Software Security

Martín Abadi

FOSAD 2012

Plan

- Introduction to access control
- Access control and programs
- Mandatory access control and information flow control

Plan

- Introduction to access control
- Access control and programs
- Mandatory access control and information flow control
- Introduction to low-level software security
- Various mitigations and defenses: canaries, NX, layout randomization, CFI and SFI
- Security in programming languages

Access control

Access control

Access control is prominent at many levels:

- memory-management hardware,
- operating systems, file systems, and the like,
- middleware,
- applications,
- firewalls,

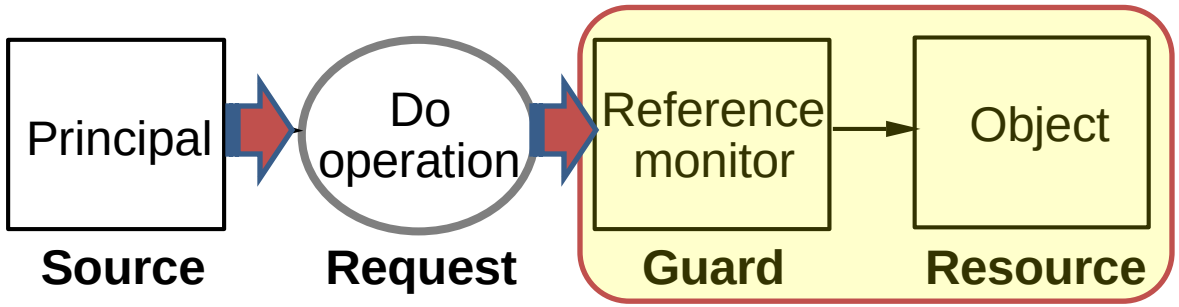
and also in physical protection.

Access control (cont.)

- Access control is a **mechanism**.
  - It aims to guarantee secrecy, integrity, and availability properties, and more.
- Access control can also be seen as a **model**, as specification for lower-level mechanisms.
  - (Higher-level policies are often not explicit.)

The access control model

- Elements:
  - Objects or resources
  - Requests
  - Sources for requests, called principals (or subjects)
  - A reference monitor to decide on requests



An access control matrix

[Lampson, 1971]

objects / principals	file1	file2	file3	file4
user1	rwx	rw	r	x
user2	r	r		x
user3	r	r		x

Strategies for representing an access control matrix

In practice, a matrix is typically represented in terms of ACLs and capabilities.

- ACL: a column of an access control matrix, attached to an object.
- Capability: (basically) a pair of an object and an operation, for a given principal.
  It means that the principal may perform the operation on the object.

Implementing capabilities

⇒ *Principals should not be allowed to forge capabilities.*

This leads to implementations of capabilities

- stored in a protected address space, or
- with special tags with hardware support, or
- as references in a typed language, or
- with a secret, or
- with cryptography, e.g., certificates.

The reference monitor and mediation

The principle of complete mediation

[Saltzer and Schroeder, 1975]

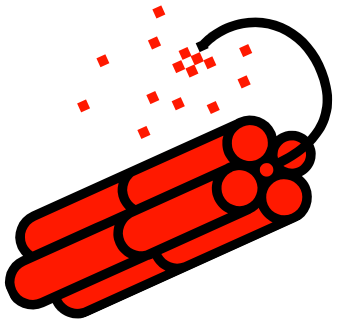
Every access to every object must be checked for authority.

This principle can be enforced in several ways:

- The OS intercepts some of the requests. The hardware catches others.
- A software wrapper / interpreter intercepts some of the requests. (E.g., as in VMs.)

Common dangers

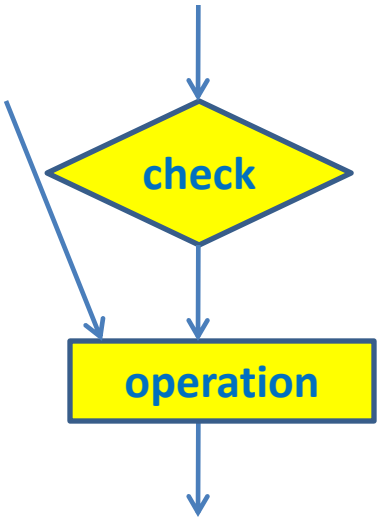
- Access control can be insufficient or irrelevant
  - when it is implemented incorrectly,
  - when the underlying operations are implemented incorrectly,
  - when the policy is wrong,
  - when it is circumvented.



Circumventing access control

Sometimes the reference monitor does not protect all important objects and operations, for example because of

- hostile platforms (e.g., for DRM systems),
- control-flow subversions (as we will see),
- race conditions,
- data recovery from memory or disks,
- side channels.



The promise of formal verification

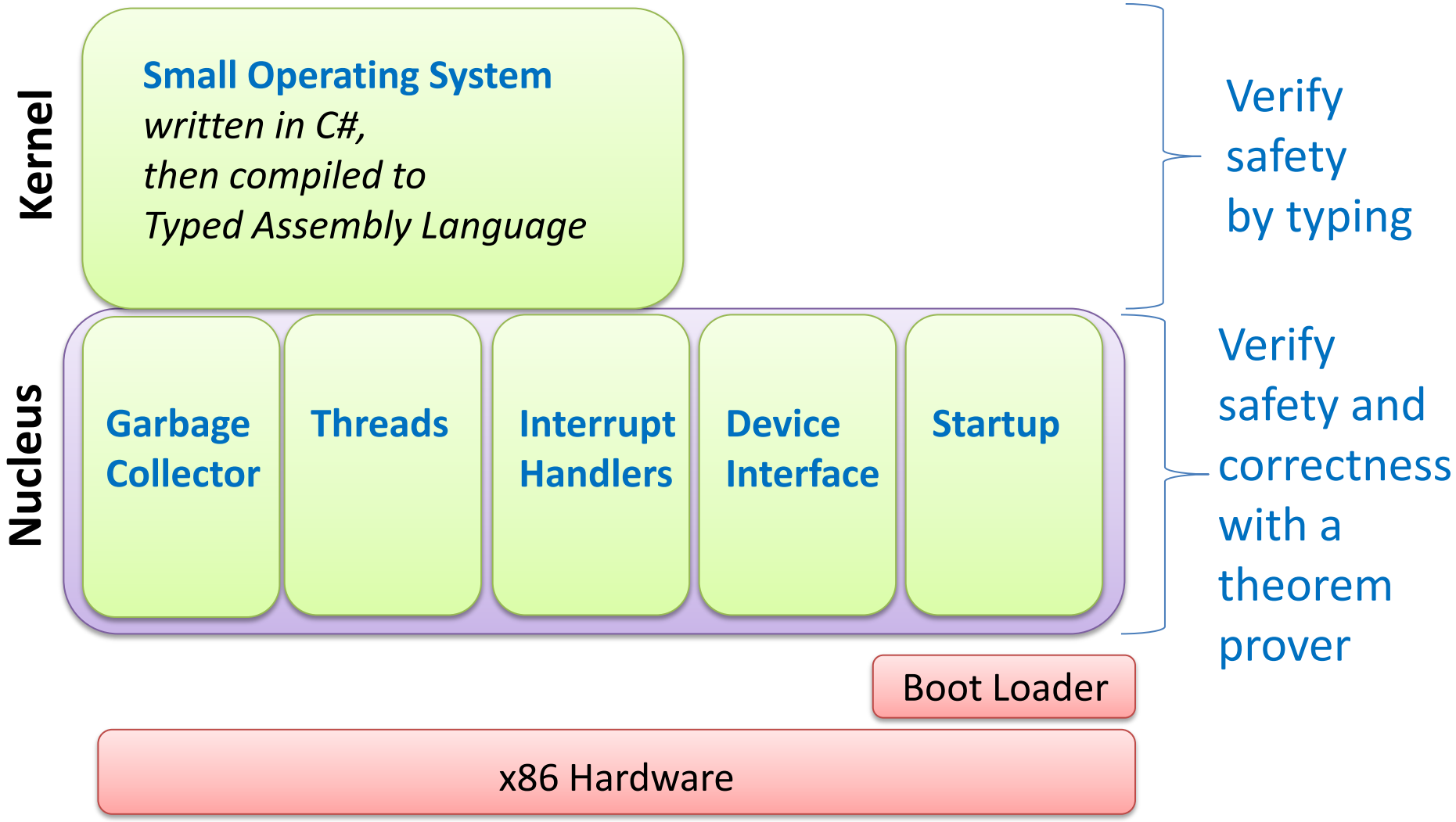
A formally verified security kernel is widely considered to offer the most promising basis for the construction of truly secure computer systems at least in the short term. A number of kernelized systems have been constructed and various models of security have been formulated to serve as the basis for their verification.
Despite the enthusiasm for this approach there remain certain difficulties and problems in its application […]

(Rushby, 1981)

A recent research system: Verve

[Yang and Hawblitzel]

"every assembly language instruction checked for safety"



Caveats

- Specifications are often incomplete or implicit, and typically rely on assumptions.
- Full verification remains hard and rare.
- Types help, but they are not a panacea.
- Legacy code and languages are problematic.

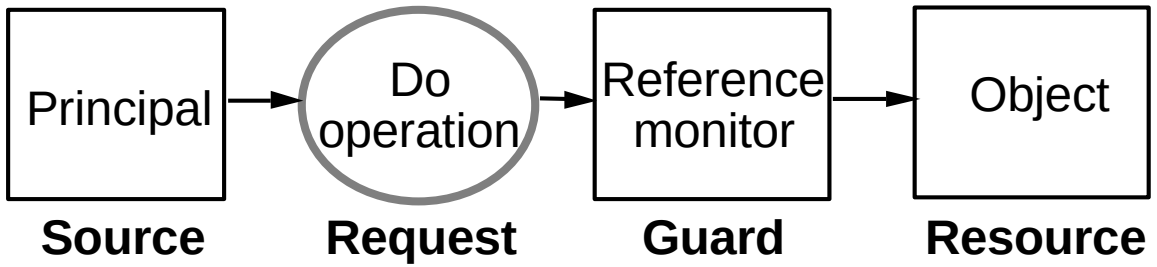
Still, we would like to be able to guarantee some basic security properties.

Software security: some approaches

- Avoiding software flaws:
  - Static analysis and proofs of correctness.
  - Safer programming languages and libraries.
- Reducing the impact of software flaws:
  - Various models and architectures.
  - Various run-time mitigation techniques.

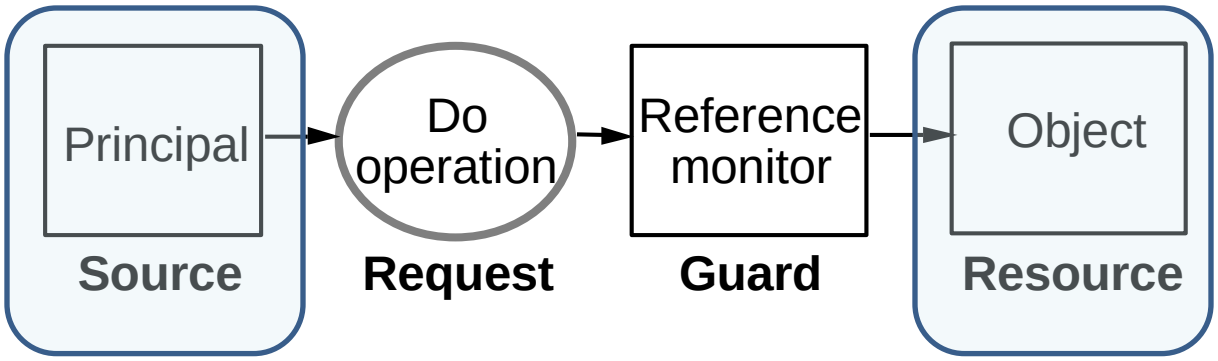
Access control and programs

Programs everywhere!



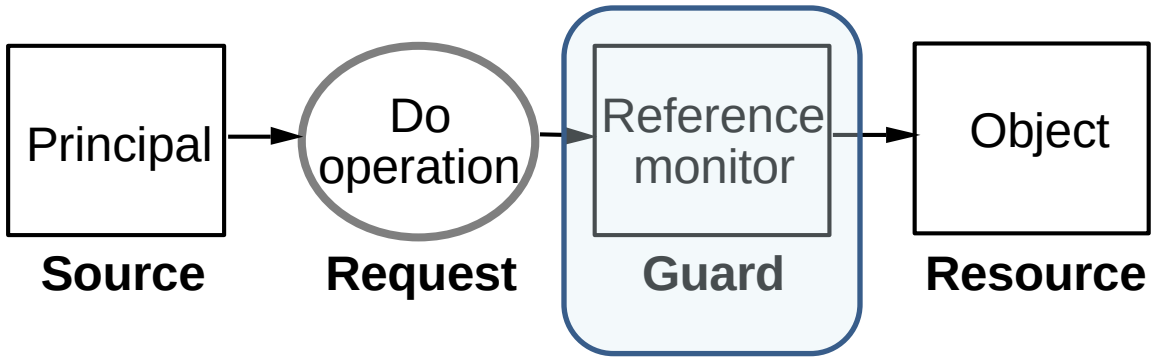
Programs everywhere!

- Programs are principals and objects.



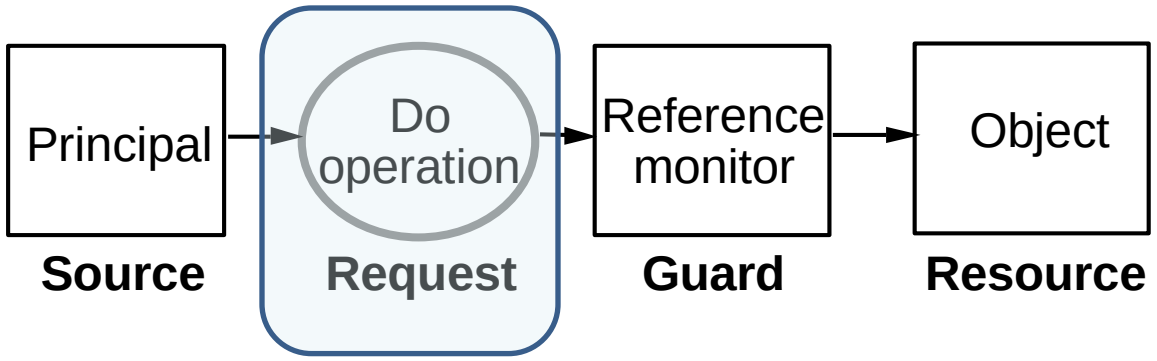
Programs everywhere!

- Programs are principals and objects.
- Programs perform the access control.
  - Often, even some of the access control policy is baked into programs, for better or for worse.



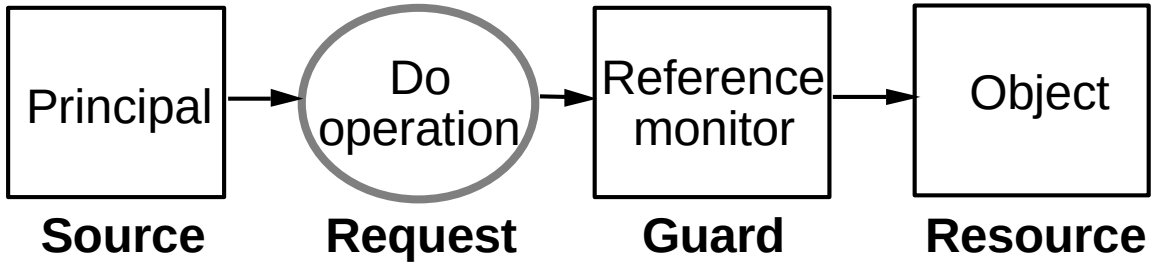
Programs everywhere!

- Programs are principals and objects.
- Programs perform the access control.
  - Often, even some of the access control policy is baked into programs, for better or for worse.
- Programs implement the operations that are the concern of access control.



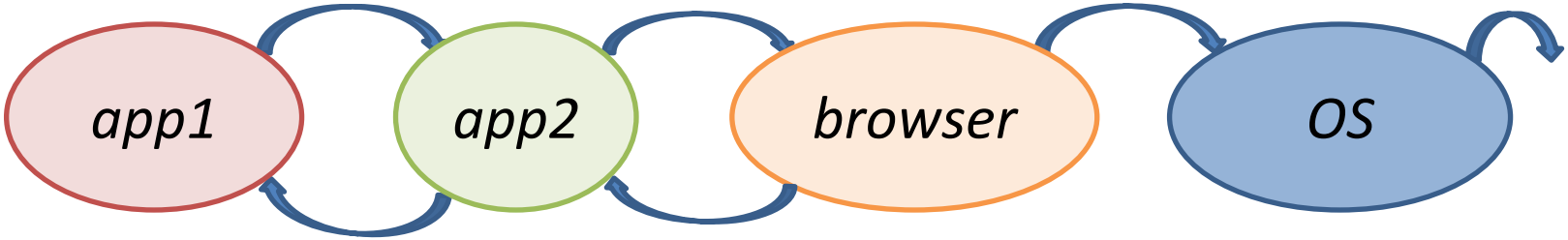
Programs everywhere!

- Programs are principals and objects.
- Programs perform the access control.
  - Often, even some of the access control policy is baked into programs, for better or for worse.
- Programs implement the operations that are the concern of access control.



Programs and other principals

- So, programs may be principals too.
- But then:
  - we need to deal with program combinations,



  - we need to connect programs to other principals
    - who write them or edit them,
    - who provide them or install them,
    - who call them.

Running programs

- What are the run-time rights of a program P?
  - those of P's caller, or
  - those of some responsible user, or
  - something else, e.g, because of P's properties, or
  - some combination.
- The same factors appear in deciding whether to run a program.

Running programs (cont.)

Some approaches to combining authorities:

- setuid,
- code access security (with stack inspection or alternatives).

Some approaches to intrinsic properties:

- proofs (and proof-carrying code),
- types,
- dynamic checks (e.g., in sandboxes),
- their combinations (e.g., proofs about sandboxes).

Protection and isolation

- Programs must be protected (always) and limited to communicate on proper interfaces.
- This is often the job of the computing platform (OS + hardware).
  - It can implement address spaces so that programs in separate spaces cannot interact directly (e.g., cannot smash or snoop on one another).
- A language and its run-time system can provide fine-grained control.

More on this later.

Examples

Access control in Unix (basics)

- Principals are users (plus root).
- Objects are files.
- Operations are read, write, and execute.
- Each file has an owner and a group.
- Each file has an ACL, which can be set by its owner and root.
- ACLs specify rights for the owner (“user”), group, and others (e.g., **rwxrw-r--**).

Access control in Unix (cont.)

- If a program file is marked as suid, then the program executes with the privilege of its owner (not that of the caller).
  - The usage of setuid is error-prone.
  - The details are complex and vary across systems.
- And there are other complications: sgid, capabilities in Linux, directories, …

See "Setuid Demystified", by Chen, Wagner, and Dean.

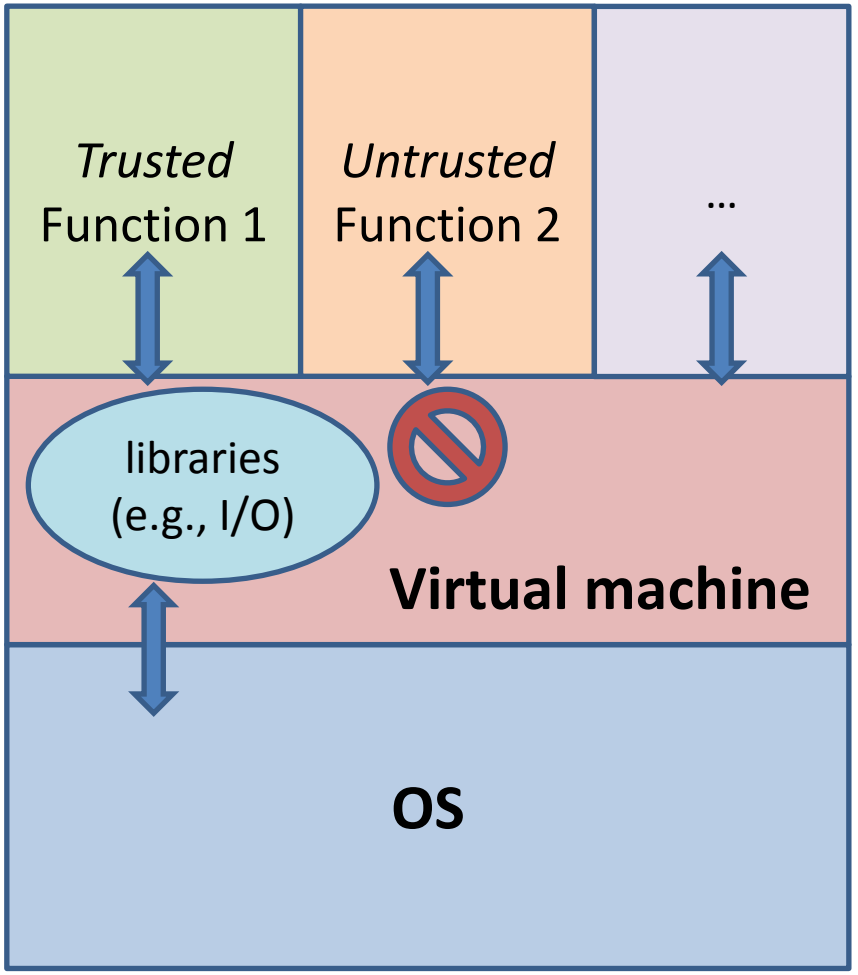
The basic sandbox policy

- *Trusted code* (e.g., local code) has the full power of the user that runs it.
- *Untrusted code* (e.g., foreign code) has very limited rights, e.g.:
  - no direct use of files,
  - network connections only to the code's origin.
- The sandbox is enforced at run-time:
  - A reference monitor ("security manager") is associated with code when the code is loaded.

