Secure Information Flow Analysis: A Science of Hacking?

Geoffrey Smith

School of Computer Science

Florida International University

Miami, Florida USA

Privacy and the Internet

"Snoop Software Gains Power and Raises Privacy Concerns"
New York Times, 10/10/03

Keystroke loggers monitor a target computer, recording all keystrokes (credit card numbers, passwords, . . .).

Said to be common on public terminals (airports, Kinko's, . . .).

"Silent deploy" lets the logger be installed via an e-mail!

Standard Techniques for Privacy

- **Access control**

  Controls *who* can directly access information.

  Does not control what they do with it!

- **Encryption**

  Secures communication channels.

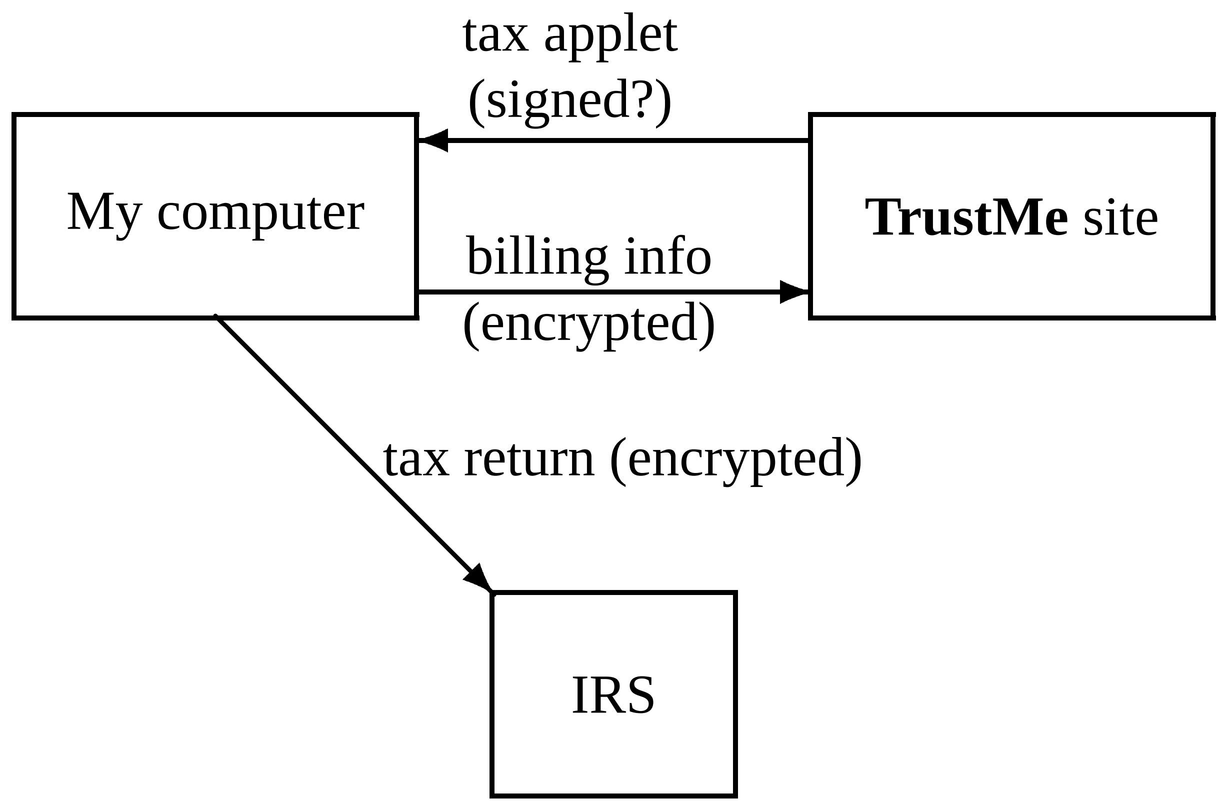
  Does not prevent the receiver from leaking the information.

