Information Flow Security

Oren Freiberg Shu-yu Guo

University of California, Los Angeles

October 21, 2010

Controlling Information

Problem

Information is power. Information now lives on computers. It goes to places we don't want it to go. How do we control information?

We want:

- End-to-end
- To control *propagation*

What's wrong with existing technologies?

- Access controls control the release of sensitive information, not propagation. Once it's out, it's out.
- Firewalls and antivirus are not end-to-end.
- Covert channels
  - Implicit flow
  - Termination
  - Timing
  - Probablistic

What's wrong with existing technologies?

- Access controls control the release of sensitive information, not propagation. Once it's out, it's out.
- Firewalls and antivirus are not end-to-end.
- Covert channels
  - Implicit flow
  - Termination
  - Timing
  - Probablistic

What's wrong with existing technologies?

- Access controls control the release of sensitive information, not propagation. Once it's out, it's out.
- Firewalls and antivirus are not end-to-end.
- Covert channels
  - Implicit flow
  - Termination
  - Timing
  - Probablistic

What's wrong with existing technologies?

- Access controls control the release of sensitive information, not propagation. Once it's out, it's out.
- Firewalls and antivirus are not end-to-end.
- Covert channels
  - Implicit flow
  - Termination
  - Timing
  - Probablistic

Programming Languages

A promising approach has been to integrate information flow primitives into programming languages: now we have guarantees at the *language* level that information “flows to the right places”.

Example: Explicit Flow

```
var h = getSecret();
var l = h; // oops
```

Example: Implicit Flow

```
var h = getSecret();
var l = 0;
if (h == 0) {
  l = 1;
}
```

l *strongly depends* on h. By observing the value of l, we know the value of h, whether the conditional is executed or not.

Example: Implicit Flow

```
var h = getSecret();
var l = 0;
if (h == 0) {
  l = 1;
}
```

l *strongly depends* on h. By observing the value of l, we know the value of h, whether the conditional is executed or not.

Policy

Question

How do we specify information flow policies for confidentiality?

State of the Art

Distributive lattice of labels

- Simple
- Nice, well-known properties

Lattice

$$\begin{tikzcd}
 & \{med, fin, crim\} & \\
\{med, fin\} \arrow[ur] & \{med, crim\} \arrow[u] & \{fin, crim\} \arrow[ul] \\
\{med\} \arrow[u] \arrow[ur] & \{fin\} \arrow[ul] \arrow[ur] & \{crim\} \arrow[ul] \arrow[u] \\
 & \bot \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Lattice

$$\begin{tikzcd}H \\ L \arrow[u]\end{tikzcd}$$

$$L \sqsubseteq H$$

L can flow to H, i.e. information can be read *from* low security variables and go into a high-security variable, but not vice versa.

Lattice

$$\begin{tikzcd} H \\ L \arrow[u] \end{tikzcd}$$

$$L \sqsubseteq H$$

L can flow to H, i.e. information can be read *from* low security variables and go into a high-security variable, but not vice versa.

Noninterference

With any formal model, the end goal is to prove properties. In this case, we want to prove some kind of property that information doesn't flow to the wrong places.

Noninterference

Noninterference

$$\forall p_1, p_2 . p_1 =_L p_2 \wedge p_1 \mapsto^* p_1' \wedge p_2 \mapsto^* p_2' \Rightarrow p_1' =_L p_2'$$

In English, it says that if the "low view" of two programs are equivalent, then during execution their "low views" should always be equivalent. That is, high-security information should not interfere with low-security information.

Implicit Flow Revisited

```
var h = getSecret();
var l = 0;
if (h == 0) {
  l = 1;
}
```

This program is not non-interfering: observable low-security data depends on unobservable high-security data.
Easy to prove statically, hard to so dynamically. (Anyone see why?)

Duality

Confidentiality

Preventing information from flowing *from* bad places. No read up.

Integrity

Preventing information from flowing *to* bad places. No write down.

Dual to each other: the integrity lattice is the confidentiality lattice upside down.

Integrity

- Instead of private and public information, the basic lattice for integrity consists of *tainted* and *untainted* data.
- Important difference from confidentiality: integrity can be compromised without interaction with the external world, simply by having bugs. Correctness is hard to prove.

Challenges in Languages

Complexity	Nobody knows how to write flow policies because they are bewilderingly complicated and principals aren't usually static. Shouldn't be this hard.
Too Strict	Noninterference is too strict a property to be useful. Systems need legitimate ways of declassifying, and dually, endorsing data. How do we do this right?
Dynamic	Kinda hard and inefficient.

Security of Entire Systems

Security, after all, is a property of *entire systems*, not just programming languages.
Plus, nobody knows how to actually use information flow-safe languages, so what do we do?

Abstract it to the operating system.

Information Flow in OS

- Information flow is controlled at process and thread boundaries.
- Can use the same lattices and theory that languages research has developed.
- Untrusted program will cause minimal damage since the operating system will be enforcing security policies.

Example: HiStar. Integrates both confidentiality and integrity policies into the operating system.

Information Flow in OS

- Information flow is controlled at process and thread boundaries.
- Can use the same lattices and theory that languages research has developed.
- Untrusted program will cause minimal damage since the operating system will be enforcing security policies.

Example: HiStar. Integrates both confidentiality and integrity policies into the operating system.

Information Flow in OS

- Information flow is controlled at process and thread boundaries.
- Can use the same lattices and theory that languages research has developed.
- Untrusted program will cause minimal damage since the operating system will be enforcing security policies.

Example: HiStar. Integrates both confidentiality and integrity policies into the operating system.

Information Flow in OS

- Information flow is controlled at process and thread boundaries.
- Can use the same lattices and theory that languages research has developed.
- Untrusted program will cause minimal damage since the operating system will be enforcing security policies.

Example: HiStar. Integrates both confidentiality and integrity policies into the operating system.

Benefits of Information Flow in OS

- Requires minimal code changes to enforce security policies.
- Eliminates the need for superusers on the OS, reducing risks to arbitrary user data.
- Authentication of users requires no highly-trusted process, minimizing leaked data.
- Allows for VPN isolation since incoming data is tainted.

Benefits of Information Flow in OS

- Requires minimal code changes to enforce security policies.
- Eliminates the need for superusers on the OS, reducing risks to arbitrary user data.
- Authentication of users requires no highly-trusted process, minimizing leaked data.
- Allows for VPN isolation since incoming data is tainted.

Benefits of Information Flow in OS

- Requires minimal code changes to enforce security policies.
- Eliminates the need for superusers on the OS, reducing risks to arbitrary user data.
- Authentication of users requires no highly-trusted process, minimizing leaked data.
- Allows for VPN isolation since incoming data is tainted.

Benefits of Information Flow in OS

- Requires minimal code changes to enforce security policies.
- Eliminates the need for superusers on the OS, reducing risks to arbitrary user data.
- Authentication of users requires no highly-trusted process, minimizing leaked data.
- Allows for VPN isolation since incoming data is tainted.

Challenges in Systems

Complexity	Still hard to use. Nobody still knows how to make the right policy.
Persistency	Have to maintain persistency across restarts.
Performance	Performance overheads are now added to processes and threads.

Discussion