The basic sandbox policy

Trusted code can access libraries and thereby the underlying OS services.

Untrusted code mostly cannot.

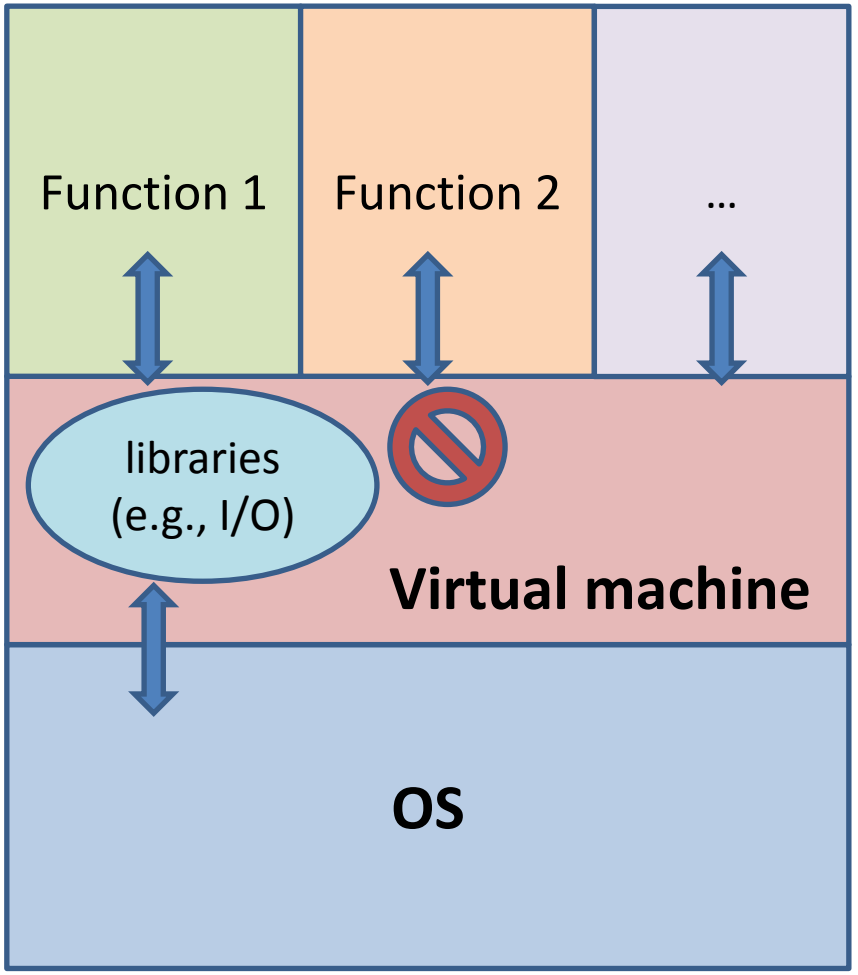


Permissions (as in Java)

Access to resources is expressed in terms of ***permissions***, such as “may perform screen I/O”.

Before execution, an annotation on each piece of code (e.g., function) indicates its permissions.

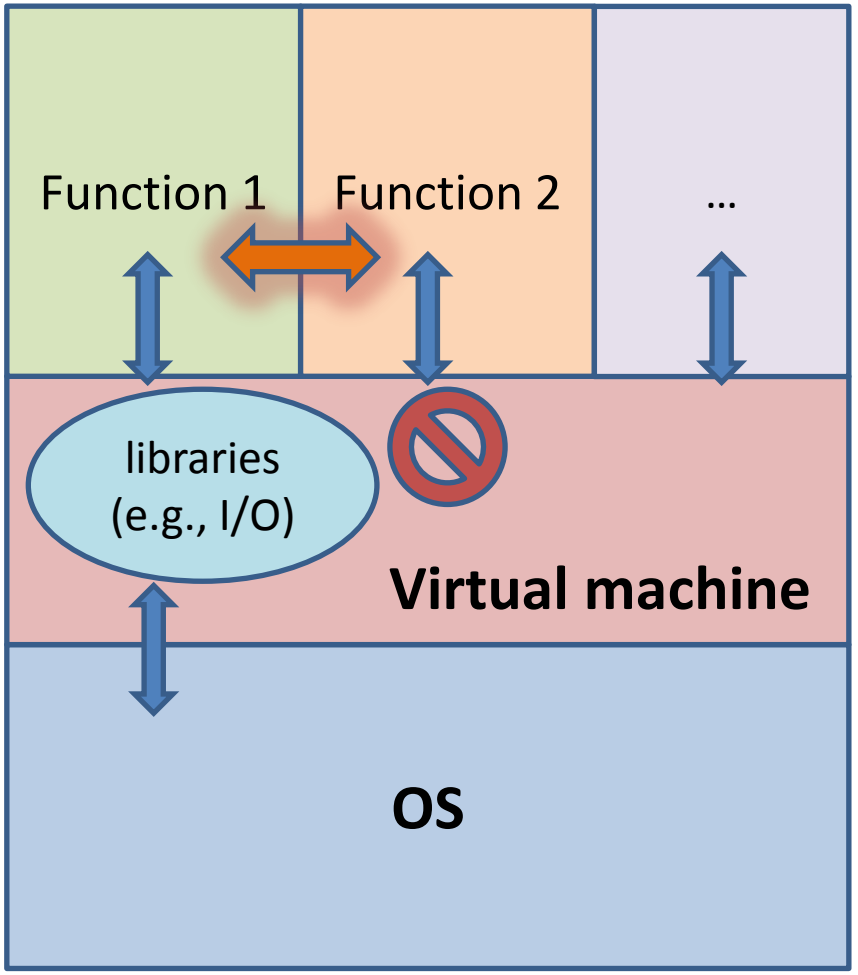
A ***configurable policy*** determines permissions depending on ***code origin***.



Permissions (cont.)

Code with a variety of origins, more or less trusted, may call one another or share data.

Should all of their permissions count in access decisions?



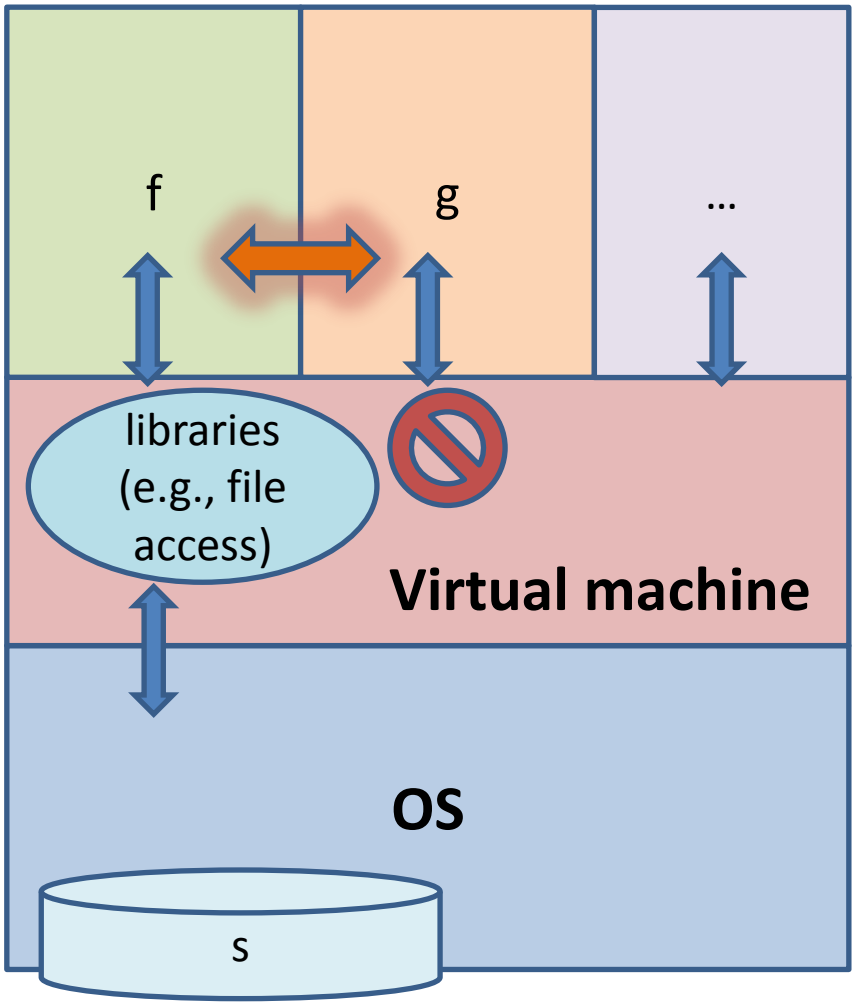
One answer, on a simple example

(also as in Java)

Suppose that f(s) modifies the file named s.

If g calls f(s), ***both*** should have permission to write to s.

(Otherwise, f may be used as a ***confused deputy***.)

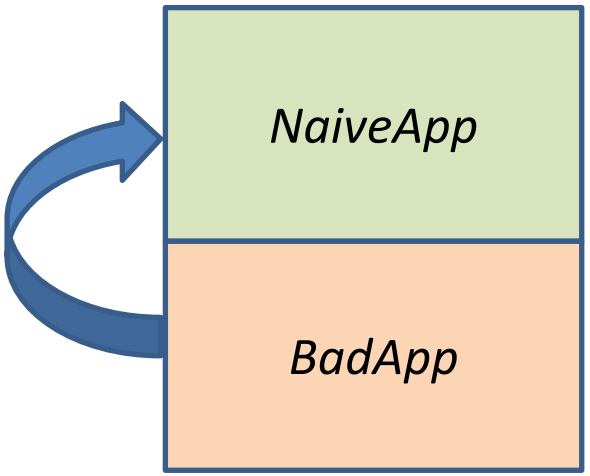


An example where looking at the stack suffices

```
// Fully trusted but naive: has all permissions
public class NaiveApp {
  public static void Write (string s, … ) {
    File.Write (s, … );
  }
}
// Untrusted: no FileIOPermission
class BadApp {
  public static void Main() {
    NaiveApp.Write (“..\\password”, …);
}
```

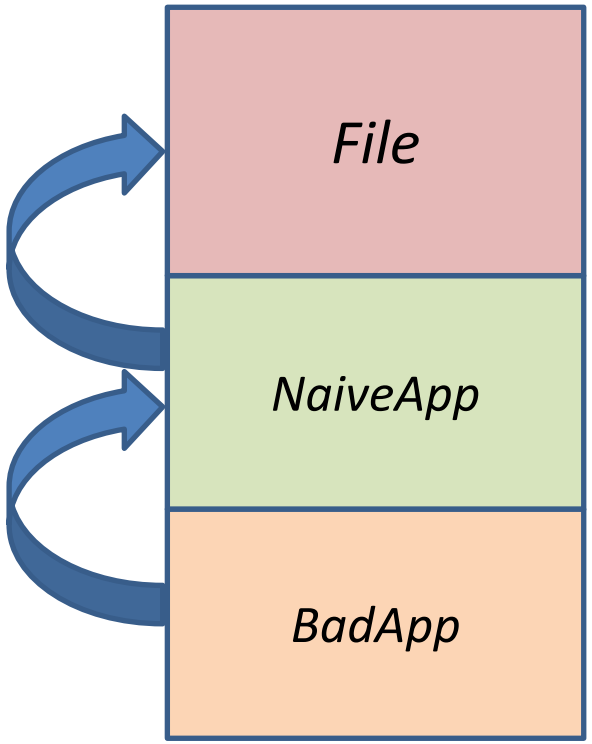
An example where looking at the stack suffices

```
// Fully trusted but naive: has all permissions
public class NaiveApp {
  public static void Write (string s, … ) {
    File.Write (s, … );
  }
}
// Untrusted: no FileIOPermission
class BadApp {
  public static void Main() {
    NaiveApp.Write (“..\\password”, …);
}
```



An example where looking at the stack suffices

```
// Fully trusted but naive: has all permissions
public class NaiveApp {
  public static void Write (string s, … ) {
    File.Write (s, … );
  }
}
// Untrusted: no FileIOPermission
class BadApp {
  public static void Main() {
    NaiveApp.Write (“..\\password”, …);
}
```

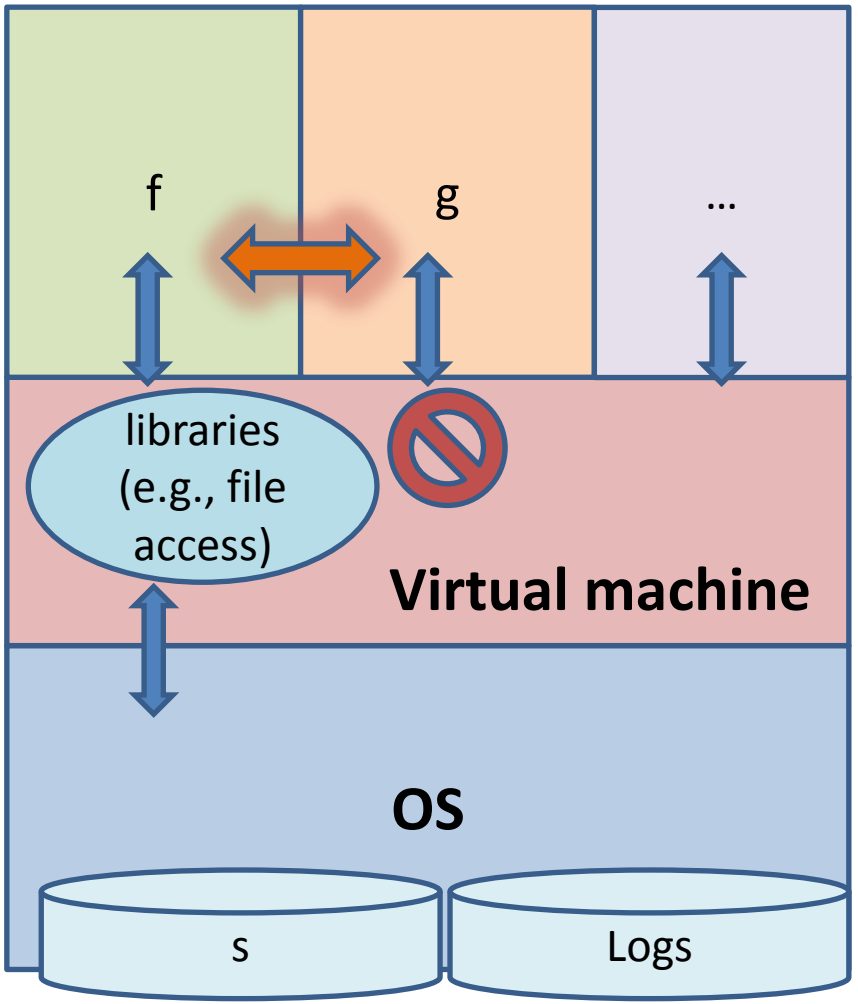


A twist

Suppose that f(s) wants to write to a log that g should not access.

If f is a trusted function, it can check that g's call is ok, assert it, and then use its own authority for writing to the log.

Afterwards, g's permissions do not matter, only f's.

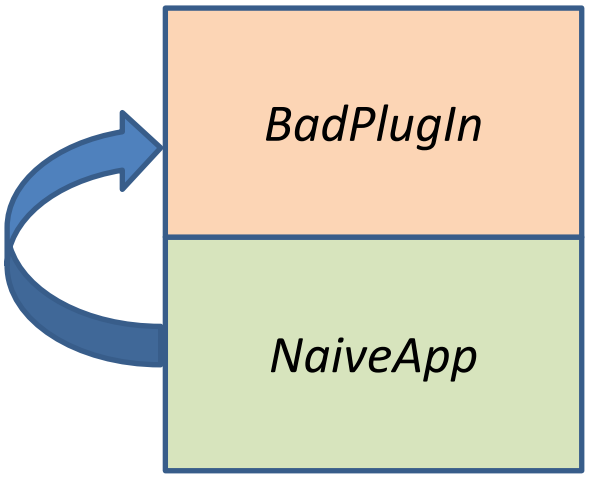


An example where looking at the stack does not suffice

```
// Fully trusted but naive: has all permissions
class NaiveApp {
  public static void Main() {
    string s = BadPlugIn.TempFile ();
    File.Write(s, … );
  }
}
// Untrusted: no FileIOPermission
public class BadPlugIn {
    public static string TempFile () {
      return "..\\ password";
  }
}
```

An example where looking at the stack does not suffice

```
// Fully trusted but naive: has all permissions
class NaiveApp {
  public static void Main() {
    string s = BadPlugIn.TempFile ();
    File.Write(s, … );
  }
}
// Untrusted: no FileIOPermission
public class BadPlugIn {
    public static string TempFile () {
      return "..\\ password";
  }
}
```

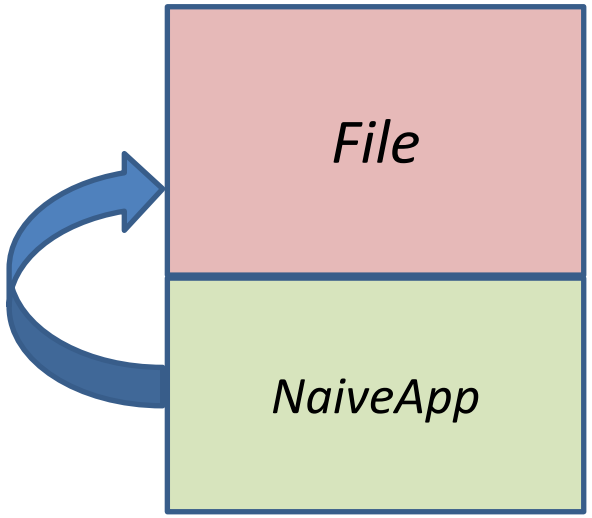


An example where looking at the stack does not suffice

```
// Fully trusted but naive: has all permissions
class NaiveApp {
  public static void Main() {
    string s = BadPlugIn.TempFile ();
    File.Write(s, … );
  }
}
// Untrusted: no FileIOPermission
public class BadPlugIn {
    public static string TempFile () {
      return "..\\ password";
  }
}
```

An example where looking at the stack does not suffice

```
// Fully trusted but naive: has all permissions
class NaiveApp {
  public static void Main() {
    string s = BadPlugIn.TempFile ();
    File.Write(s, … );
  }
}
// Untrusted: no FileIOPermission
public class BadPlugIn {
    public static string TempFile () {
      return "..\\ password";
  }
}
```



Criticisms

- Does this technique achieve real security? *for what policy?*
- Looking at chains of calls is not satisfactory.
  - Some other constructs require careful treatment.
  - A standard formulation ("stack inspection") is tied to a particular stack implementation.
    $\Rightarrow$ It rules out or complicates optimizations.
- It can get hard to understand security.

