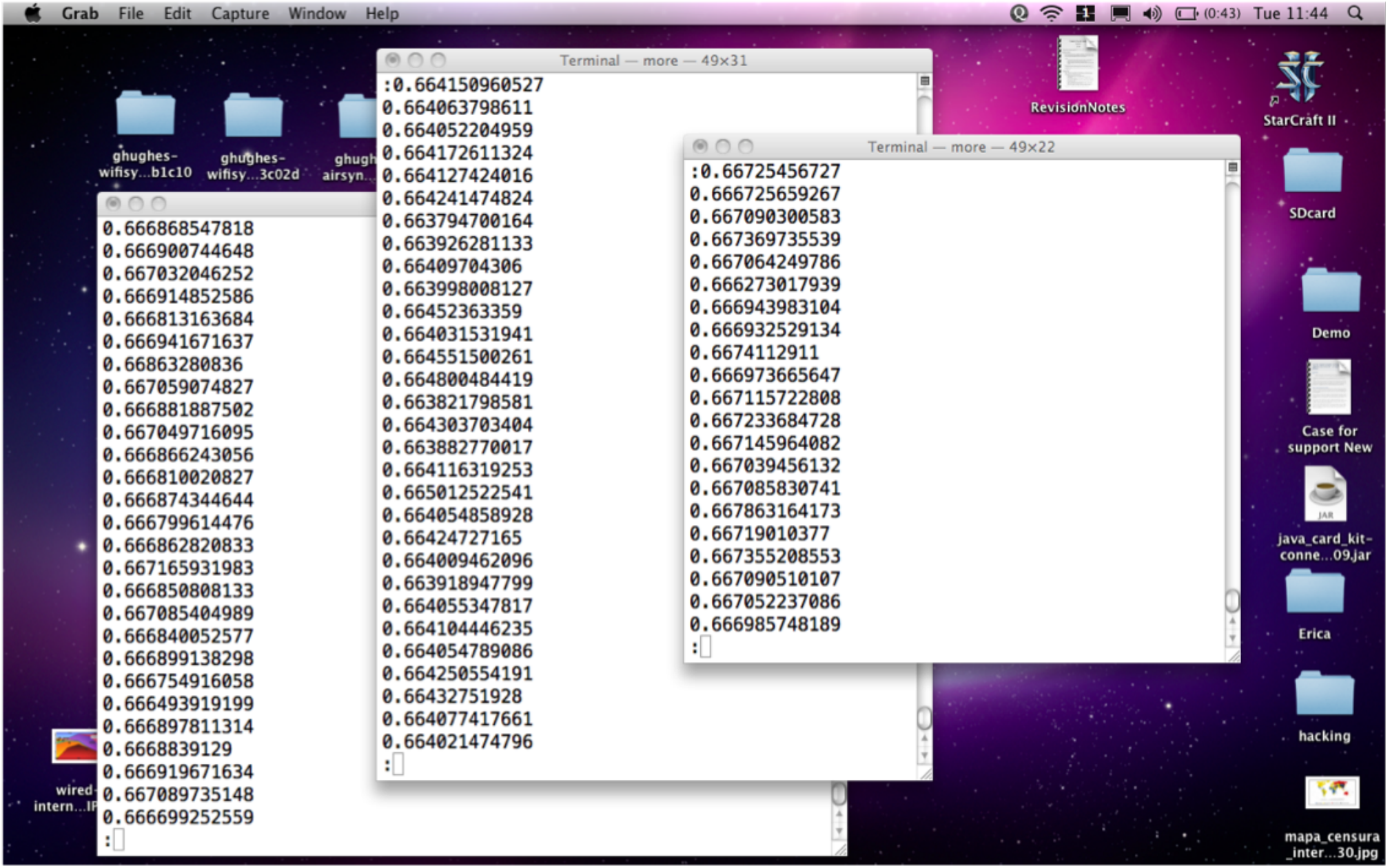
# An Infinite Number of Outputs (Finite Secrets)

Define $Sec^n$ and $Obs^n$ to be the distribution on secrets and observations after n steps (may be empty).