- **Digital signatures**

  Tells you *who* wrote a program.

  Doesn’t ensure that they are trustworthy!

Example: A Tax Return Applet



Secure Information Flow

We want an *end-to-end* privacy property.

Classify program variables as

- L (low, public) or
- H (high, private).

(Or a richer classification: $U < C < S < TS$.)

Want to prevent the program from *leaking* any information from H variables to L variables.

Some Hacking

Assume `secret` is H and `leak` is L.

An explicit flow:

```
leak = secret;
```

Two permissible flows:

```
secret = leak;

leak = 76318;
```

An *implicit* flow:

```
if (secret % 2 == 0)
  leak = 0;
else
  leak = 1;
```

A Multi-Threaded Example

Assume `trigger` is initially 0.

Thread *Odd*:

```
if (secret % 2 == 0)
  while (trigger == 0)
    ;
leak = 0;
trigger = 1;
```

Thread *Even*:

```
if (secret % 2 == 1)
  while (trigger == 0)
    ;
leak = 1;
trigger = 1;
```

An Example Using Out-of-Bounds Array Indexing

Assume `leak` is initially 0.

Thread b, where $0 \leq b < 10$:

```
int [] a = new int [1];
a[1 - (secret >> b) % 2] = 0;
leak |= (1 << b);
```

Once all the threads terminate or abort, `leak` contains a copy of the 10-bit `secret`.

An Example Using Timing

Thread α:

```
while (secret > 0)
  secret--;
leak = 2;
```

Thread β:

```
Math.sqrt(2.0);
leak = 1;
```

Assuming a probabilistic thread scheduler, there is a probabilistic flow: the larger the initial value of `secret`, the larger the probability that the final value of `leak` is 2.

An Example using Caching [Agat 00]

```
int i, count, xs[4096], ys[4096];

for (count = 0; count < 100000; count++) {
  if (secret)
    for (i = 0; i < 4096; i += 2)
      xs[i]++;
  else
    for (i = 0; i < 4096; i += 2)
      ys[i]++;
  for (i = 0; i < 4096; i += 2)
    xs[i]++;
}
```

On Sparc server `goliath` (with a 16K data cache), this takes about twice as long when `secret` is 0 as it takes when `secret` is 1!

Noninterference [Goguen and Meseguer 82]

What does it mean to say “Program P leaks no information from H variables to L variables”?

Idea: If we run P twice, on initial memories that differ only on the values of H variables, then an “L observer” shouldn’t be able to tell the difference.

Def: Memories μ and ν are L-*equivalent*, written $\mu \sim_L \nu$, if μ and ν agree on the values of all L variables.

Noninterference Property: If $\mu \sim_L \nu$, then $(P, \mu) \sim_L (P, \nu)$.

But what should $(P, \mu) \sim_L (P, \nu)$ mean?

It depends on what is L-observable.

Possible L-observations:

- The final values of L variables.
- Whether or not the program aborts.
- Whether or not the program terminates.
- The program's running time.
- The number of page faults during the program's execution.
- ...

Of course, the more allowed L-observations there are, the harder it is to prevent leaks.

(Especially nasty is that leaks may occur at a lower level than our formal semantics.)

A further complication is that with multi-threading the program may be *nondeterministic* or *probabilistic*.

Static Analyses for Secure Information Flow

Idea: Analyze programs statically before executing them. Programs that pass the analysis can be executed safely.

(But because the analysis must be conservative, some safe programs will necessarily be rejected.)

Pioneered by [Denning and Denning 77].

[Volpano, Smith, and Irvine 96]:

- Denning's analysis can be formulated as a type system.
- Denning's analysis guarantees a noninterference property.

Much work has followed: A recent survey [Sabelfeld and Myers 03] includes about 150 references.

Probabilistic Noninterference

Focus on the multi-threaded case with a uniform probabilistic scheduler.

Limit L observations to the final values of L variables.