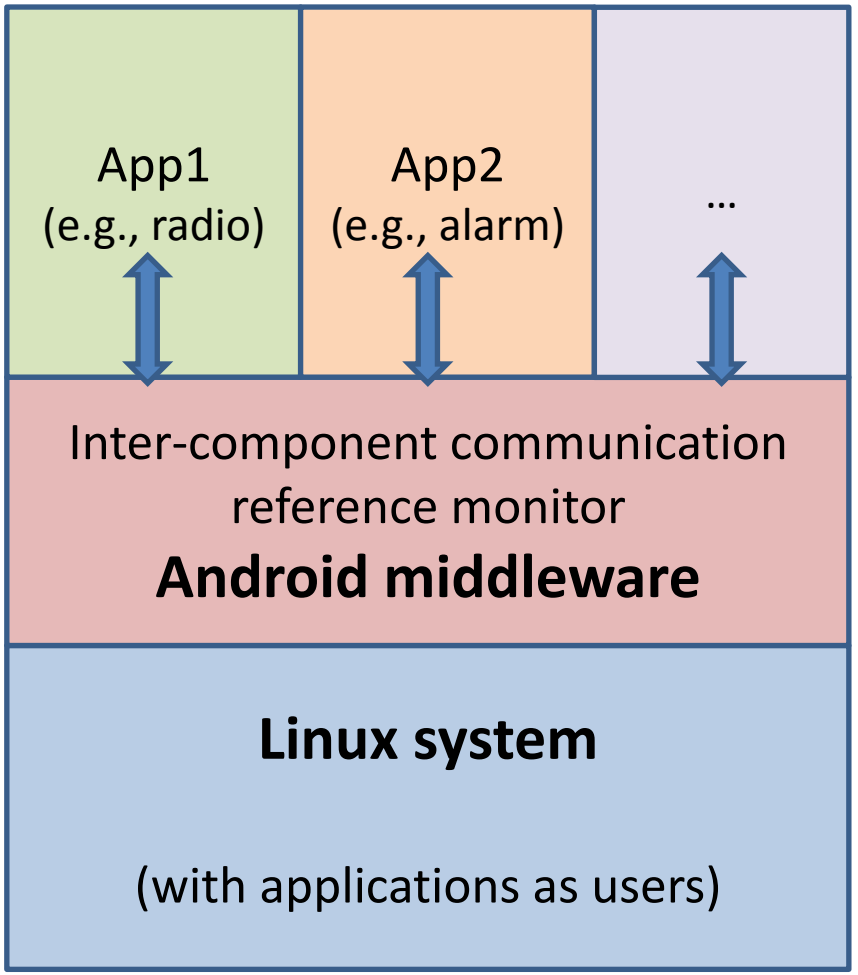
See "Stack Inspection: Theory and Variants", by Fournet and Gordon.

Access control in Android

Applications are principals.

Each application comes with fixed permissions

- declared by developer;
- accepted by user at installation time;
- checked at run-time;
- some standard, e.g., access network;
- others defined by developers;
- over 100.



(For many other aspects to Android security, see "Understanding Android Security", by Enck et al..)

Mandatory access controls and security levels

DAC vs. MAC

Discretionary access control

- This is the familiar case.
  - E.g., the owner of a file can make it accessible to anyone.
- This access control is intrinsically limited in saving principals from themselves.
- It is hard to enforce system-wide security.

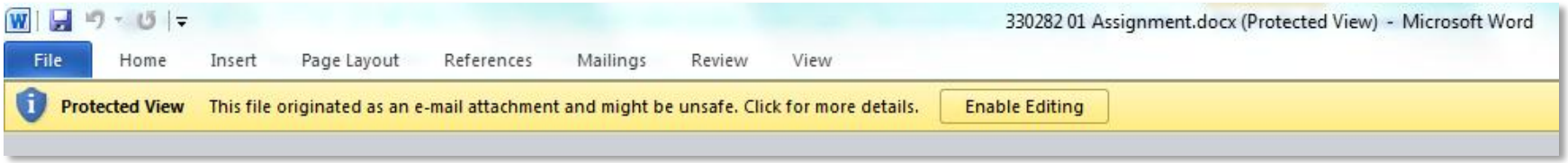
Mandatory access control

- The system assigns security attributes (labels) to both principals and objects.
  - E.g., objects may be “work” or “fun”, and principals may be “trusted” or “guest”.
- These attributes constrain accesses. (Discretionary controls apply in addition.)
  - E.g., “guest” principals cannot modify “work” objects.

MAC (cont.)

- MAC appeared in systems since the 1960s.
- Despite difficulties and disappointments, it also appears more recently, e.g., in
  - *Windows Mandatory Integrity Controls*, where there are four levels for principals and objects:
    - System integrity (e.g., system services)
    - High integrity (e.g., administrative processes)
    - Medium integrity (the default)
    - Low integrity (e.g., for documents from the Internet)
  - *SELinux* "Security-Enhanced Linux" (richer)

A manifestation: protected view of files that arrive by e-mail



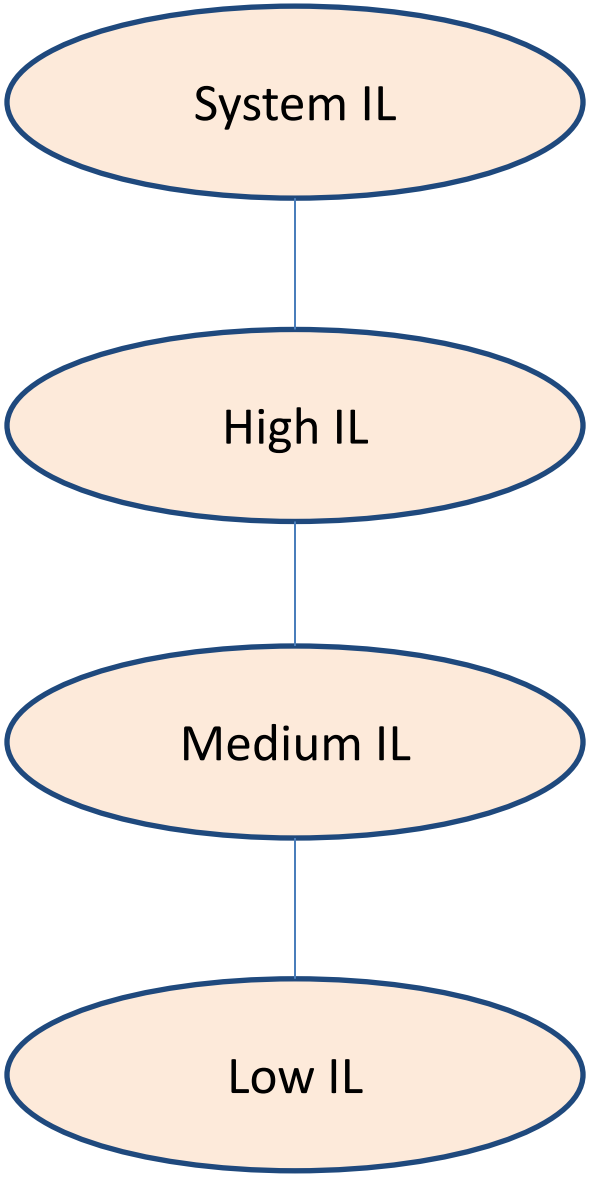
Applications in ***protected mode*** are subject to various restrictions.

Some of these restrictions are achieved by running the applications at Low IL.

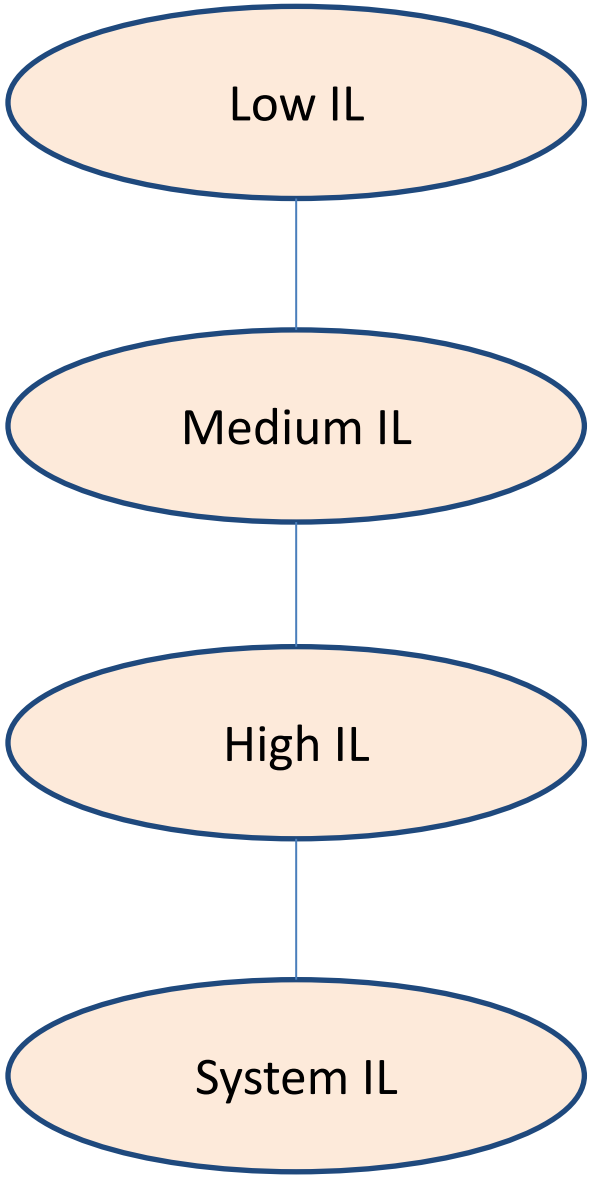
Multilevel security

- As in the examples, MAC is often associated with ***security levels***.
- The security levels can pertain to secrecy and integrity properties in a variety of contexts.
- The levels need not be linearly ordered.
  - Often, they form a partial order or a lattice.
  - They may in part reflect a compartment structure.

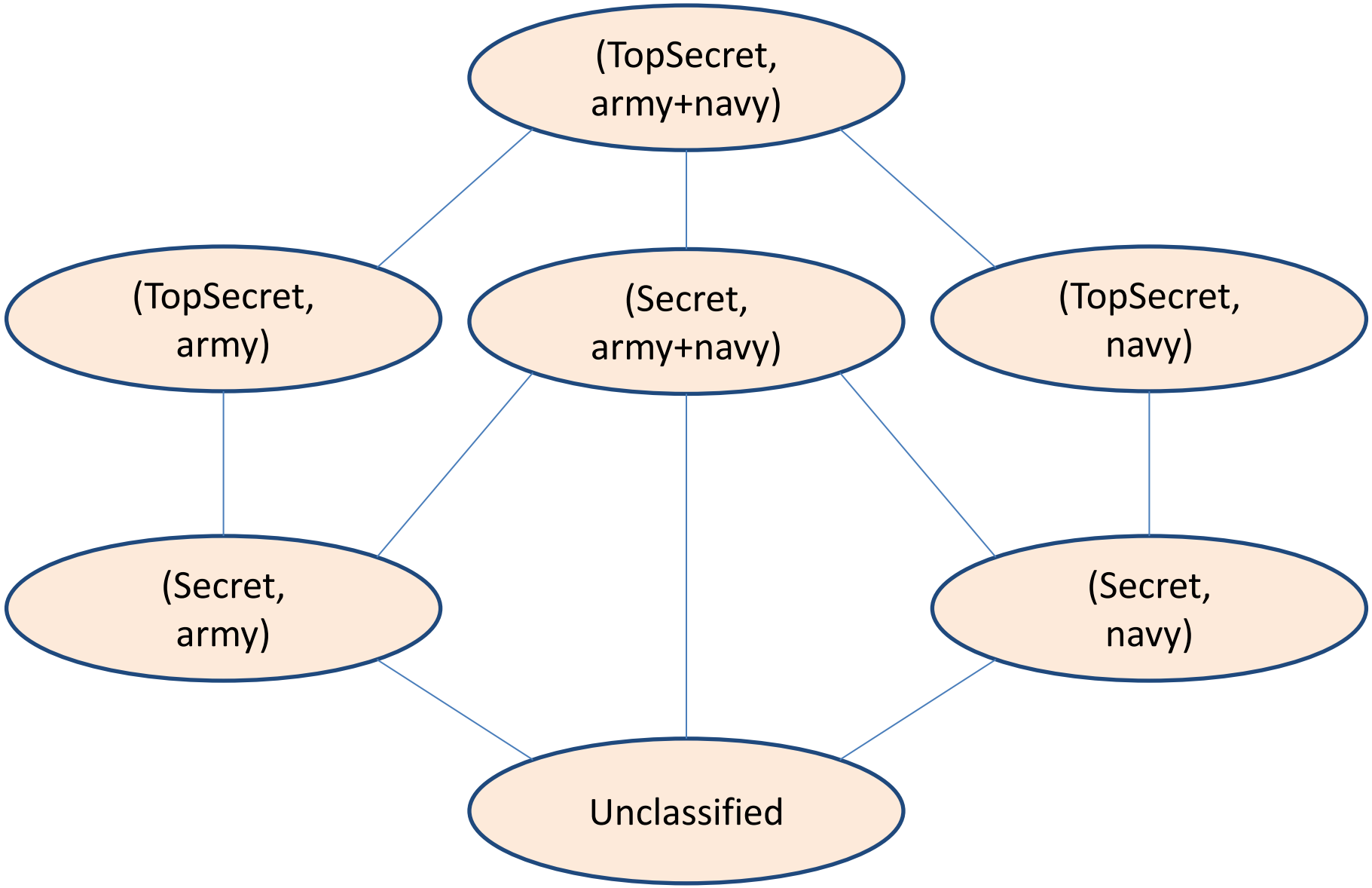
A partial order



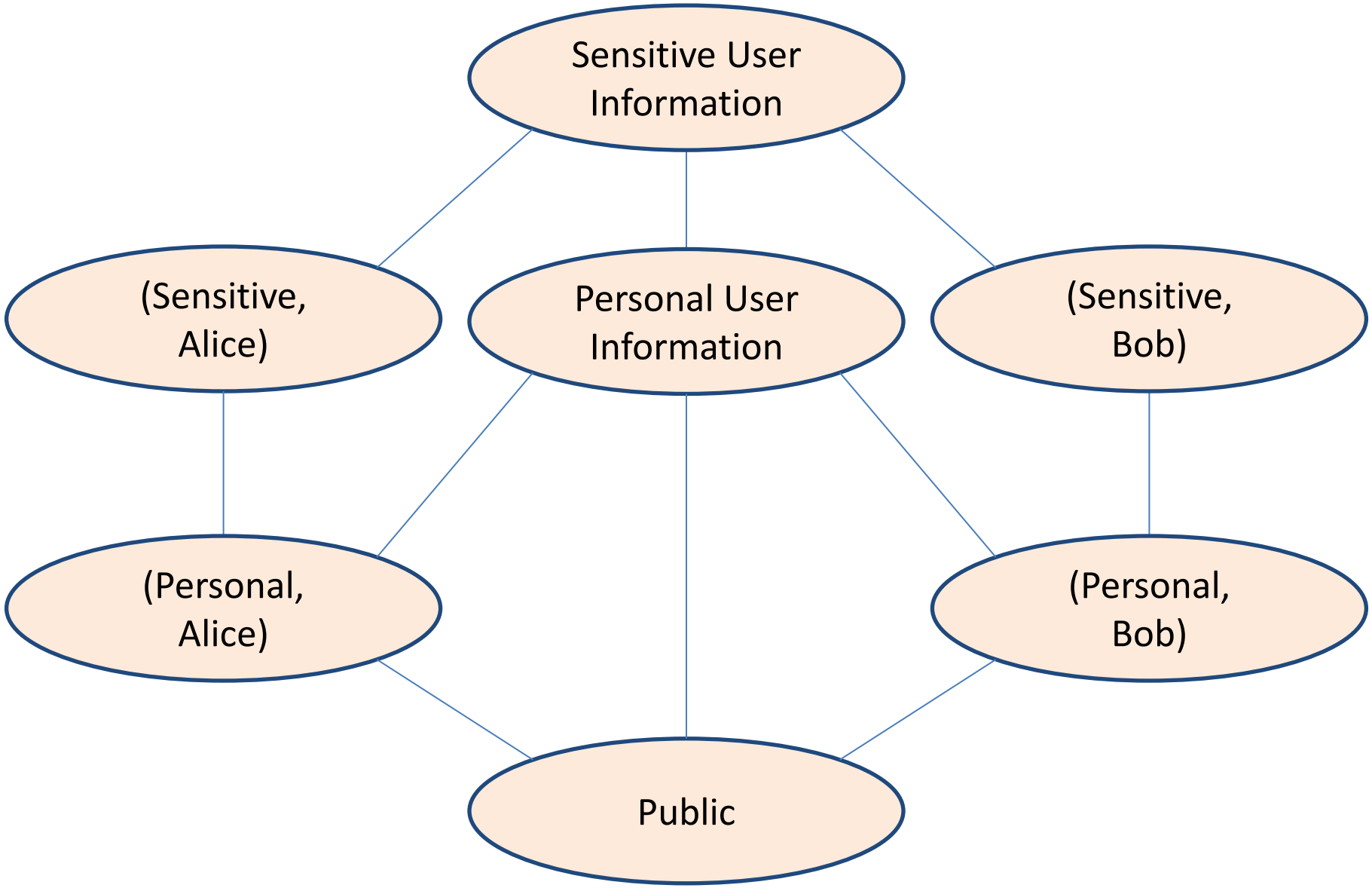
Another partial order



Another partial order



Another partial order



Bell-LaPadula requirements

- ***No read-up:***
  a principal at a given security level may not read an object at a higher security level.
- ***No write-down:***
  a principal at a given security level may not write to an object at a lower security level.

$\Rightarrow$ protects against **Trojan horses**
(bad programs or other principals that work at high security levels)

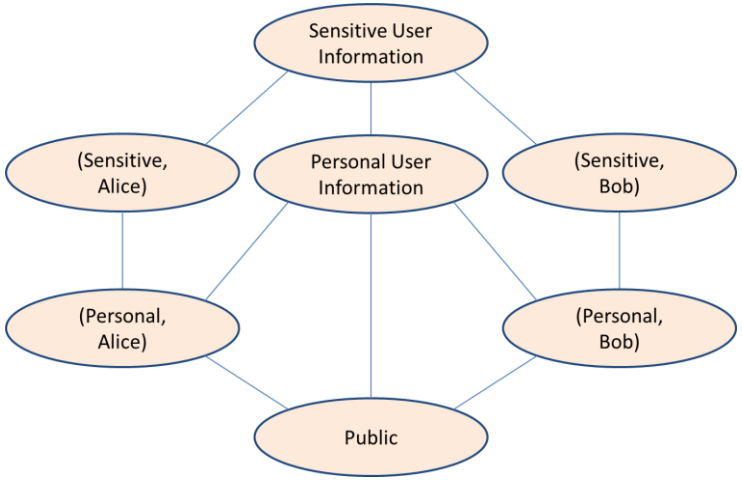
Some difficulties: level creep, declassification, covert channels

Level creep

- Security levels may change over time.
- Security levels tend to creep up.

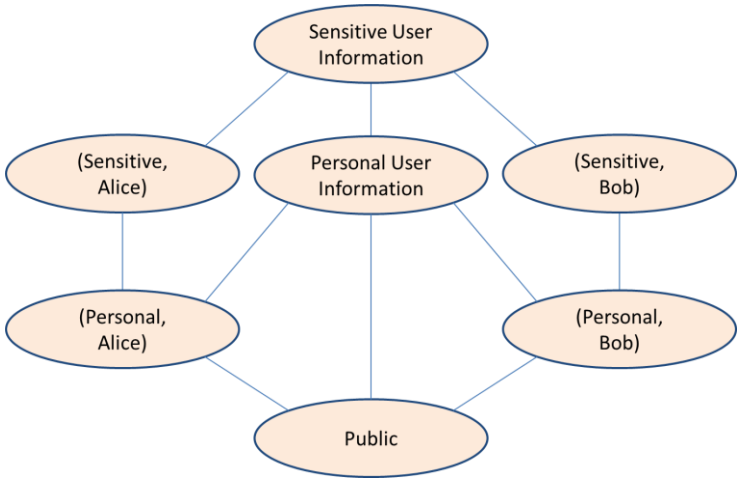
Level creep

- Security levels may change over time.
- Security levels tend to creep up. E.g.:
  - P is a program that may run at any level.
  - blog.html is a file of initial level Public,
  - AliceDiary.txt is a file of initial level (Sensitive, Alice).
  - P may start at Public and write to blog.html, then go to (Sensitive, Alice) and read AliceDiary.txt.
  - Afterwards, P can no longer write to blog.html unless blog.html's level is raised to (Sensitive, Alice).