We can define the leakage as:

$$Leakage = \lim_{n \to \infty} I(Sec^n, Obs^n)$$

Bounded above (finite secrets) and increasing therefore converges.


Grab File Edit Capture Window Help
(0:43) Tue 11:44
Terminal — more — 49×31
:0.664150960527
0.664063798611
0.664052204959
0.664172611324
0.664127424016
0.664241474824
0.663794700164
0.663926281133
0.66409704306
0.663998008127
0.66452363359
0.664031531941
0.664551500261
0.664800484419
0.663821798581
0.664303703404
0.663882770017
0.664116319253
0.665012522541
0.664054858928
0.66424727165
0.664009462096
0.663918947799
0.664055347817
0.664104446235
0.664054789086
0.664250554191
0.66432751928
0.664077417661
0.664021474796
:
Terminal — more — 49×22
:0.66725456727
0.666725659267
0.667090300583
0.667369735539
0.667064249786
0.666273017939
0.666943983104
0.666932529134
0.6674112911
0.666973665647
0.667115722808
0.667233684728
0.667145964082
0.667039456132
0.667085830741
0.667863164173
0.66719010377
0.667355208553
0.667090510107
0.667052237086
0.666985748189
:
0.666868547818
0.666900744648
0.667032046252
0.666914852586
0.666813163684
0.666941671637
0.66863280836
0.667059074827
0.666881887502
0.667049716095
0.666866243056
0.666810020827
0.666874344644
0.666799614476
0.666862820833
0.667165931983
0.666850808133
0.667085404989
0.666840052577
0.666899138298
0.666754916058
0.666493919199
0.666897811314
0.6668839129
0.666919671634
0.667089735148
0.666699252559
:
ghughes-wifisy...b1c10
ghughes-wifisy...3c02d
ghugh airsyn...
RevisionNotes
StarCraft II
SDcard
Demo
Case for support New
java_card_kit-conne...09.jar
Erica
hacking
wired intern...IF
mapa_censura_inter...30.jpg

# Continuous Mutual Information

$$I(X;Y) = \sum_x \sum_y p(x,y)\, log\left(\frac{p(x,y)}{p(x)p(y)}\right)$$

- We let the resolution of y tented to 0

$$I(X;Y) = \sum_x \int_y p(x,y)\, log\left(\frac{p(x,y)}{p(x)p(y)}\right) dy$$

- Attacker with arbitrary accuracy:
  - Information leak: how much easier it is to guess after observing the system

# Kernel Estimation

We estimate the value of p( _ | x ) at y by looking at each observation and deciding how much that should affect the estimate

$$\hat{p}(y \mid x) = \frac{1}{N.h} \sum_{x} K\left(\frac{Y_i - y}{h}\right)$$

Epanechnikov kernel:

$$K(u) = \frac{3}{4}(1 - u^2)\chi\{|u| \le 1\}$$

h = bandwidth

$$h = 1.06 \times SD(Y) \times N^{-\frac{1}{5}}$$

# Lebesgue vs Riemann Integration

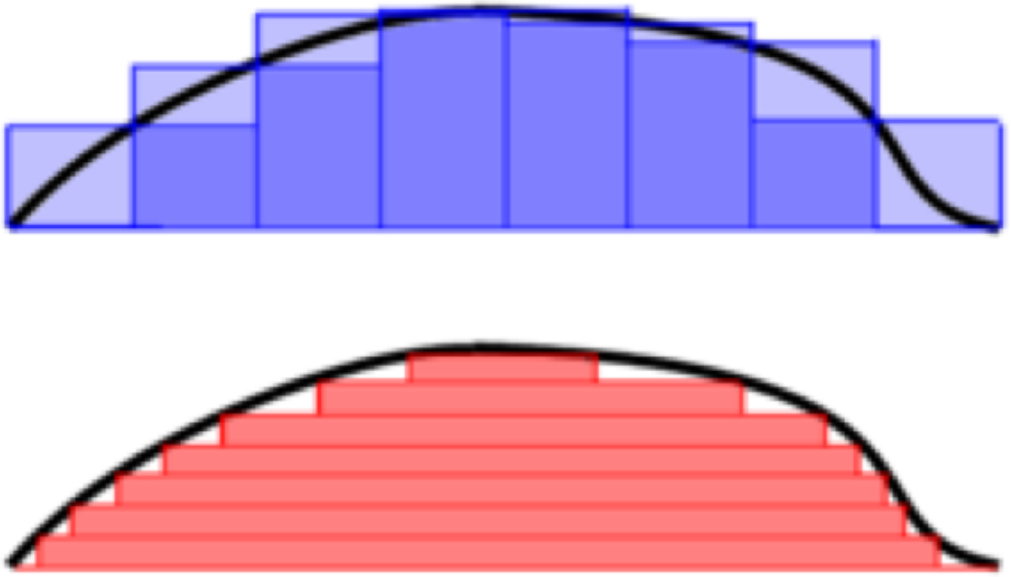
We know how to reason about leakage from the infinite domain of real numbers, using Riemann Integration:

$$I(X;Y) = \int_x \int_y p(x,y) \left( \frac{p(x)p(y)}{p(x,y)} \right) \; dx \; dy$$

but the output of a non-terminating program is not Riemann Integrable ☹

# Lebesgue vs Riemann Integration

- But, the output of a non-terminating program is Lebesgue Integrable.



stolen from wikipedia

# Lebesgue vs Riemann Integration

We define the leakage of a non-terminating program as the Lebesgue integral of the mutual information:

$$I(X;Y) = \int \log\left(\frac{dP_{XY}}{d(P_X \times P_Y)}\right) dP_{XY}$$

Radon-Nikodym derivative

# An Infinite Number of Secret Values

For infinite secrets values we use a rate:

$$\lim_{n\to\infty} \frac{1}{\#secrets} \sum_x \sum_y p^n(x,y)\, log\left(\frac{p^n(x,y)}{p^n(x)p^n(y)}\right)$$

"#secrets" is the number of secret values defined so far.

Pros: a useful measure.

Cons: Doesn't always converge,
Doesn't match finite version,
More secrets, less leakage.

# Summary

- AIM: calculate information leakage from Java Programs using statistical methods.
- 1st Step: formally define the leakage model.
- 2nd Step: statistical estimation results for probabilistic mutual information and min-entropy leakage.
- 3rd Step: a Java Framework
- RESULT: Estimation of information leakage for complex Java Programs (but small secret domains).

# Estimating mutual information

- We need to know how the estimated values of mutual information relates to the real value.
  - We have done this when p(x) is know and p(y|x) is estimated.

- Moddemejer (1989) and Brillinger (2004), have already done this for both X & Y unknown. Estimates $\hat{I}(X;Y)$ have:

**Mean**

$$I(X;Y) + \frac{(\#\mathcal{X}-1)(\#\mathcal{Y}-1)}{2n} + O\left(\frac{1}{n^2}\right)$$

**Variance**

$$\frac{1}{n}\left(\sum_{x,y}\hat{P}_{XY}(x,y)\log^2\left(\frac{\hat{P}_{XY}(x,y)}{\hat{P}_X(x)\hat{P}_Y(y)}\right) - \left(\sum_{x,y}\hat{P}_{XY}(x,y)\log\left(\frac{\hat{P}_{XY}(x,y)}{\hat{P}_X(x)\hat{P}_Y(y)}\right)\right)^2\right) + O\left(\frac{1}{n^2}\right)$$

# Collecting data from programs

- Java or C? We picked Java.
- Collecting samples: Rewrite “secret” and “observable” and execute the program many time??
  - No!!
- We found that start a JVM is VERY, VERY slow.
  - This stops calling “`java program`” many times from being practical.