(But multi-threading makes *abstract* time observable internally!)

Probabilistic Noninterference property:

> Changing the initial values of H variables does not change the *joint distribution* of possible final values of L variables.

Disallowing observations of real running time makes it much easier to prevent leaks.

Reasonable in the case of mobile code, since we can control what an outsider can observe about mobile code running on our computer.

A Type System for Probabilistic Noninterference

[Smith 01]:

Classify and *restrict* expressions and commands.

Classifications:

1. An expression e is H if it contains H variables; otherwise it is L.
2. A command c is $\boxed{\tau_1 \ \mathit{cmd} \ \tau_2}$ if it assigns only to variables of type τ_1 (or higher) and its running time depends only on variables of type τ_2 (or lower).
3. A command c is $\boxed{\tau \ \mathit{cmd} \ n}$ if it assigns only to variables of type τ (or higher) and it terminates in exactly n steps.

Restrictions:

1. An H expression cannot be assigned to an L variable.

   (No explicit leaks.)

2. A guarded command with H guard cannot assign to L variables.

   (No implicit leaks.)

3. A command whose running time depends on H variables cannot be followed sequentially by a command that assigns to L variables.

   (No timing leaks.)

(A very similar type system was developed independently by [Boudol and Castellani 01].)

Example Typings

`secret = 0;` : *H cmd* 1

`leak = 0;` : *L cmd* 1

`if (leak == 0) leak = 5 else leak = 6` : *L cmd* 2

`while (leak == 0)` { } : *H cmd L*

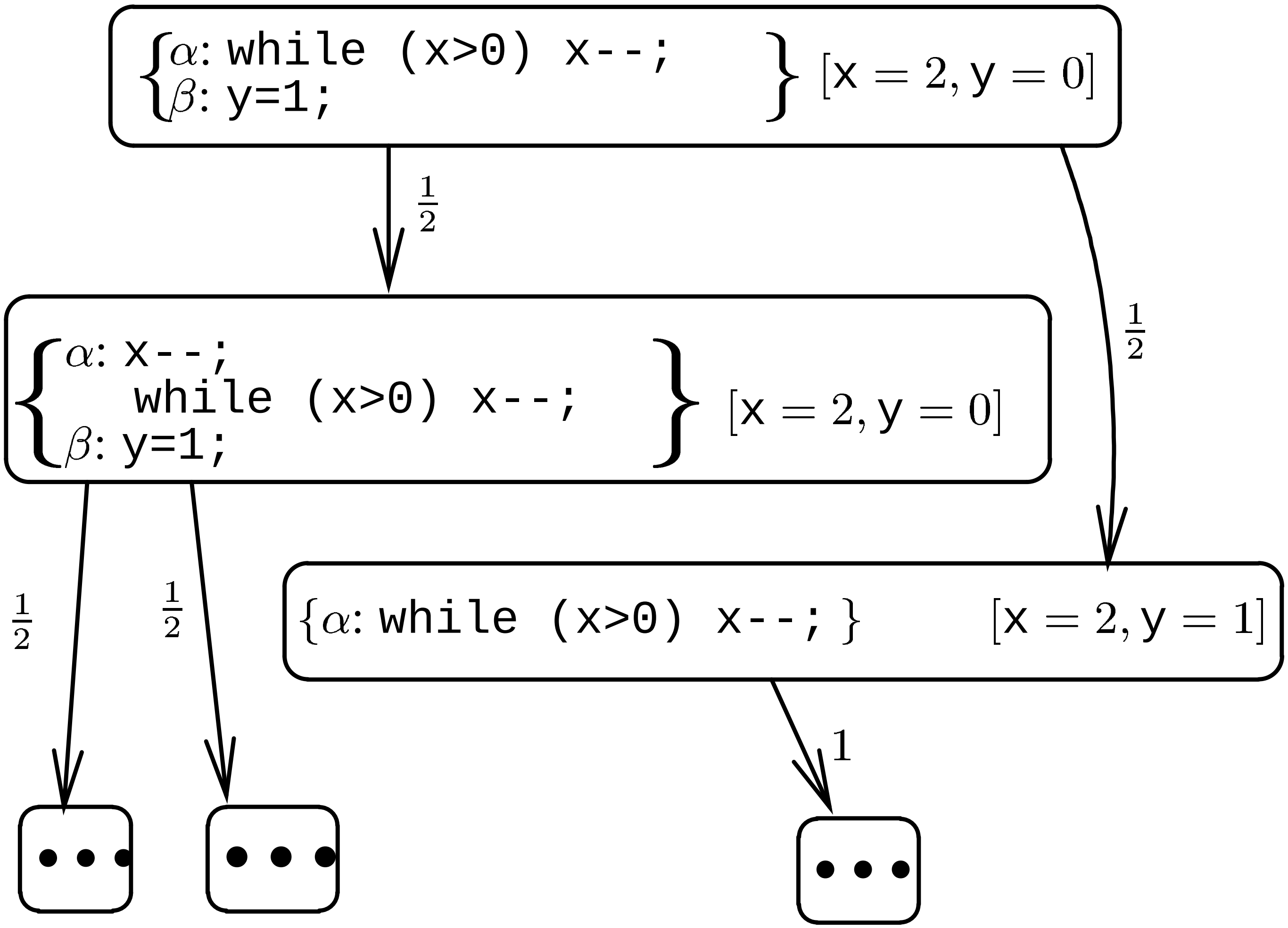
`while (leak == 0) leak -= 1;` : *L cmd L*