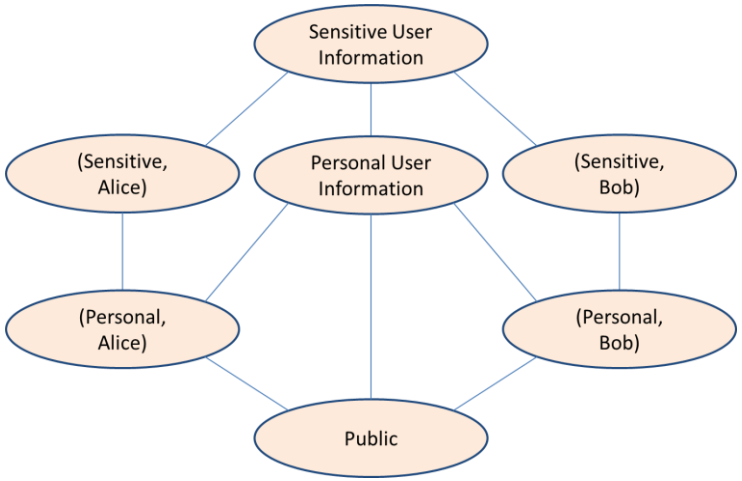
Level creep

- Security levels may change over time.
- Security levels tend to creep up. E.g.:
  - P is a program that may run at any level.
  - blog.html is a file of initial level Public,
  - AliceDiary.txt is a file of initial level (Sensitive, Alice).
  - P may start at Public and write to blog.html, then go to (Sensitive, Alice) and read AliceDiary.txt.
  - Afterwards, P can no longer write to blog.html unless blog.html's level is raised to (Sensitive, Alice).



Level creep

- Security levels may change over time.
- Security levels tend to creep up. E.g.:
  - P is a program that may run at any level.
  - blog.html is a file of initial level Public,
  - AliceDiary.txt is a file of initial level (Sensitive, Alice).
  - P may start at Public and write to blog.html, then go to (Sensitive, Alice) and read AliceDiary.txt.
  - Afterwards, P can no longer write to blog.html unless blog.html's level is raised to (Sensitive, Alice).

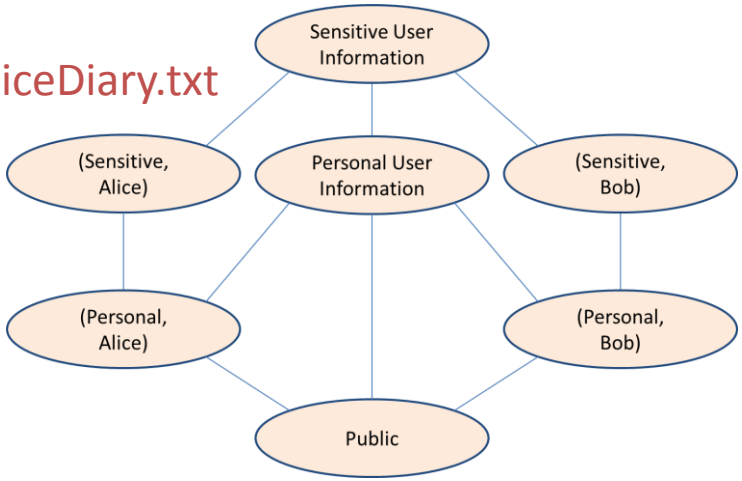


blog.html

Level creep

- Security levels may change over time.
- Security levels tend to creep up. E.g.:
  - P is a program that may run at any level.
  - blog.html is a file of initial level Public,
  - AliceDiary.txt is a file of initial level (Sensitive, Alice).
  - P may start at Public and write to blog.html, then go to (Sensitive, Alice) and read AliceDiary.txt.
  - Afterwards, P can no longer write to blog.html unless blog.html's level is raised to (Sensitive, Alice).

AliceDiary.txt



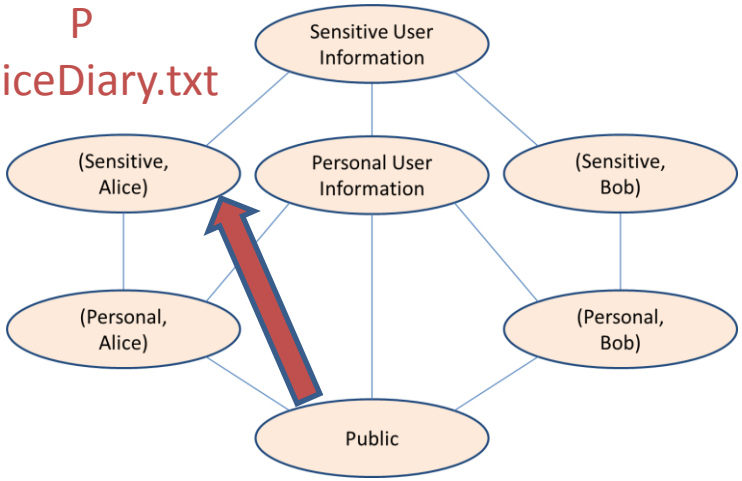
blog.html

Level creep

- Security levels may change over time.
- Security levels tend to creep up. E.g.:
  - P is a program that may run at any level.
  - blog.html is a file of initial level Public,
  - AliceDiary.txt is a file of initial level (Sensitive, Alice).
  - P may start at Public and write to blog.html, then go to (Sensitive, Alice) and read AliceDiary.txt.
  - Afterwards, P can no longer write to blog.html unless blog.html's level is raised to (Sensitive, Alice).

P

AliceDiary.txt



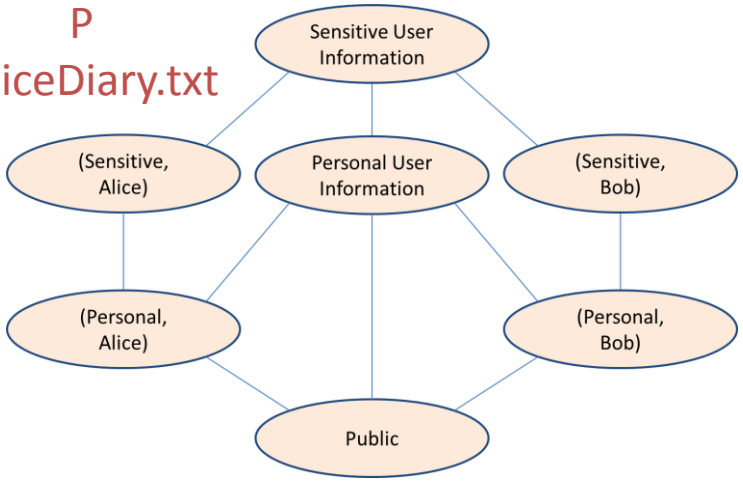
blog.html

Level creep

- Security levels may change over time.
- Security levels tend to creep up. E.g.:
  - P is a program that may run at any level.
  - blog.html is a file of initial level Public,
  - AliceDiary.txt is a file of initial level (Sensitive, Alice).
  - P may start at Public and write to blog.html, then go to (Sensitive, Alice) and read AliceDiary.txt.
  - Afterwards, P can no longer write to blog.html unless blog.html's level is raised to (Sensitive, Alice).

P

AliceDiary.txt



blog.html

Declassification

- Reclassification consists in changing the security attributes. Declassification is the case in which this is not automatically ok.
- Declassification is needed sometimes.
  E.g., the password-checking program reads a secret database and says yes/no to a user.

Declassification

- Reclassification consists in changing the security attributes. Declassification is the case in which this is not automatically ok.
- Declassification is needed sometimes.
  E.g., the password-checking program reads a secret database and says yes/no to a user.
- It is difficult
  - because of hidden messages
    (in text, in pictures, …)

Declassification

- Reclassification consists in changing the security attributes. Declassification is the case in which this is not automatically ok.
- Declassification is needed sometimes.
  E.g., the password-checking program reads a secret database and says yes/no to a user.
- It is difficult
  - because of hidden messages (in text, in pictures, …)

A boat, beneath a sunny sky
Lingering onward dreamily
In an evening of July -
Children three that nestle near,
Eager eye and willing ear,

Declassification

- Reclassification consists in changing the security attributes. Declassification is the case in which this is not automatically ok.
- Declassification is needed sometimes.
  E.g., the password-checking program reads a secret database and says yes/no to a user.
- It is difficult
  - because of hidden messages (in text, in pictures, …)

A
L
I
C
E

Declassification

- Reclassification consists in changing the security attributes. Declassification is the case in which this is not automatically ok.
- Declassification is needed sometimes.
  E.g., the password-checking program reads a secret database and says yes/no to a user.
- It is difficult
  - because of hidden messages
    (in text, in pictures, …)

Declassification

- Reclassification consists in changing the security attributes. Declassification is the case in which this is not automatically ok.
- Declassification is needed sometimes.
  E.g., the password-checking program reads a secret database and says yes/no to a user.
- It is difficult
  - because of hidden messages
    (in text, in pictures, …)

Declassification

- Reclassification consists in changing the security attributes. Declassification is the case in which this is not automatically ok.
- Declassification is needed sometimes.
  E.g., the password-checking program reads a secret database and says yes/no to a user.
- It is difficult
  - because even the act of declassification may reveal some information

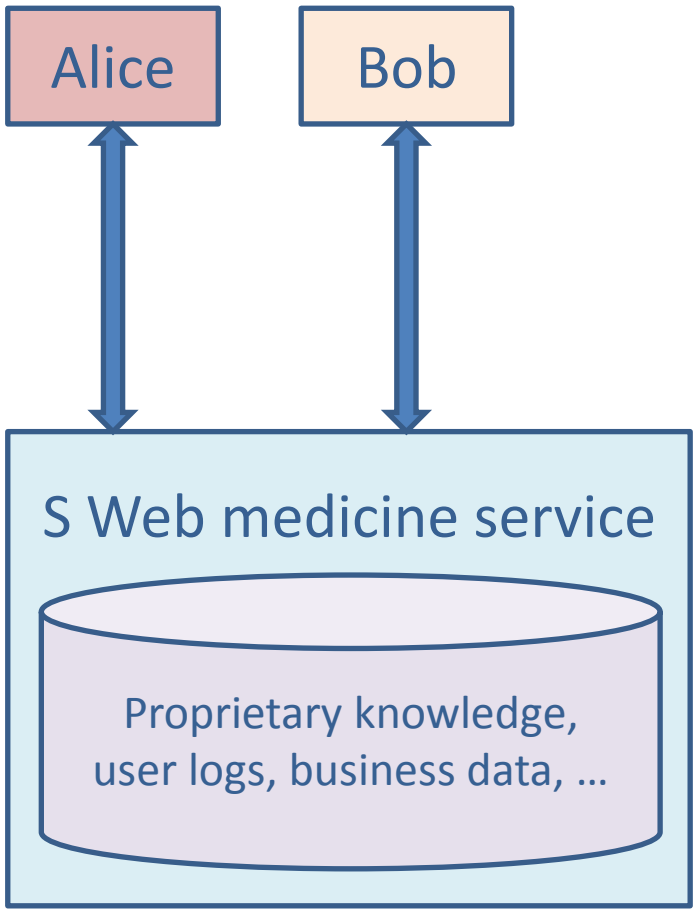
Declassification

- Reclassification consists in changing the security attributes. Declassification is the case in which this is not automatically ok.
- Declassification is needed sometimes.
  E.g., the password-checking program reads a secret database and says yes/no to a user.
- It is difficult.

$\Rightarrow$ *It is a special process, often manual.*

Mutual distrust

Consider a Web service S that offers information to users (e.g., advice or ads).
S relies on proprietary information and user data (e.g., financial data, preferences, email, clicks).
What is a reasonable policy?
Who can declassify what?



Covert channels

Covert channels are communication channels for which the model does not account and which were not intended for communication.

E.g., programs may communicate by the use of shared resources:

By varying its ratio of computing to input/output or its paging rate, the service can transmit information which a concurrently running process can receive by observing the performance of the system.

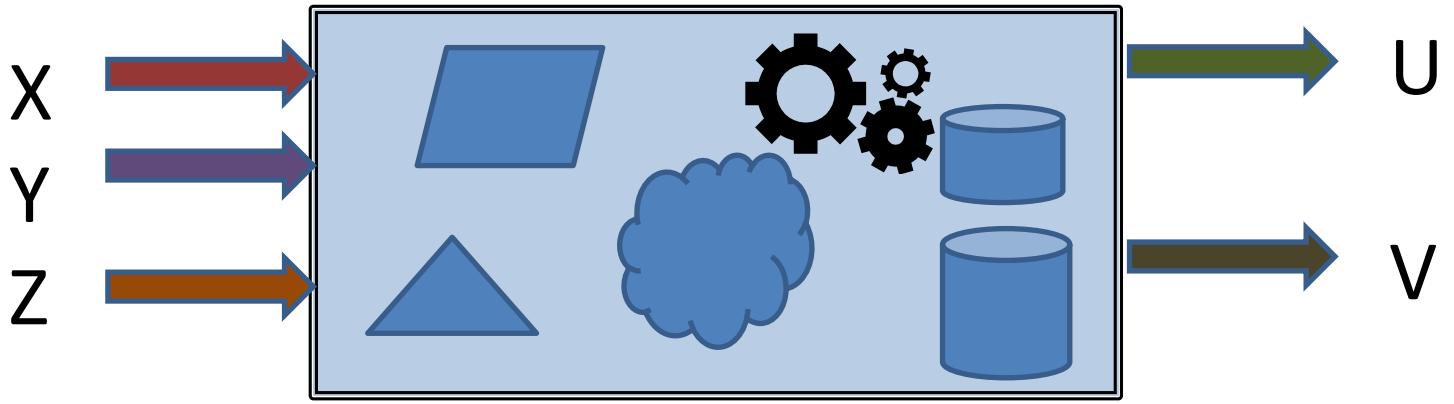
Lampson, 1973

- The “service” may be a Trojan horse, without network access.
- The “concurrently running process” may have a lower level and send any information that it receives on the network.

Information flow

Information flow security

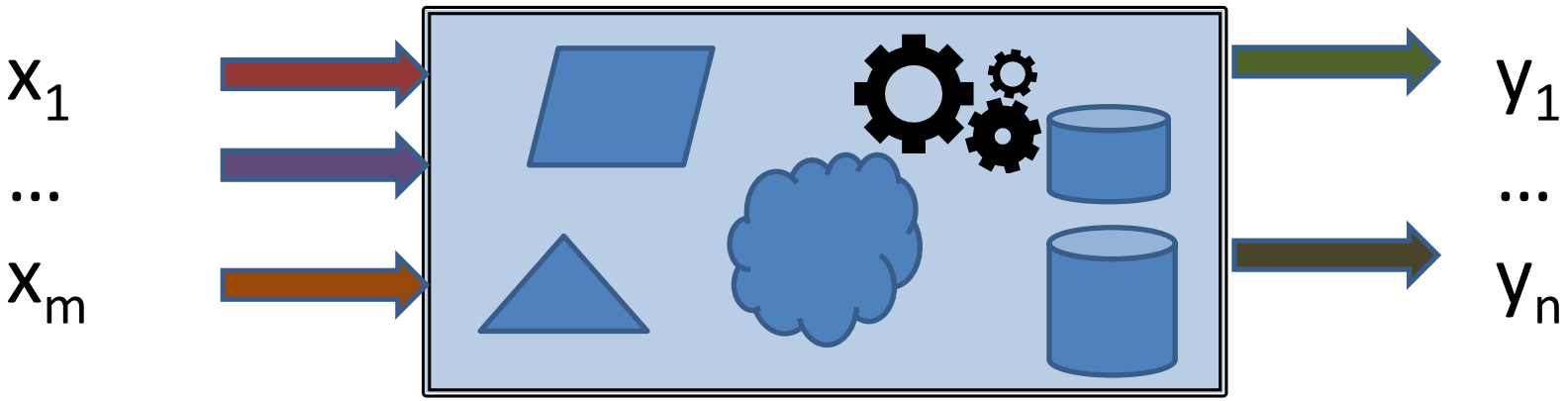
- Access control, of any kind, is limited to the defined principals, objects, and operations.
- Instead, information flow control focuses on the information being protected, *end-to-end*.



E.g., we may want that U do not depend on Z, that is, that Z does not *interfere* with U.

Noninterference: preliminaries

- Consider a system with inputs $x_1,\ldots,x_m$ and outputs $y_1,\ldots,y_n$.
- Suppose $y_j = f_j(x_1,\ldots,x_m)$.
  - Extensions deal with infinite computations, probabilities, nondeterminism, and more.



Noninterference: independence

- So, suppose $y_j = f_j(x_1,\ldots,x_m)$.
- Then y_j *does not depend* on x_i if, always (for all actual values for the inputs),
  $f_j(v_1, \ldots, v_i, \ldots, v_m) = f_j(v_1, \ldots, v_i', \ldots, v_m)$.

$\Rightarrow$ Secrecy: the value of y_j reveals nothing about the value of x_i.

$\Rightarrow$ Integrity: the value of y_j is not affected by corruptions in the value of x_i.

Noninterference

- Pick some levels (e.g., Public, TopSecret, etc.) with an order on the levels.
- Assign a level to each input $x_1,\ldots,x_m$ and to each output $y_1,\ldots,y_n$.
- ***Noninterference***:
  An output may depend on inputs of the same level, or lower levels, but not on other inputs.
  - So, e.g., outputs of level Public must not depend on inputs of level Sensitive User Information.

Simple examples

- Suppose that Public $<$ Secret.
- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.

Simple examples

- Suppose that Public $<$ Secret.
- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$

Simple examples

- Suppose that Public $<$ Secret.
- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$ *ok* (y_2 does not matter)

Simple examples

- Suppose that Public $<$ Secret.
- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$ *ok* (y_2 does not matter)
- $y_1 = x_2$

Simple examples

- Suppose that Public $<$ Secret.
- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$ *ok* (y_2 does not matter)
- $y_1 = x_2$ *not ok*

  There is an ***explicit flow*** of information.

Simple examples

- Suppose that Public $<$ Secret.
- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$ *ok* (y_2 does not matter)
- $y_1 = x_2$ *not ok*
  There is an ***explicit flow*** of information.
- if x_2 is odd then $y_1 = 1$ else $y_1 = 0$

Simple examples

- Suppose that Public $<$ Secret.
- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$ *ok* (y_2 does not matter)
- $y_1 = x_2$ *not ok*
  There is an ***explicit flow*** of information.
- if x_2 is odd then $y_1 = 1$ else $y_1 = 0$ *not ok*
  There is an ***implicit flow*** of information.

Simple examples

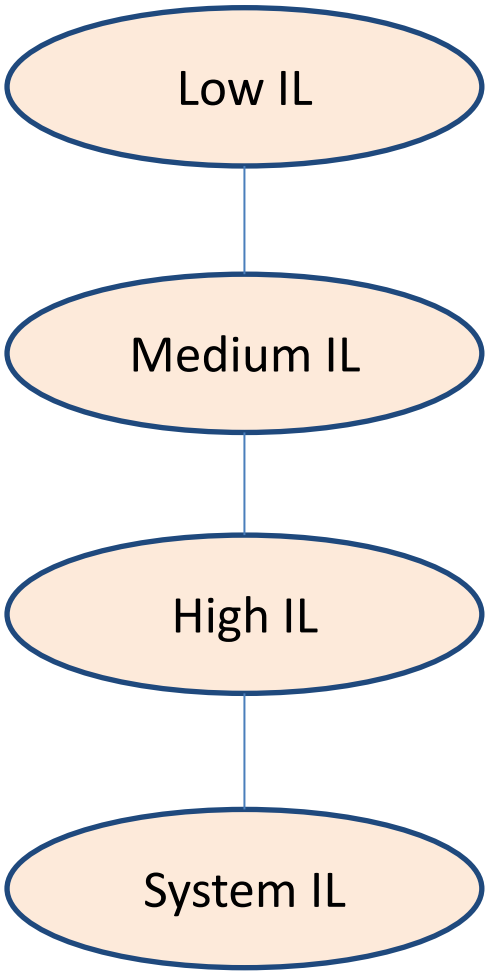
- Suppose that Public $<$ Secret.
- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$ *ok* (y_2 does not matter)
- $y_1 = x_2$ *not ok*
  There is an ***explicit flow*** of information.
- if x_2 is odd then $y_1 = 1$ else $y_1 = 0$ *not ok*
  There is an ***implicit flow*** of information.
- if x_2 is odd then $y_1 = 1$ else $y_1 = 1$

Simple examples

- Suppose that Public $<$ Secret.
- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$ *ok* (y_2 does not matter)
- $y_1 = x_2$ *not ok*
  There is an ***explicit flow*** of information.
- if x_2 is odd then $y_1 = 1$ else $y_1 = 0$ *not ok*
  There is an ***implicit flow*** of information.
- if x_2 is odd then $y_1 = 1$ else $y_1 = 1$ *ok*

Noninterference for integrity

- The definition of noninterference applies to integrity.
- Intuitively the levels need to be ordered “upside-down”.
  - E.g., so that System IL outputs cannot depend on Low IL inputs.