# Collecting data from Java

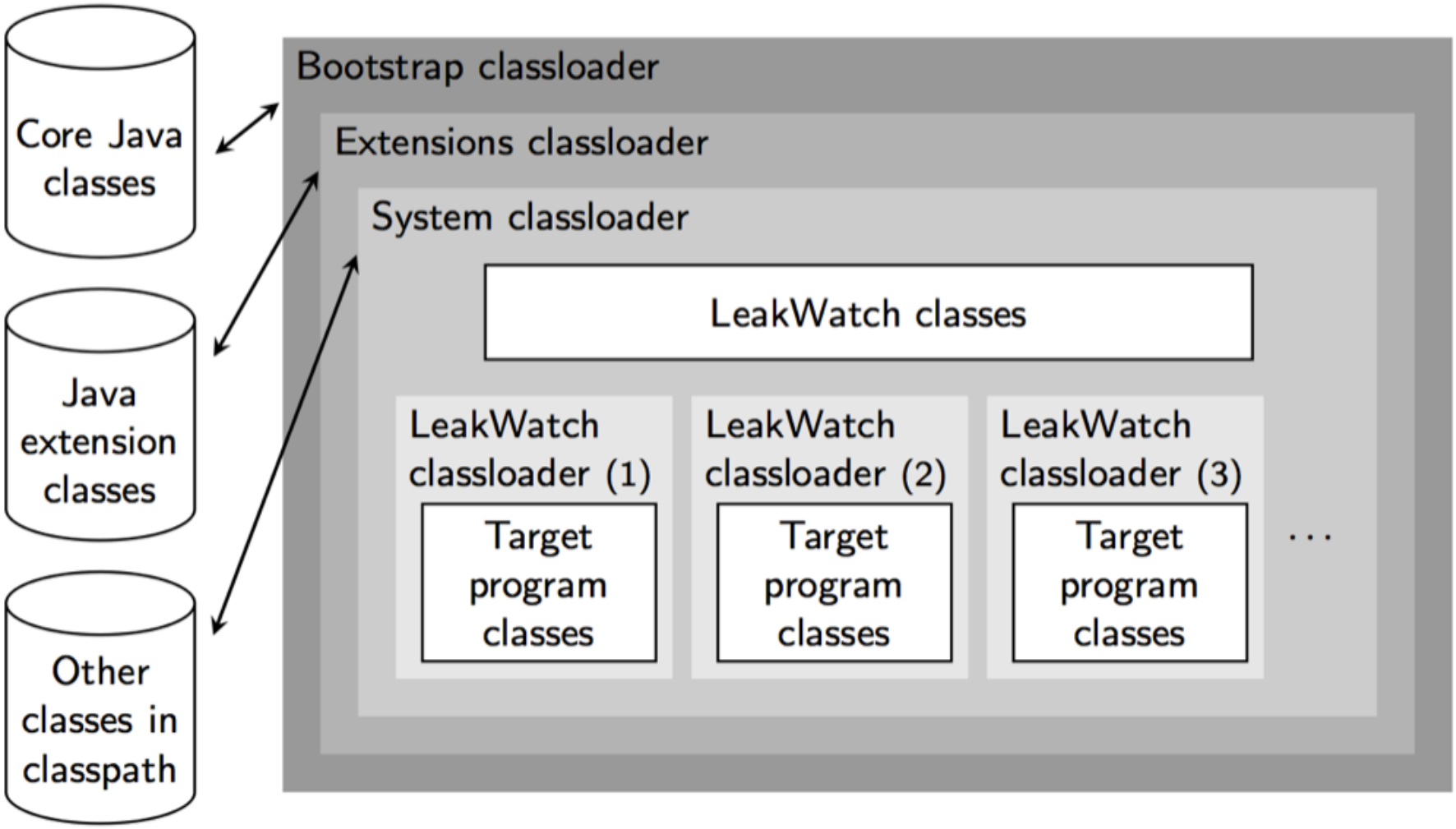
- So we used the Java Classloader to repeatedly load and execute the Java main method.

- Problem?
  - Java caches the Object that contains the main method.
  - Static variables are presented from one run to the next.

# LeakWatch

- We have written our own custom Java classloader.
- Class being tested are always reload (not cached).
- System classes with no static values are cached.
- Copies of the test program are run in parallel to take advantage of multicore machines.

# Leakwatch Overview



# Conclusion

- We adapted the classic information leakage model to allow us to tag any variables in a program as “observable” or “secret”.
  - Correlation, Mutual information, Min-entropy leakage, g-leakage
- We built a formal model that measures what the observable values tell us about the secrets.
- We discussed leakage from non-terminating programs.
- LeakWatch: a Framework for statistical measuring leakage from Java programs.