`leak = 5; while (secret == 0)` { } : *L cmd H*

`if (secret == 0) leak = 1;` : **illegal**

`while (secret == 0)` { }`; leak = 5;` : **illegal**

Multi-Threaded Programs as Markov Chains



Soundness of Type System

Key property for commands:
If a well-typed command c is run under two L-equivalent memories, it makes exactly the same assignments to L variables, at the same times.

But it need not *terminate* at the same time!

Idea for multi-threaded programs:
If program P is well typed and $\mu \sim \nu$, then the Markov chains starting from (P, μ) and from (P, ν) are (somehow) L-*equivalent*.

But threads may terminate at different times under μ and ν!

Probabilistic Bisimulation on Markov Chains

Given a Markov chain with state set S and transition probabilities p_{ab} for $a, b \in S$.

Let $\approx$ be an equivalence relation on S.

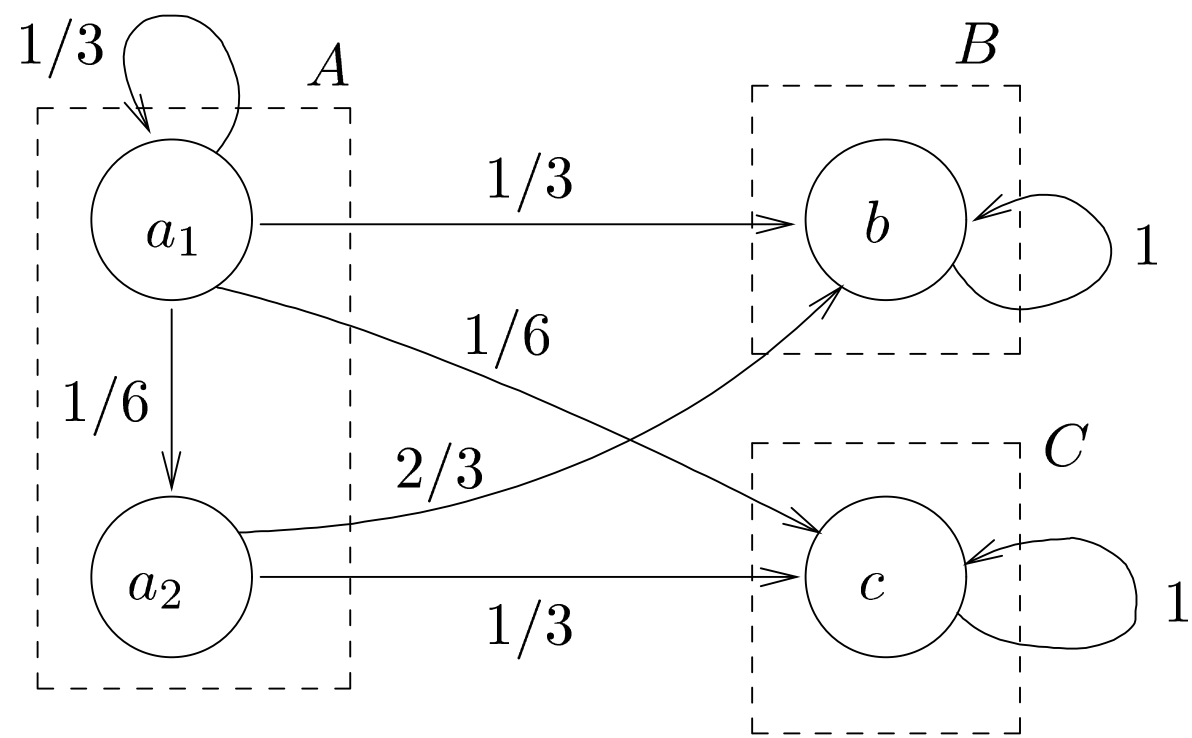
Definition: $\approx$ is a *probabilistic bisimulation* if

$a_1 \approx a_2$ implies

$$\sum_{b \in B} p_{a_1 b} = \sum_{b \in B} p_{a_2 b}.$$

[Kemeny Snell 60, Larsen Skou 91, Sabelfeld Sands 00]

Strong with respect to timing—from a_1 and a_2 we pass through the same equivalence classes *at the same times*.



$\approx$ is *not* a probabilistic bisimulation:

- a_1 goes to B (in one step) with probability $1/3$.
- a_2 goes to B (in one step) with probability $2/3$.

But, *abstracting away from time*, both a_1 and a_2 go to B with probability $2/3$, possibly after some "stuttering" within A.

Weak Probabilistic Bisimulation on Markov Chains