Dynamic information flow control

Dynamic information flow control

- Detect information flows dynamically.
  - A very old idea.
  - In use, e.g., in Perl taint propagation.
  - Often expensive, imprecise, but useful.
  - Still an active research subject, e.g.,
    - for JavaScript in browsers,
    - in operating systems such Asbestos or HiStar,
    - for selective re-execution in “undo computing” (loosely) [Kim et al.].

Tracking levels: simple examples

Propagate security levels at run-time:

- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$ *ok, allowed*
- $y_1 = x_2$ *not ok, easily blocked*
- temp = x_1; y_1 = temp *ok, allowed*
- temp = x_2; y_1 = temp *not ok, easily blocked*

Tracking levels: simple examples

Propagate security levels at run-time:

- Input x_1 and output y_1 have level Public.
- Input x_2 and output y_2 have level Secret.
- $y_1 = x_1$ *ok, allowed*
- $y_1 = x_2$ *not ok, easily blocked*
- temp = x_1; y_1 = temp *ok, allowed*
- temp = x_2; y_1 = temp *not ok, easily blocked*
- if x_2 is odd then $y_1 = 1$ else $y_1 = 0$ *blocked?*
- if x_2 is odd then $y_1 = 1$ else $y_1 = 1$ *blocked?*

A more challenging example

$y_2 = x_2$;

$y_1 = 1$;

temp = 1;

if $y_2 = 1$ then temp = 0;

if temp = 1 then $y_1 = 0$;

A more challenging example

$y_2 = x_2$;

$y_1 = 1$;

temp = 1;

if $y_2 = 1$ then temp = 0;

if temp = 1 then $y_1 = 0$;

At the end, if $x_2 = 0$ then $y_1 = 0$, and $y_1 = 1$ otherwise.

A more challenging example

$y_2 = x_2$;

$y_1 = 1$;

temp = 1;

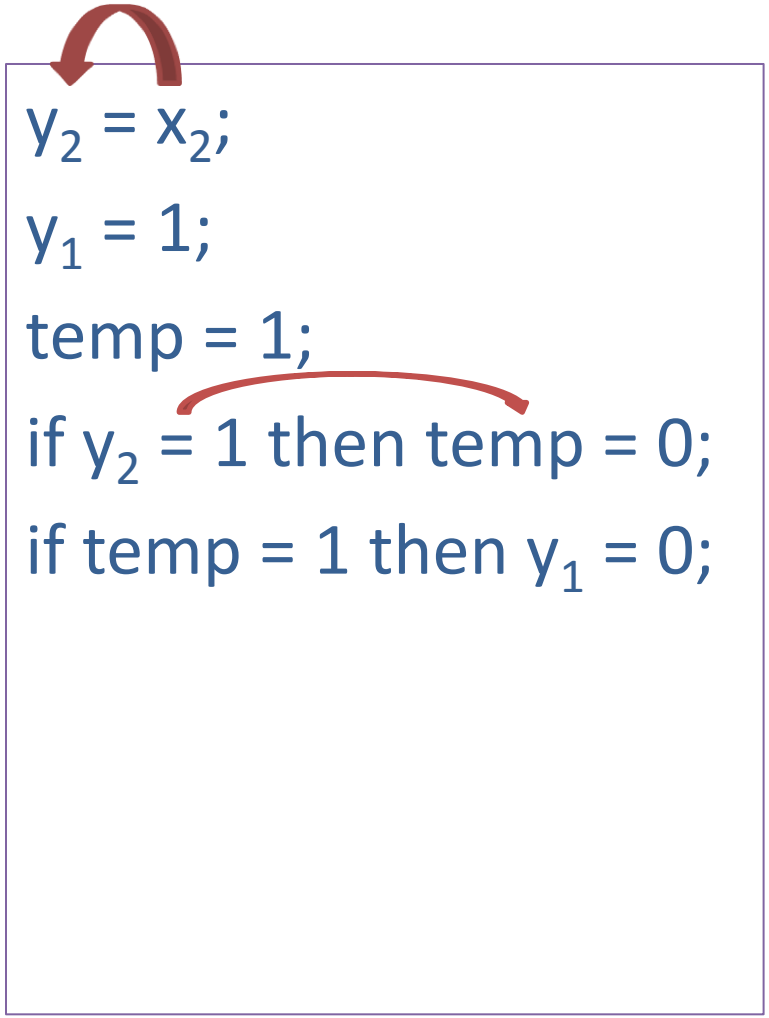
if $y_2 = 1$ then temp = 0;

if temp = 1 then $y_1 = 0$;

At the end, if $x_2 = 0$ then $y_1 = 0$, and $y_1 = 1$ otherwise.

$\Rightarrow$ *So there is a flow.*

A more challenging example



At the end, if $x_2 = 0$ then $y_1 = 0$, and $y_1 = 1$ otherwise.

$\Rightarrow$ *So there is a flow.*

When $x_2 = 1$, dynamic taint propagation may suggest that temp is of level Secret.

A more challenging example

$y_2 = x_2$;

$y_1 = 1$;

temp = 1;

if $y_2 = 1$ then temp = 0;

if temp = 1 then $y_1 = 0$;

At the end, if $x_2 = 0$ then $y_1 = 0$, and $y_1 = 1$ otherwise.

$\Rightarrow$ *So there is a flow.*

When $x_2 = 0$, dynamic taint propagation may suggest that temp is of level Public.

A more challenging example

$y_2 = x_2$;

$y_1 = 1$;

temp = 1;

if $y_2 = 1$ then temp = 0;

if temp = 1 then $y_1 = 0$;

At the end, if $x_2 = 0$ then $y_1 = 0$, and $y_1 = 1$ otherwise.

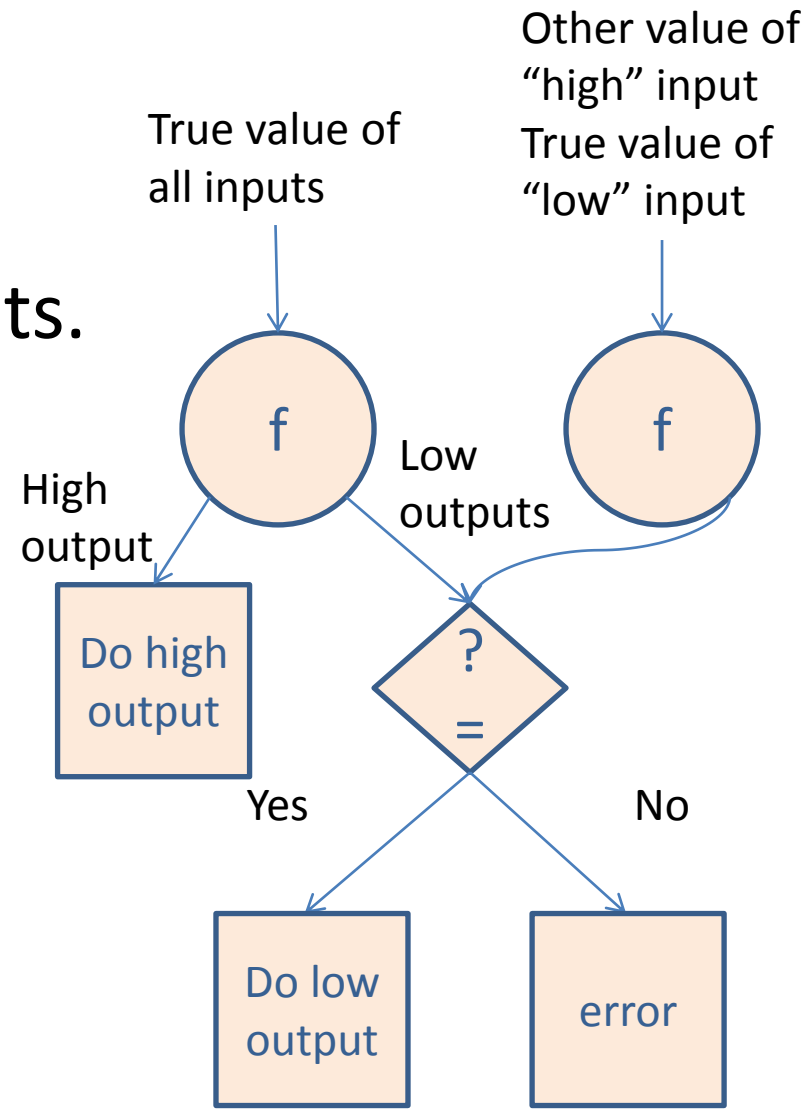
$\Rightarrow$ *So there is a flow.*

In each case, code that is not executed is crucial to the flow!

$\Rightarrow$ *Code analysis is needed.*

Another dynamic technique: multiple executions

- Run multiple copies with different “high” inputs.
- Compare the “low” outputs.
  - If they are equal, then release them.
  - If they are different, then there is information flow, so stop with an error.



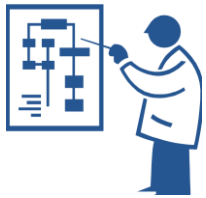
Another dynamic technique: multiple executions (cont.)

- This technique encounters difficulties.
  - Choice of inputs.
  - Efficiency of running multiple copies.
  - Dealing with deliberate nondeterminism.
- But there is research progress.
  - ML^2 [Simonet and Pottier et al.]
  - Self-composition [Barthe et al.]
  - TightLip [Yumerefendi et al.]
  - Secure Multiexecution [Devriese and Piessens]

Static information flow control

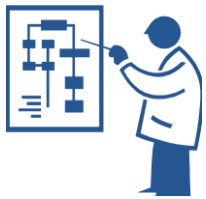
Static information flow control

- Analyze programs before execution.
  - An old idea too.
  - Also with applications to current problems (e.g., finding bugs in Javascript browser extensions with Vex [Bandhakavi et al.]).
  - In recent years, relying on programming-language research (e.g., type systems).



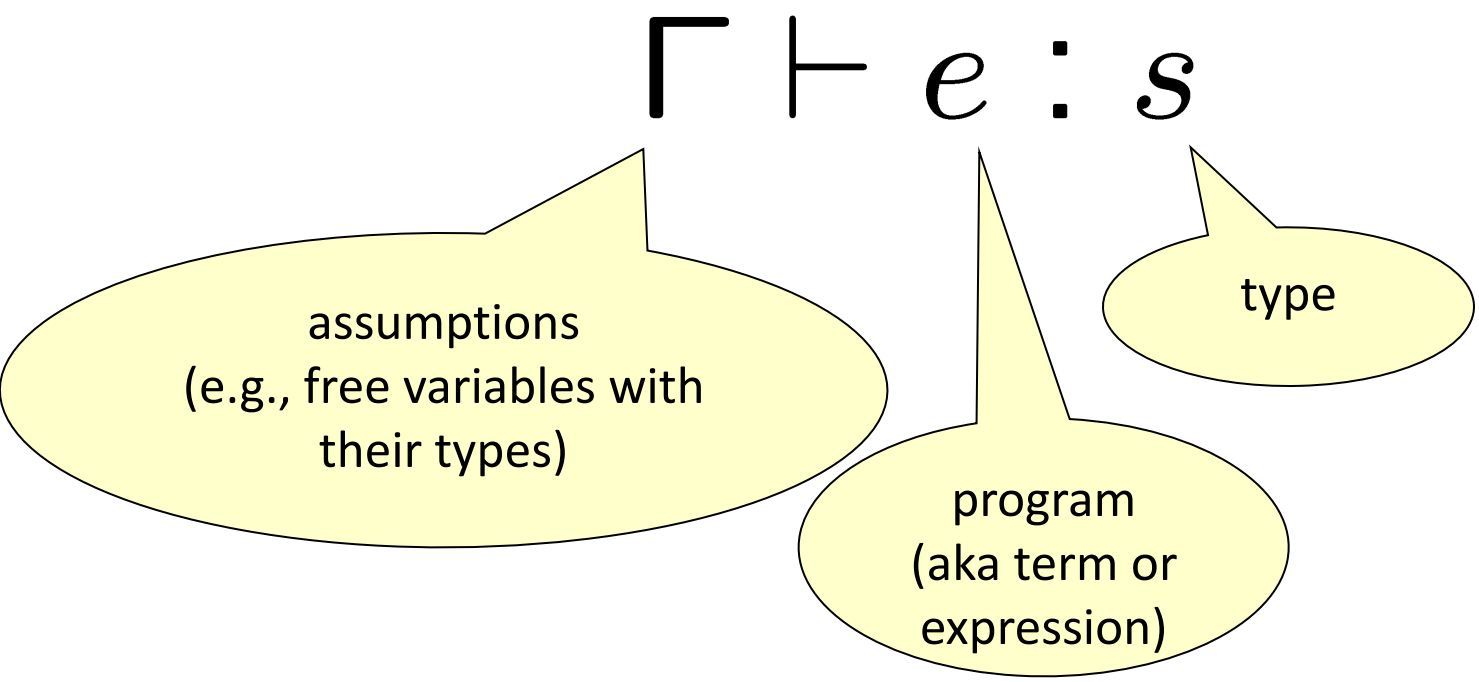
Example of a static approach

- We treat only simple language constructs (following a “monadic” approach).
- One security level (“High”) is explicit.
  All the rest is implicitly of a “Low” level.
  - E.g., int represents the type of (“Low”) integers.
  - High int represents the type of High integers (i.e., the secret integers, in one interpretation).
  - int outputs should not depend on High int inputs.



Typing rules

- As usual, typing rules are rules for deducing judgments (assertions) of the form:



Example judgments and rules

- A judgment: $x : \texttt{int} \vdash x + 0 : \texttt{int}$

- Some rules:

$$\Gamma, x : s, \Gamma' \vdash x : s \qquad\qquad \Gamma \vdash 0 : \texttt{int}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

The Simply Typed λ-calculus: rules

$$\Gamma, x : s, \Gamma' \vdash x : s$$

$$\Gamma \vdash () : \mathtt{true}$$

$$\frac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1.\, e) : (s_1 \to s_2)}$$

$$\frac{\Gamma \vdash e : (s_1 \to s_2) \qquad \Gamma \vdash e' : s_1}{\Gamma \vdash (e\ e') : s_2}$$

$$\frac{\Gamma \vdash e_1 : s_1 \qquad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)}$$

$$\frac{\Gamma \vdash e : (s_1 \times s_2)}{\Gamma \vdash (\mathtt{proj}_1\ e) : s_1}$$

$$\frac{\Gamma \vdash e : (s_1 \times s_2)}{\Gamma \vdash (\mathtt{proj}_2\ e) : s_2}$$

$$\frac{\Gamma \vdash e : s_1}{\Gamma \vdash (\mathtt{inj}_1\ e) : (s_1 + s_2)}$$

$$\frac{\Gamma \vdash e : s_2}{\Gamma \vdash (\mathtt{inj}_2\ e) : (s_1 + s_2)}$$

$$\frac{\Gamma \vdash e : (s_1 + s_2) \qquad \Gamma, x : s_1 \vdash e_1 : s \qquad \Gamma, x : s_2 \vdash e_2 : s}{\Gamma \vdash (\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2) : s}$$

Rules for High

- High can always be added:

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash (\eta\ e) : \texttt{High}\ s}$$

Rules for High

- High can always be added:

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash (\eta\ e) : \texttt{High}\ s}$$

  - So for example $(\eta\ 0) : \texttt{High int}$

Rules for High

- High can always be added:

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash (\eta\ e) : \texttt{High}\ s}$$

  - So for example $(\eta\ 0) : \texttt{High int}$
- High expressions can be used in other High expressions:

$$\frac{\Gamma \vdash e : \texttt{High}\ s \qquad \Gamma, x : s \vdash e' : \texttt{High}\ t}{\Gamma \vdash \texttt{bind}\ x = e\ \texttt{in}\ e' : \texttt{High}\ t}$$

Rules for High

- High can always be added:

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash (\eta\ e) : \texttt{High}\ s}$$

  - So for example $(\eta\ 0) : \texttt{High int}$
- High expressions can be used in other High expressions:

$$\frac{\Gamma \vdash e : \texttt{High}\ s \qquad \Gamma, x : s \vdash e' : \texttt{High}\ t}{\Gamma \vdash \texttt{bind}\ x = e\ \texttt{in}\ e' : \texttt{High}\ t}$$

  - So for example, if $e : \texttt{High int}$ then $\texttt{bind}\ x = e\ \texttt{in}\ (\eta\ (x + 1)) : \texttt{High int}$

Rules for High

- High can always be added:

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash (\eta\ e) : \texttt{High}\ s}$$

  – So for example $(\eta\ 0) : \texttt{High int}$

- High expressions can be used in other High expressions:

$$\frac{\Gamma \vdash e : \texttt{High}\ s \qquad \Gamma, x : s \vdash e' : \texttt{High}\ t}{\Gamma \vdash \texttt{bind}\ x = e\ \texttt{in}\ e' : \texttt{High}\ t}$$

  – So for example, if $e : \texttt{High int}$ then $\texttt{bind}\ x = e\ \texttt{in}\ (\eta\ (x + 1)) : \texttt{High int}$

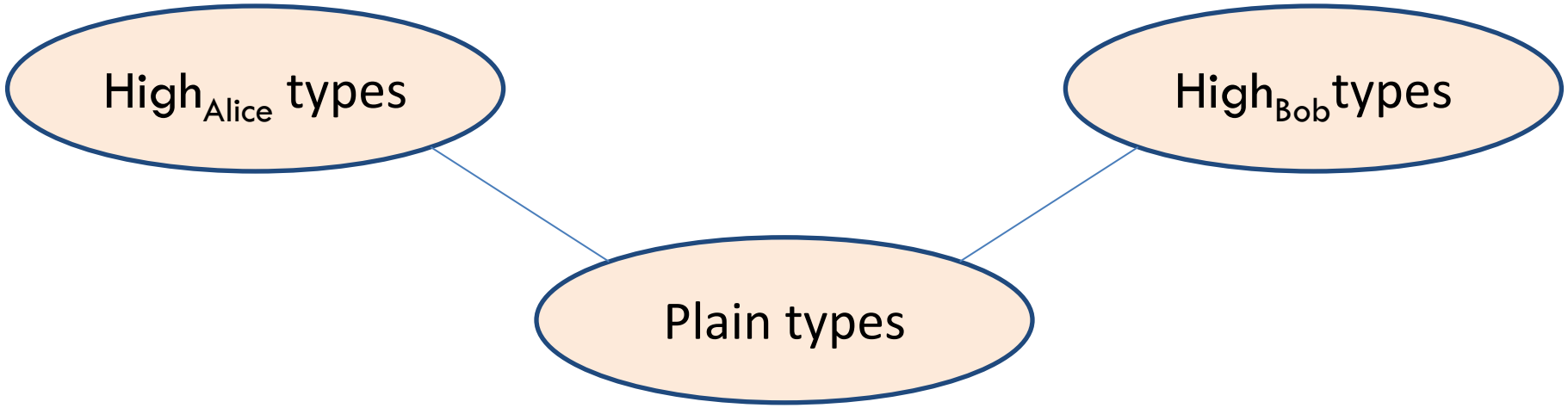
  – But there is no way to go from High to Low.

A simple noninterference property

If $\vdash f : (\texttt{High int}) \to \texttt{int}$,
$\vdash e_1 : \texttt{High int}$, and $\vdash e_2 : \texttt{High int}$,
then $f(e_1)$ and $f(e_2)$ are equal
(that is, evaluate to the same integer).

A first generalization

- Consider multiple principals (Alice, Bob, ...).
- We replace the single level High with a different level High_A for each principal *A*.
  - High_A int may represent the type of *A*'s integer secrets,
  - or the type of integers whose integrity *A* trusts.



Rules for $High_A$

- The rules are basically those for High:
  - $High_A$ can always be added.
  - $High_A$ expressions can be used in computing other $High_A$ expressions (but there is no way to go from $High_A$ to Low or to $High_B$).

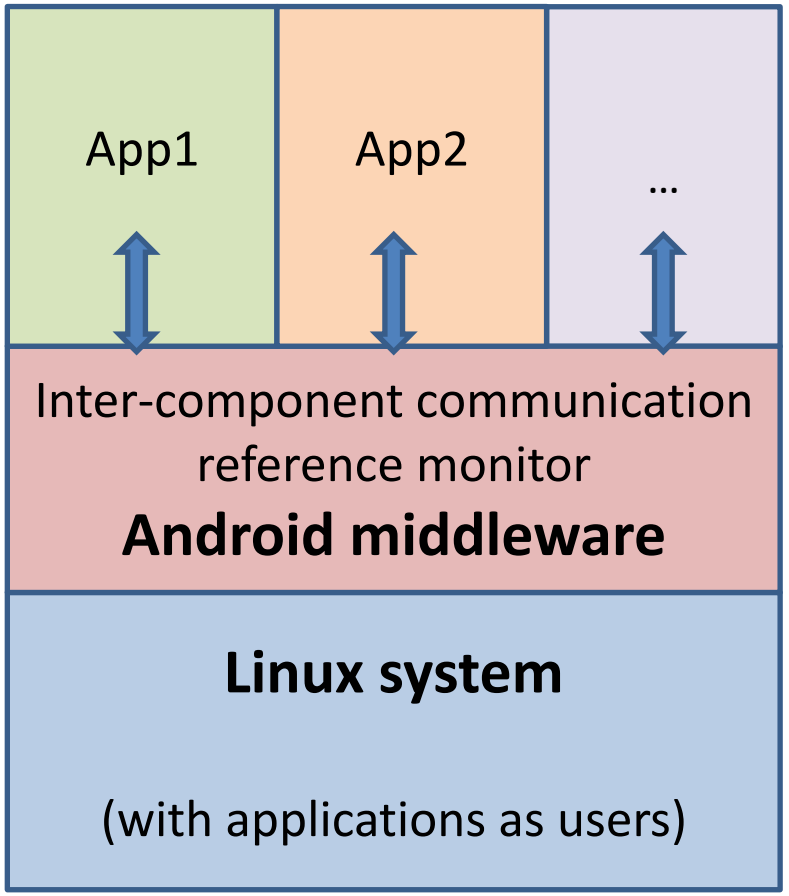
Further work

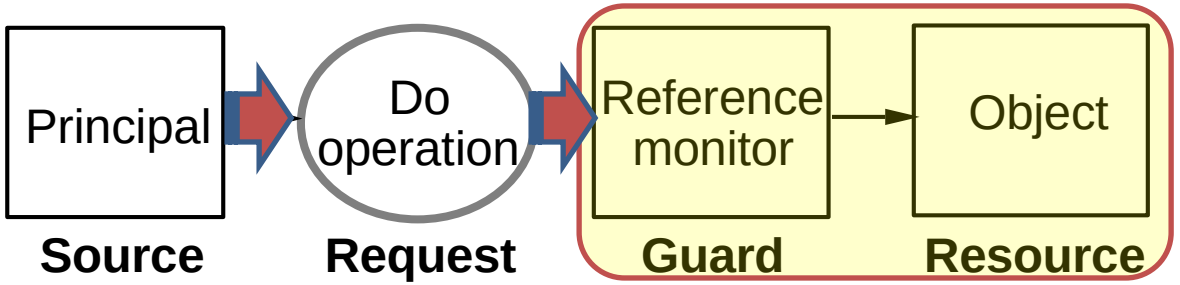
- Theorems, in particular noninterference.
- More general versions, with more levels, etc..
- Use in languages and systems.
- Connections to access control.

- For richer, more useful and real systems, see in particular Jif [Myers et al.].

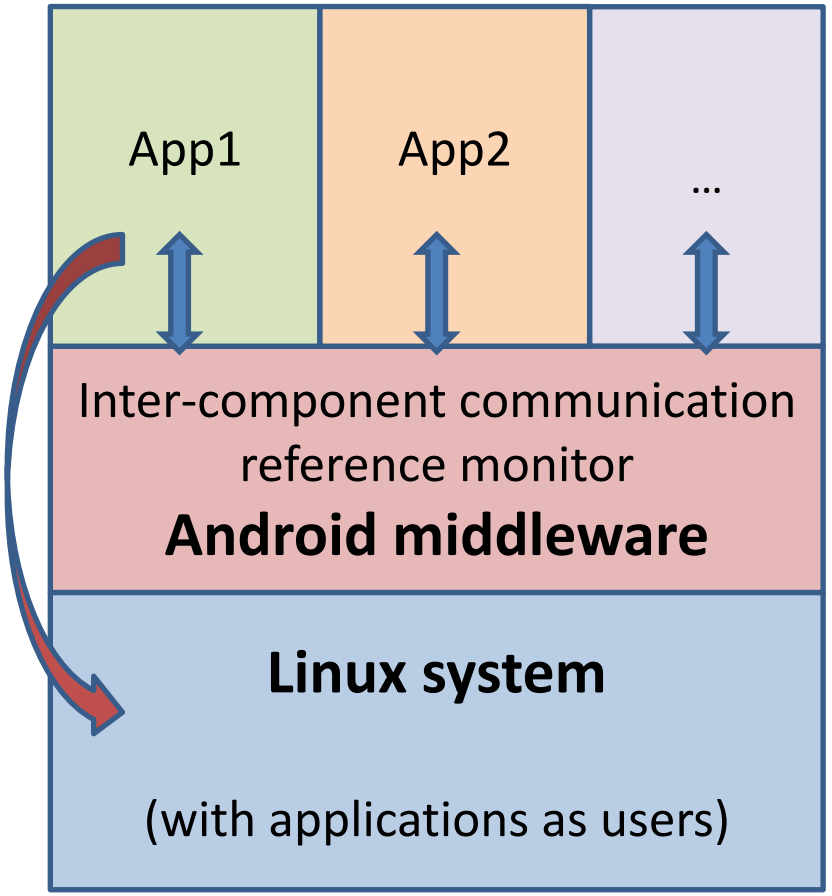
Low-level software security

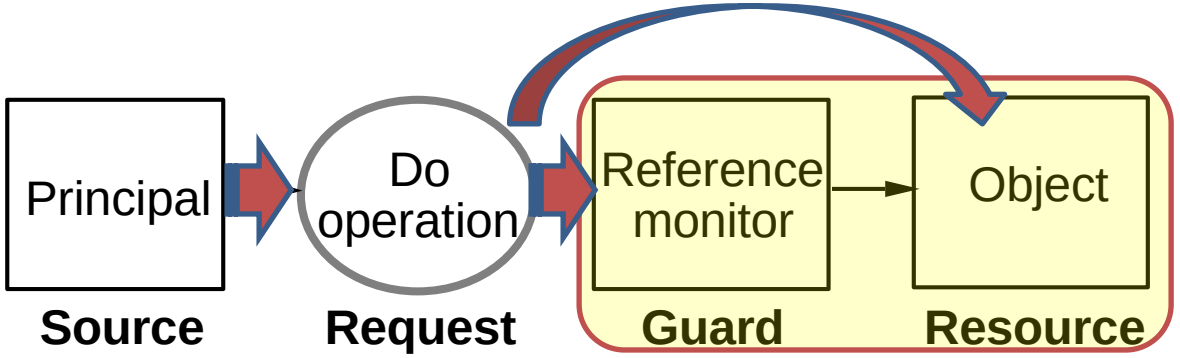
Pictures such as these ones make sense only if a component cannot circumvent or hijack other components.



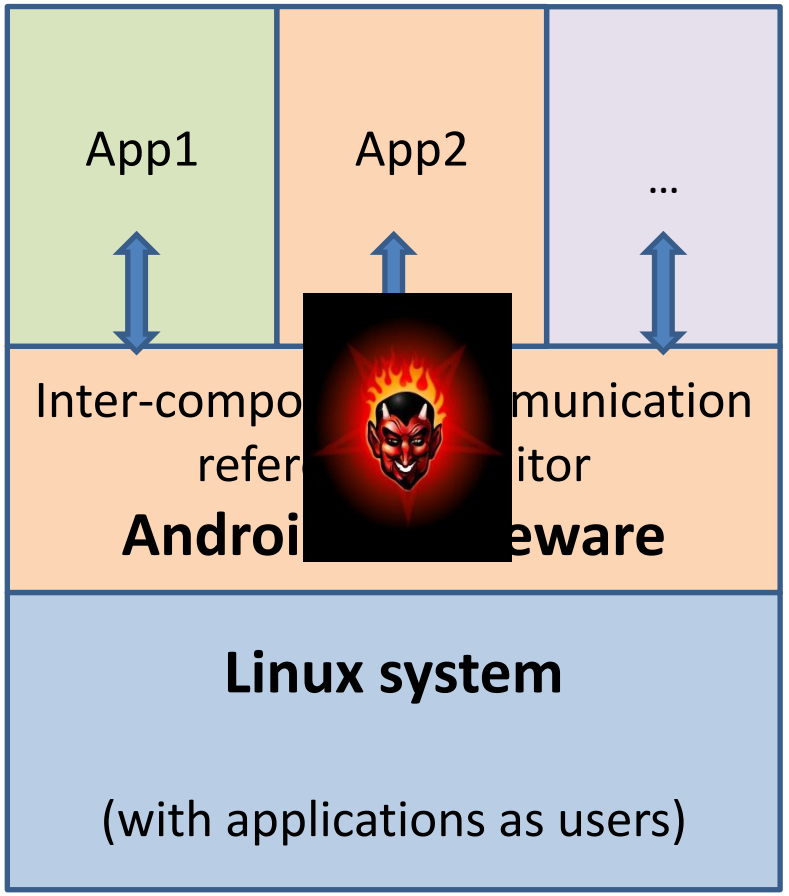


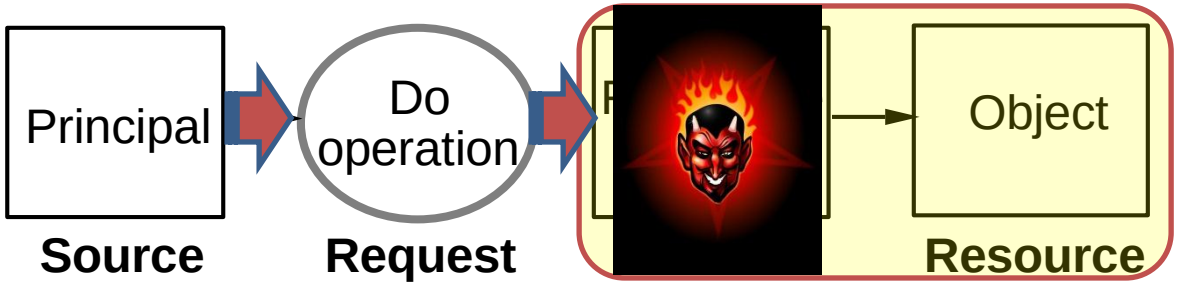
Pictures such as these ones make sense only if a component cannot circumvent or hijack other components.





Pictures such as these ones make sense only if a component cannot circumvent or hijack other components.





Flaws

- Circumvention and hijacking are common in security in many realms.
  - Tanks drive around fortifications.
  - Robbers bribe bank guards.

- In computer systems, they are sometimes the consequence of design weaknesses.
- But many result from implementation flaws: small but catastrophic errors in code.

An example

An example

```
int f(int x, char y)  {
    char t[16] ;
    initialize(t) ;
    t[x] = y ;
    return 0 ;
   }
```

An example

```
int f(int x, char y)  {
    char t[16] ;
    initialize(t) ;
    t[x] = y ;
    return 0 ;
   }
```

So what?

- Threat model: The attacker chooses inputs.

$\Rightarrow$ The attacker can (try to) modify a location of their choice at some offset from t's address.

- Some possible questions:
  - Can the attacker find the vulnerability and call f?
  - Can the attacker identify good target locations?
  - Can the attacker predict t's address?
  - Will the exploit work reliably? cause crashes?

Going further: two examples

[from Chen, Xu, Sezer, Gauriar, and Iyer]

- Attack NULL-HTTPD (a Web server on Linux).
  - POST commands can trigger a buffer overflow.

  Change the configuration string of the CGI-BIN path:
  - The mechanism of CGI:
    - Server name = www.foo.com
    - CGI-BIN = /usr/local/httpd/exe
    - Request URL = http://www.foo.com/cgi-bin/bar

    → Normally, the server runs /usr/local/httpd/exe/bar
  - An attack:
    - Exploiting the buffer overflow, set CGI-BIN = /bin
    - Request URL = http://www.foo.com/cgi-bin/sh

    → The server runs /bin/sh

⇒ ***The attacker gets a shell on the server.***

- Attack SSH Communications SSH Server:

```
void do_authentication(char *user, ...) {
    int auth = 0;                    /* initially auth is false  */
    ...
    while (!auth) {
   /* Get a packet from the client */
      type = packet_read(); /* has overflow bug                  */
      switch (type) {       /* can make auth true                */
      ...
      case SSH_CMSG_AUTH_PASSWORD:
       if (auth_password(user, password))
            auth = 1;
      case ...
      }
      if (auth) break;
   }
 /* Perform session preparation. */
 do_authenticated(…);
}
```

⇒ ***The attacker circumvents authentication.***

- Attack SSH Communications SSH Server:

```
void do_authentication(char *user, ...) {
    int auth = 0;                    /* initially auth is false  */
    ...
    while (!auth) {
   /* Get a packet from the client */
      type = packet_read(); /* has overflow bug                  */
      switch (type) {       /* can make auth true                */
      ...
      case
       if

      case
      }
      if (a
   }
 /* Perform
 do_authenticated(...);
}
```

- These are ***data-only*** attacks.
- The most classic attacks often inject code.
- Injecting code is also central in higher-level attacks such as SQL injection and XSS.

⇒ ***The attacker circumvents authentication.***

Run-time protection: the arms race

- Many attack methods:
  - Buffer overflows
  - Jump-to-libc exploits
  - Use-after-free exploits
  - Exception overwrites
  - …

- Many defenses:
  - Stack canaries
  - Safe exception handling
  - NX data
  - Layout randomization
  - …
- Not necessarily perfect in a precise sense
- Nor all well understood
- But useful mitigations

See Erlingsson's "Low-level Software Security: Attacks and Defenses".

New Windows zero-day surfaces as researcher releases attack code

SMB bug could be exploited on Windows XP, Server 2003 to hijack machines, say experts

By Gregg Keizer

February 15, 2011 03:59 PM ET

COMPUTERWORLD

Secunia added that a buffer overflow could be triggered by sending a too-long Server Name string in a malformed Browser Election Request packet. In this context, "browser" does not mean a Web browser, but describes other Windows components which access the OS' browser service.

A buffer overflow

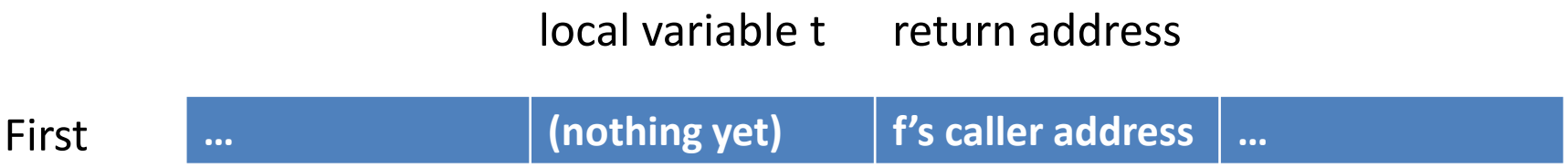
```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

- The expectation is that the contents of arg is at most of size n.

A buffer overflow

```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

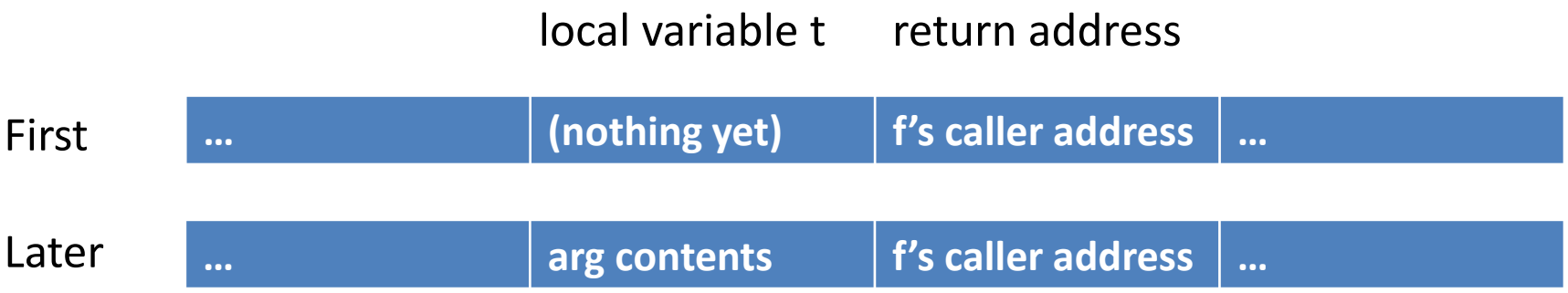
- The expectation is that the contents of arg is at most of size n.
- In memory, we would have:



A buffer overflow

```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

- The expectation is that the contents of arg is at most of size n.
- In memory, we would have:



A buffer overflow

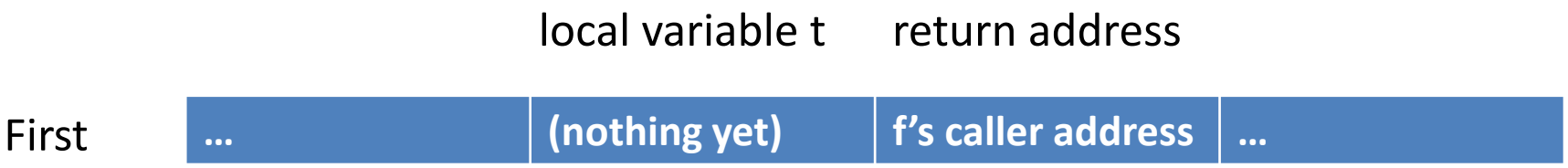
```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

- If this size is too big and not checked (either statically or dynamically), there can be trouble.