Let A and B be distinct equivalence classes, $a \in A$.

$\mathcal{P}(a, A, B)$ is the probability of starting at a, *moving for 0 or more steps within* A, and then entering B.

Definition: $\approx$ is a *weak probabilistic bisimulation* if

$a_1 \approx a_2$ implies

$$\mathcal{P}(a_1, A, B) = \mathcal{P}(a_2, A, B).$$

Appropriate if we are interested in the sequence of equivalence classes that are visited, but not how long the chain remains in each class.

([Baier and Hermanns 97] define a similar notion for a process algebra.)

Applying Weak Probabilistic Bisimulation

[Smith 03]:

We can define a weak probabilistic bisimulation $\sim_L$ on well-typed configurations (P, μ).

And $\sim_L$ respects the L-observability of L variables:

If $(P, \mu) \sim_L (P', \nu)$, then $\mu \sim_L \nu$.

This implies that our type system guarantees probabilistic noninterference.

Challenges

- *Type Inference.*

  We'd like to infer the security class of "internal" variables.

- *Enrich the language.*

  Consider arrays, exceptions, object-oriented features, . . .

- *False positives.*

  Is the analysis too conservative to be usable?

- *Limited leaks.*

  What if you *want* to leak a little information?

  (Password checkers, average salaries, . . .)

Collaborators

Dennis Volpano (Naval Postgraduate School, Cranite Systems)

Cynthia Irvine (Naval Postgraduate School)

Ryan Yocum (FIU master's thesis 02)

Zhenyue Deng (Current PhD student)

Daniel Cazalis (Current PhD student)

Supported by NSF grants CCR-9596113, CCR-9612176, CCR-9900951.