A buffer overflow

```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

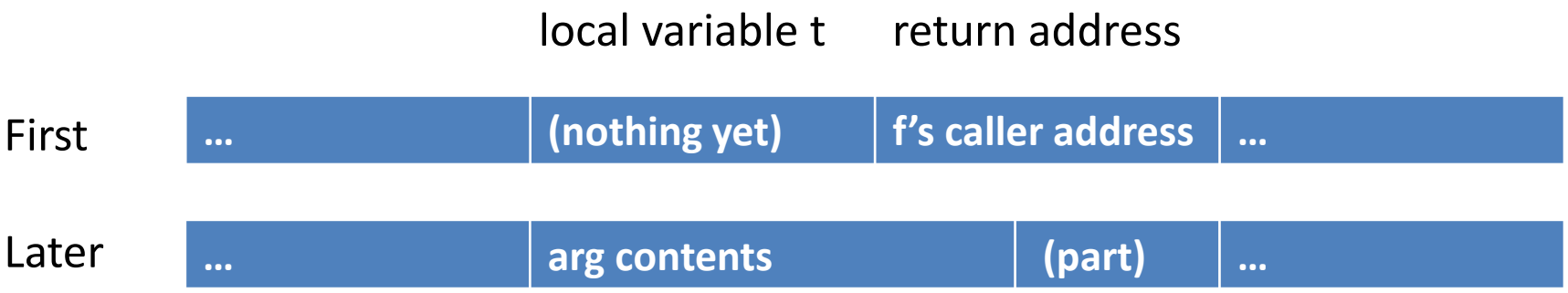
- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could have:



A buffer overflow

```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

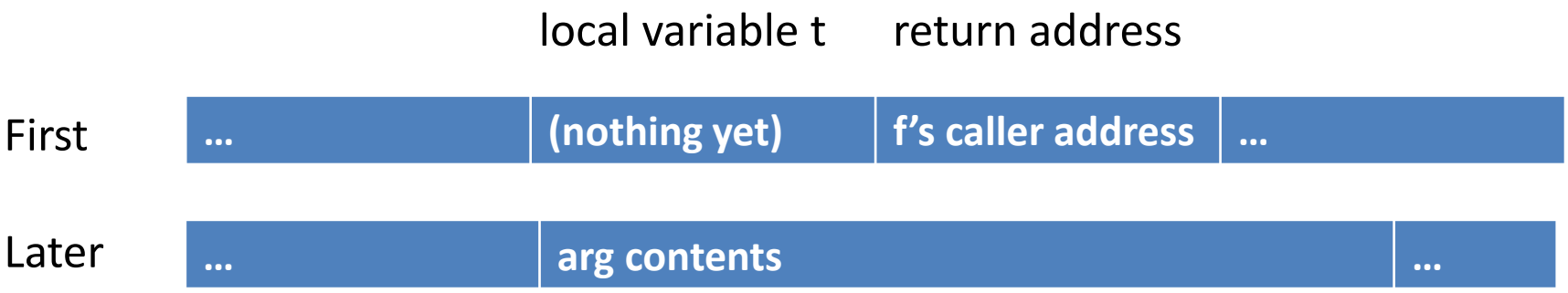
- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could have:



A buffer overflow

```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

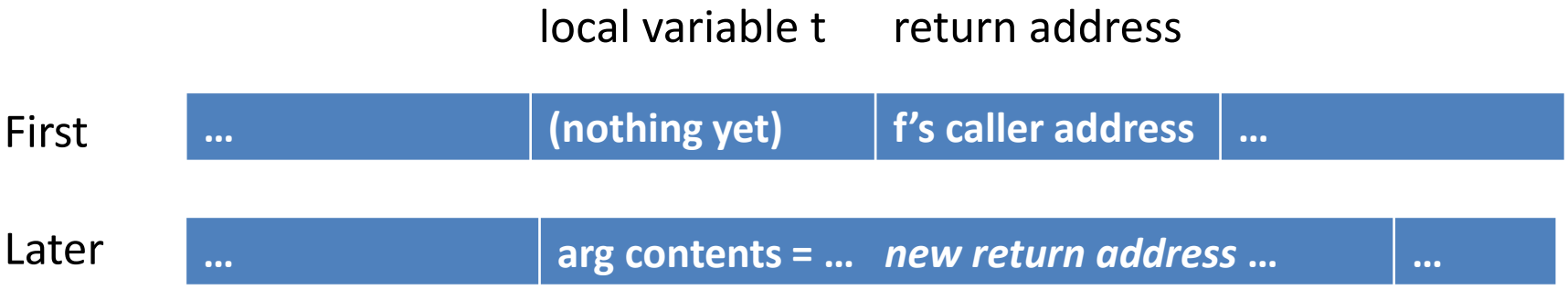
- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could also have:



A buffer overflow

```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

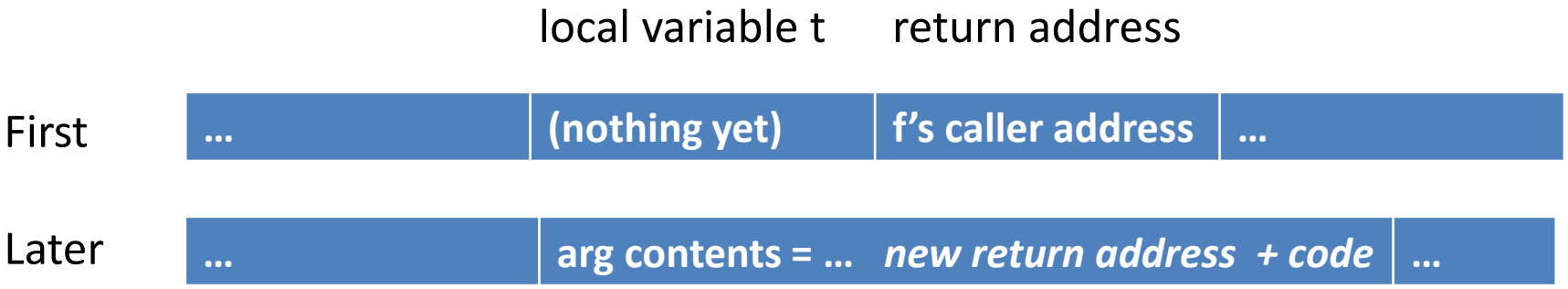
- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could also have:



A buffer overflow

```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

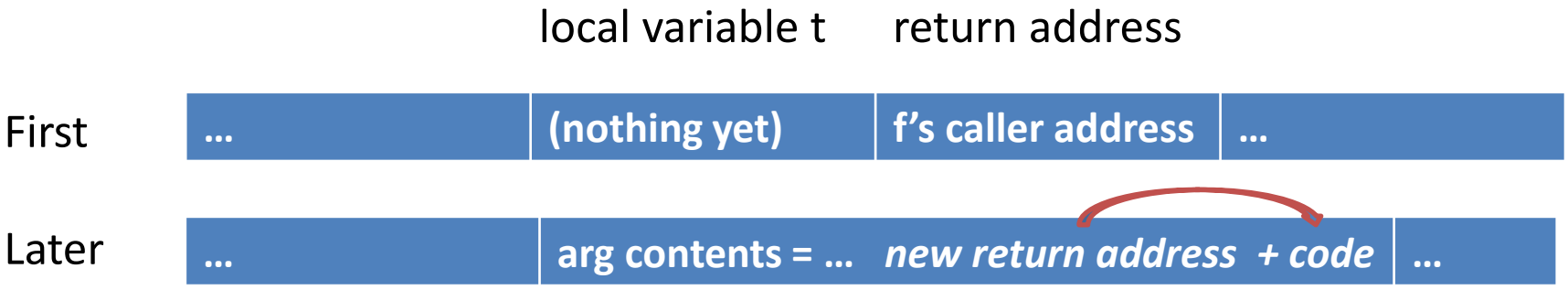
- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could also have:



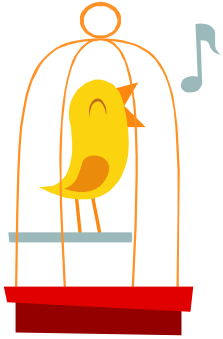
A buffer overflow

```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could also have:

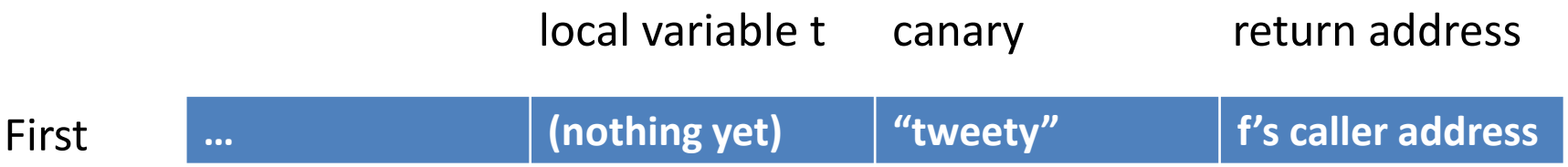


Stack canaries and cookies



```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

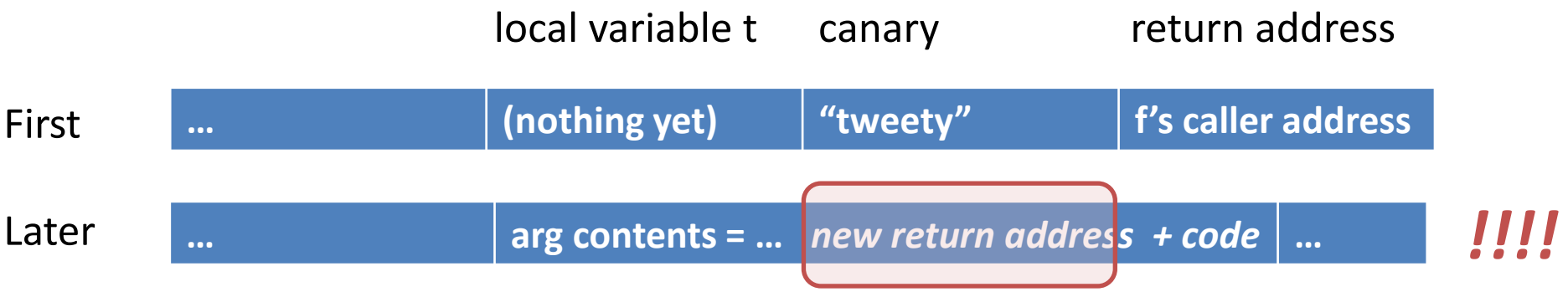
- A known quantity (fixed or random) can be inserted between the local variable and the return address so that any corruption can be detected.



Stack canaries and cookies


```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

- A known quantity (fixed or random) can be inserted between the local variable and the return address so that any corruption can be detected.



There are more things

- Stack canaries and cookies can be effective in impeding many buffer overflows on the stack.

But:

- They need to be applied consistently.
- Sometimes they are judged a little costly.
- They do not help if corrupted data (e.g., a function pointer) is used before the return.
- And there are many kinds of overflows, and many other kinds of vulnerabilities.

NX (aka DEP)

Many attacks rely on injecting code.

⇒ ***So a defense is to require that data that is writable cannot be executed.***

- This requirement is supported by mainstream hardware (e.g., x86 processors).

NX (aka DEP)

Many attacks rely on injecting code.

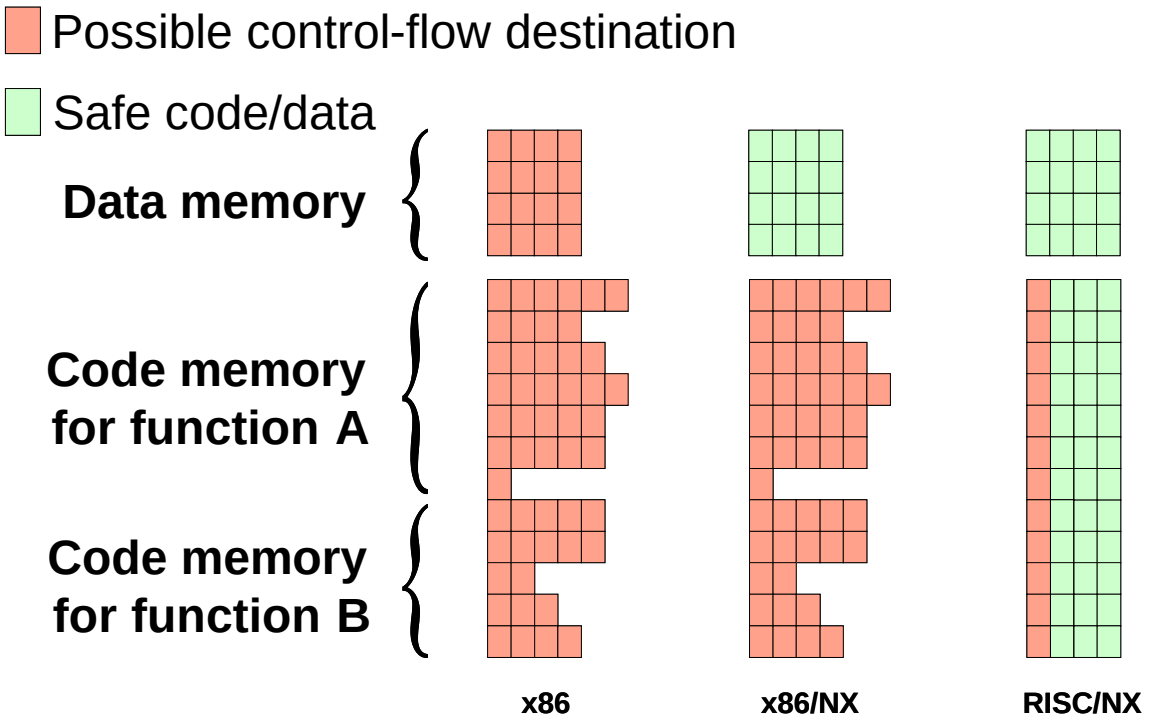
⇒ ***So a defense is to require that data that is writable cannot be executed.****

- This requirement is supported by mainstream hardware (e.g., x86 processors).

** An exception must be made in order to allow compilation (e.g., JIT compilation for JavaScript).*

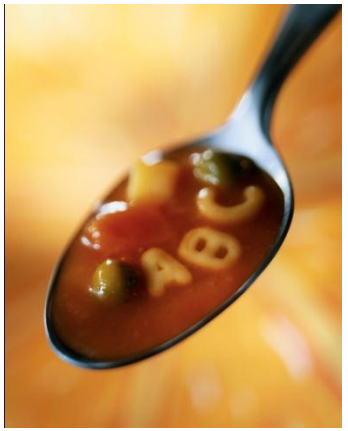
What bytes will the CPU interpret?

- Mainstream hardware typically places few constraints on control flow.
- A call can lead to many places:



Executing existing code

- With NX defenses, attackers cannot simply inject data and then run it as code.
- But attackers can still run existing code:
  - the intended code in an unintended state,
  - an existing function, such as `system()`,
  - even dead code,
  - even code in the middle of a function,
  - even "accidental" code (e.g., starting half-way in a long x86 instruction).



An example of accidental x86 code

[Roemer et al.]

Two instructions in the entry point ecb_crypt are encoded as follows:

f7 c7 07 00 00 00	test $0x00000007, %edi
0f 95 45 c3	setnzb -61(%ebp)

Starting one byte later, the attacker instead obtains

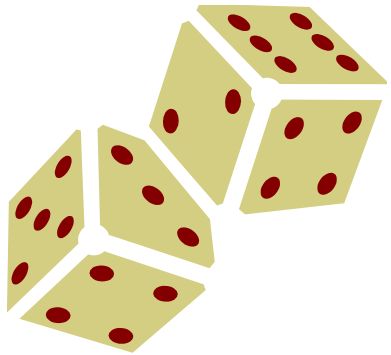
c7 07 00 00 00 0f	movl $0x0f000000, (%edi)
95	xchg %ebp, %eax
45	inc %ebp
c3	ret

Layout randomization

Attacks often depend on addresses.

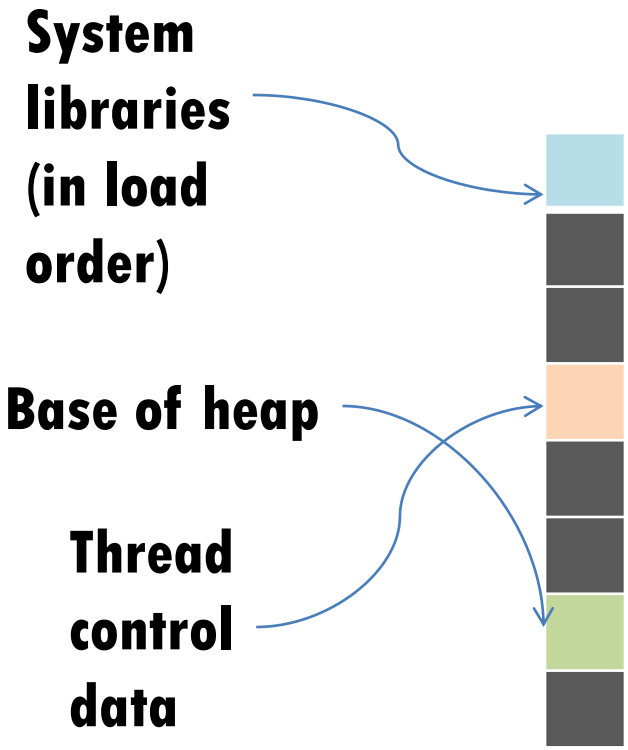
⇒ ***Let us randomize the addresses!***

- Considered for data at least since the rise of large virtual address spaces
  (e.g., [Druschel & Peterson, 1992] on fbufs).
- Now present in Linux (PaX), Windows, Mac OS X, iOS, Android (4.0).



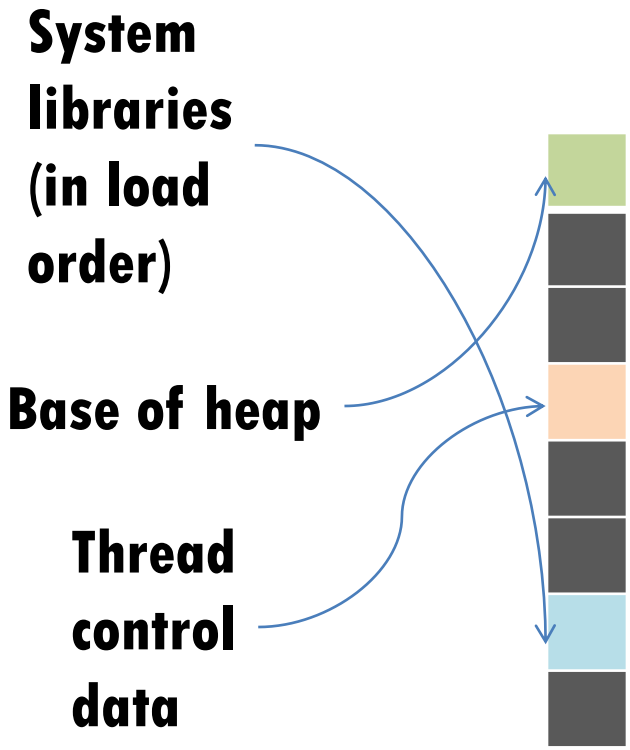
Implementations

- The randomization can be performed at build, install, boot, or load time.



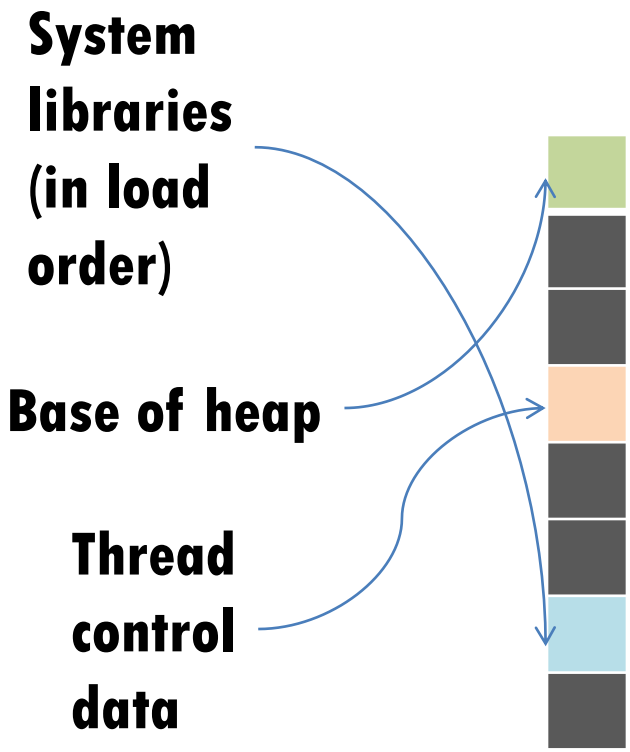
Implementations

- The randomization can be performed at build, install, boot, or load time.



Implementations

- The randomization can be performed at build, install, boot, or load time.
- It may be at various granularities.
- It need not have performance cost, but it may complicate compatibility.



A theory of layout randomization

[with Gordon Plotkin, now Jérémy Planul]

- Define *high-level programs*, with symbolic locations (e.g., $l := 3$), and *low-level programs*, with numbers as addresses (e.g., $8686 := 3$).

  → View randomization as part of a translation.

A theory of layout randomization

[with Gordon Plotkin, now Jérémy Planul]

- Define *high-level programs*, with symbolic locations (e.g., $l := 3$), and *low-level programs*, with numbers as addresses (e.g., $8686 := 3$).

  → View randomization as part of a translation.

- View attackers as contexts, i.e., other programs with which our programs interact.

  → Relate low-level contexts to high-level contexts.

A theory of layout randomization

[with Gordon Plotkin, now Jérémy Planul]

- Define *high-level programs*, with symbolic locations (e.g., $l := 3$), and *low-level programs*, with numbers as addresses (e.g., $8686 := 3$).

  → View randomization as part of a translation.

- View attackers as contexts, i.e., other programs with which our programs interact.

  → Relate low-level contexts to high-level contexts.

- Phrase security properties as equivalences.

  → Study whether equivalences are preserved.

The source language

- Higher-order lambda calculus,
- with read/write/execute operations on locations that hold natural numbers,
- with standard base types and optionally a type of locations,
- also sometimes with an error constant (which we assume here).

Syntax

- Types:

$$\sigma ::= b \mid \texttt{unit} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \to \sigma$$

where b ranges over basic types which always include **`nat`** and may include **`loc`**.

Syntax (cont.)

- Programs:

$$
\begin{array}{rcl}
M & ::= & x \mid c \mid * \mid (M, M) \mid \mathtt{fst}\, M \mid \mathtt{snd}\, M \mid \\
& & \mathtt{inl}_{\sigma,\sigma}\, M \mid \mathtt{inr}_{\sigma,\sigma}\, M \mid \\
& & \mathtt{cases}\, M\, \mathtt{inl}\, x{:}\sigma.\, M\, \mathtt{inr}\, x{:}\sigma.\, M \mid \\
& & \lambda x{:}\sigma.\, M \mid M\, M \mid \mathtt{rec}(f{:}\sigma \to \tau, x{:}\sigma).\, M
\end{array}
$$

where c ranges over constants, each of a unique type. These include the natural numbers, the usual arithmetic operations, constants for memory access (e.g., run, $:=$), and constants for raising errors.

Memory access

(some specifics)

- Memory-access constants:

$$l : \mathtt{loc}\ (l \in \mathrm{Loc})$$
$$!_{\mathtt{loc}} : \mathtt{loc} \to \mathtt{nat}$$
$$:=_{\mathtt{loc}} : \mathtt{loc} \times \mathtt{nat} \to \mathtt{unit}$$
$$\mathtt{run}_{\mathtt{loc}} : \mathtt{loc} \to \mathtt{unit}$$

- Some semantics:

$$(s, !_{\mathtt{loc}} l) \longrightarrow (s, n) \quad (\text{if } s(l) = n)$$
$$(s, l :=_{\mathtt{loc}} n) \longrightarrow (s[l \mapsto n], *) \quad (\text{if } l \in \mathrm{DataLoc})$$
$$(s, \mathtt{run}_{\mathtt{loc}} l) \longrightarrow (s', *) \quad (\text{if } l \in \mathrm{CodeLoc}, s(l) = n, s' = Dc(n)(s))$$

where a **store** s is a function from Loc to natural numbers, and Dc is an "instruction decoding" function.

The target language

- Much like the source language,
- but with natural-number addresses rather than locations.

$$l : \mathtt{nat} \quad (\text{for } l \in \mathrm{Loc})$$
$$!_{\mathtt{nat}} : \mathtt{nat} \to \mathtt{nat}$$
$$:=_{\mathtt{nat}} : \mathtt{nat} \times \mathtt{nat} \to \mathtt{unit}$$
$$\mathtt{run}_{\mathtt{nat}} : \mathtt{nat} \to \mathtt{unit}$$

The target model(s), informally

- A **layout** w is a function $\mathrm{Loc} \hookrightarrow \{0, \ldots, c\}$ chosen at random (for instance, uniformly).
- A **memory** m is a function: $\{0, \ldots, c\} \longrightarrow \mathbb{N} + \mathbf{1}$
  - Memory may be accessed directly through natural-number addresses.
  - Some addresses may be unused.
- Accesses to unused addresses are either **fatal errors** or **recoverable errors**.
  - These two variants both make sense, but lead to different results.

Attackers as contexts

- A **public program** is one that cannot access private locations directly. I.e.:
  - Our languages have constants for locations (Loc).
  - We distinguish sets of public locations (PubLoc) and private locations (PriLoc).
  - Private ones cannot occur in public programs.
- For us, attackers are public contexts.

Equivalences

In the source language, two programs are ***publically equivalent*** *if no public context can distinguish them:*

for M, N of the same type σ, $M \approx_{h,p} N$
iff for every initial store s, every public C of type $\sigma \to \texttt{bool}$
(1) CM and CN both diverge,
(2) or they both give an error,
(3) or they both yield the same result value and two new stores that coincide on PubLoc.

In the target language, $M \approx_{l,p} N$ *is similar, but with probabilities (over the choice of layout).*

Equivalences (cont.)

Secrecy and integrity properties can be phrased as public equivalences.

E.g., for a private location l

$$l := c \;\approx_{h,p}\; l := c'$$

$$\begin{array}{l} \lambda f : \mathtt{nat} \to \mathtt{unit}. \\ \quad l := c; \\ \quad f(c); \\ \quad \mathtt{if}\, !l = c\, \mathtt{then}\, l' := c\, \mathtt{else}\, l' := c' \end{array} \quad \approx_{h,p} \quad \begin{array}{l} \lambda f : \mathtt{nat} \to \mathtt{unit}. \\ \quad l := c; \\ \quad f(c); \\ \quad l' := c \end{array}$$

Preserving equivalences

("full abstraction")

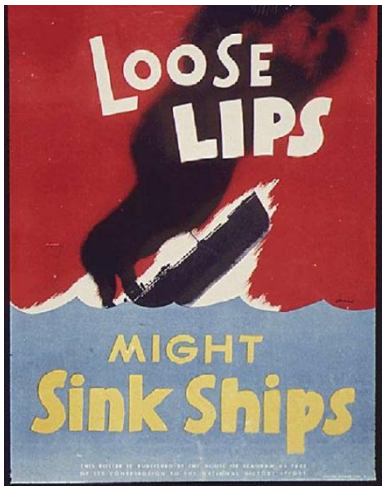
With each high-level program M
we associate a low-level program $M^{\downarrow}$.

Theorem: Suppose that M and N are high-level terms of type σ. Assume that σ is **`loc`**-free.

If $M \approx_{h,p} N$ then $M^{\downarrow} \approx_{l,p} N^{\downarrow}$.

Layout randomization depends on secrecy, but…

- The secrecy is not always strong.
  - E.g., there cannot be much address randomness on 32-bit machines.
  - E.g., low-order address bits may be predictable.
- The secrecy is not always well-protected.
  - Pointers may be disclosed.
  - Functions may be recognized by their behavior.



Layout randomization depends on secrecy, but…

- This secrecy is not always effective.
  - "Heap spraying" can fill parts of the address space predictably, including with JIT-compiled code.

Browser

A nice Web site that attracts traffic

(owned by the attacker)

Layout randomization depends on secrecy, but…

- This secrecy is not always effective.
  - "Heap spraying" can fill parts of the address space predictably, including with JIT-compiled code.

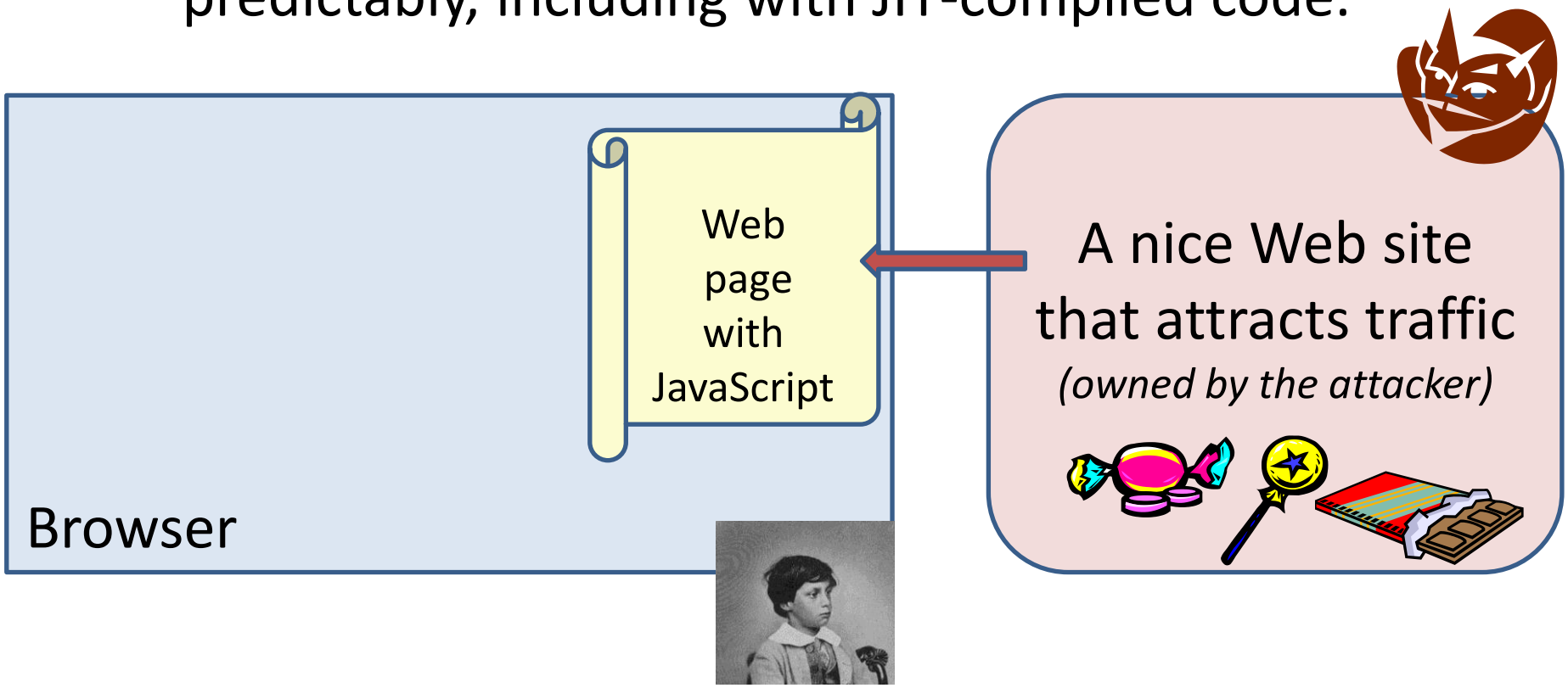
Browser

A nice Web site that attracts traffic

(owned by the attacker)

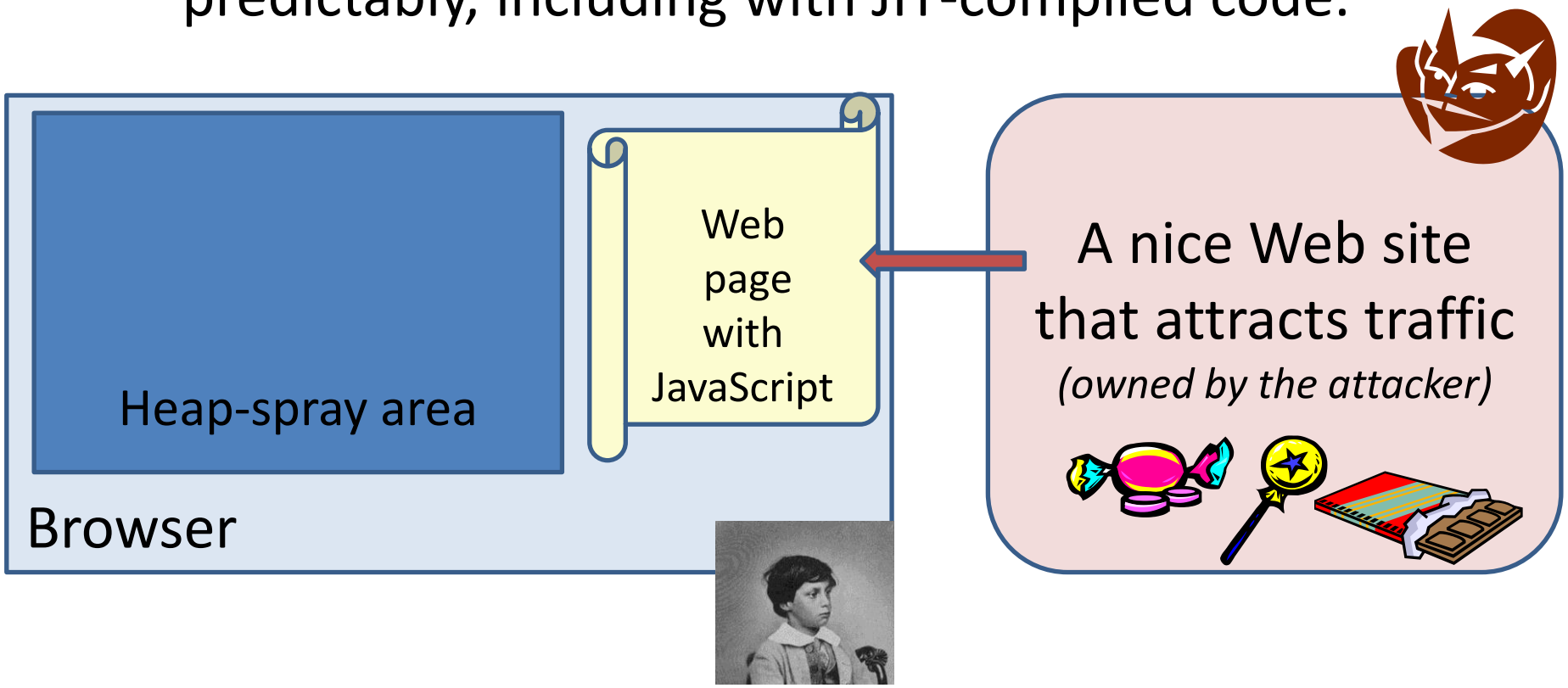
Layout randomization depends on secrecy, but…

- This secrecy is not always effective.
  - "Heap spraying" can fill parts of the address space predictably, including with JIT-compiled code.



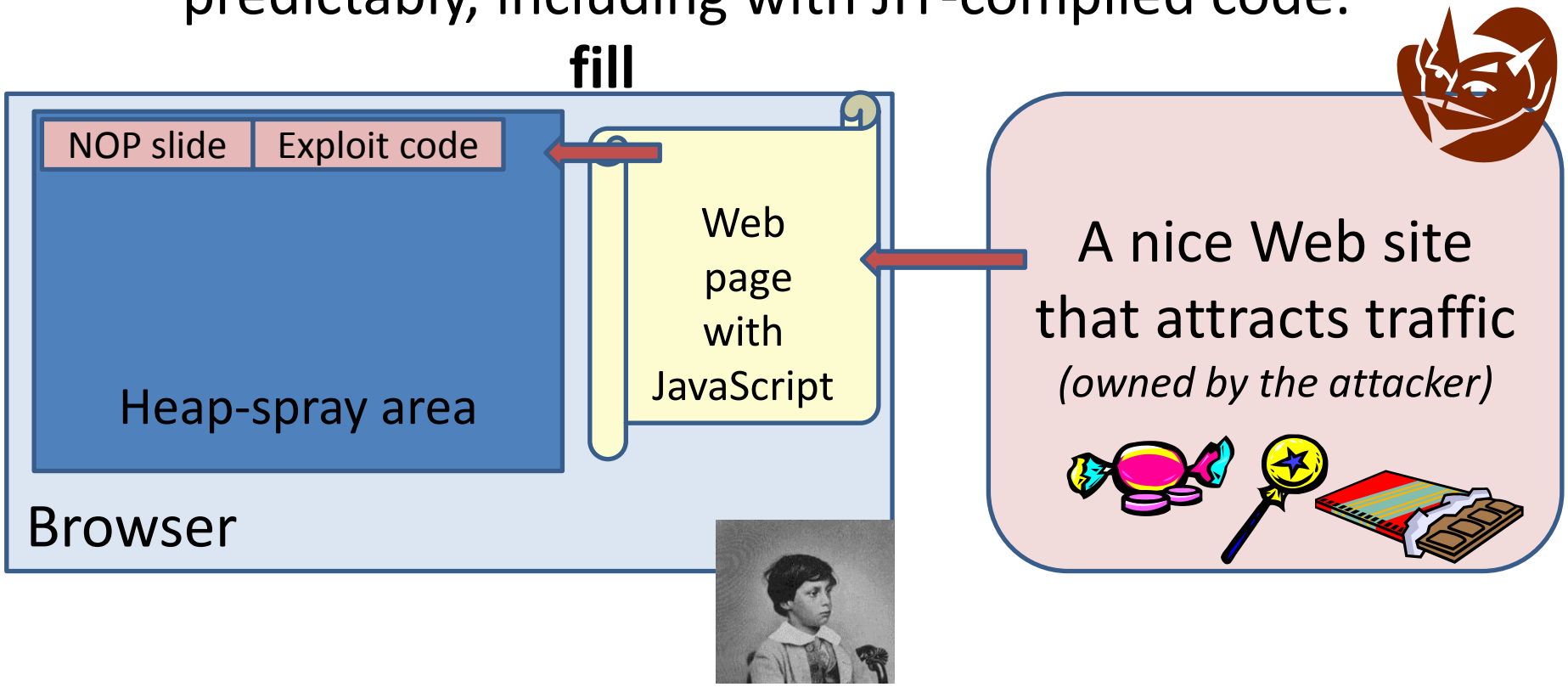
Layout randomization depends on secrecy, but…

- This secrecy is not always effective.
  - "Heap spraying" can fill parts of the address space predictably, including with JIT-compiled code.



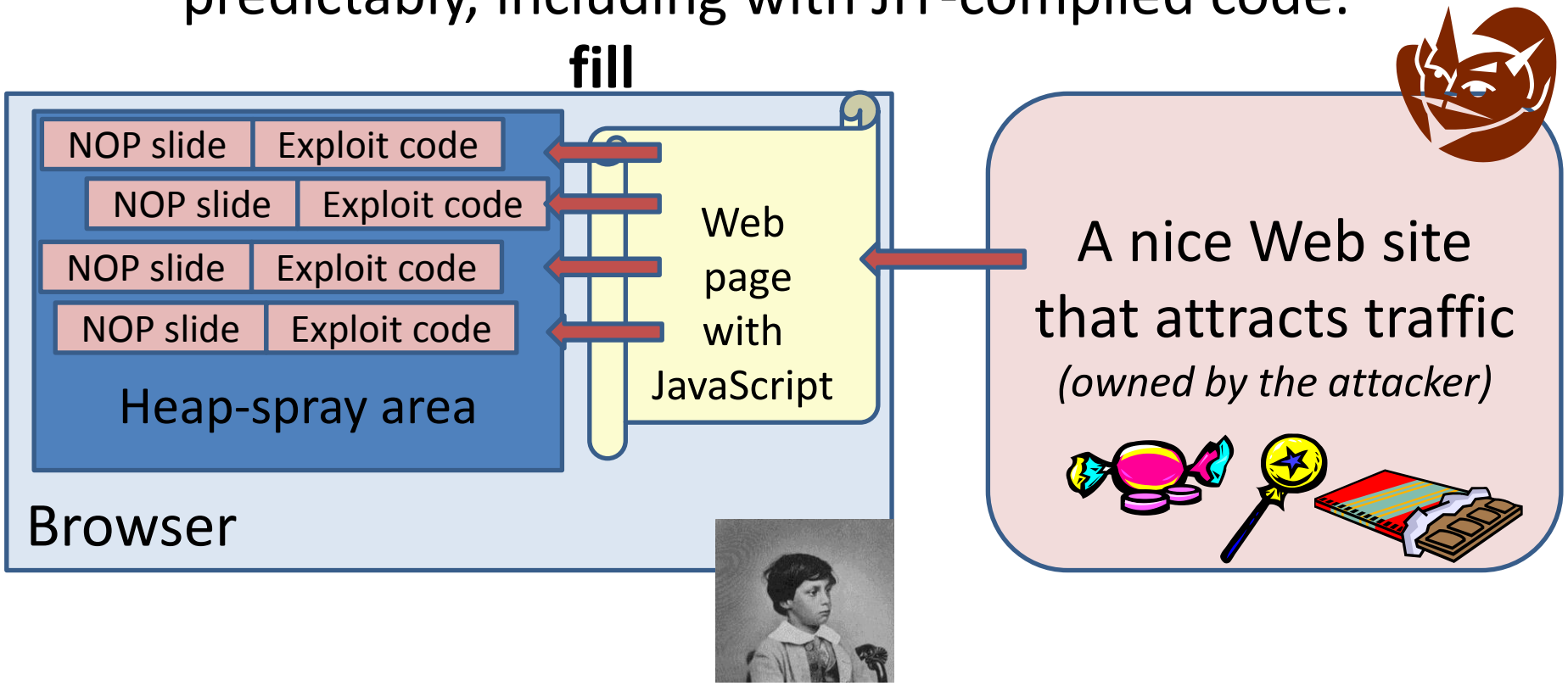
Layout randomization depends on secrecy, but…

- This secrecy is not always effective.
  - "Heap spraying" can fill parts of the address space predictably, including with JIT-compiled code.



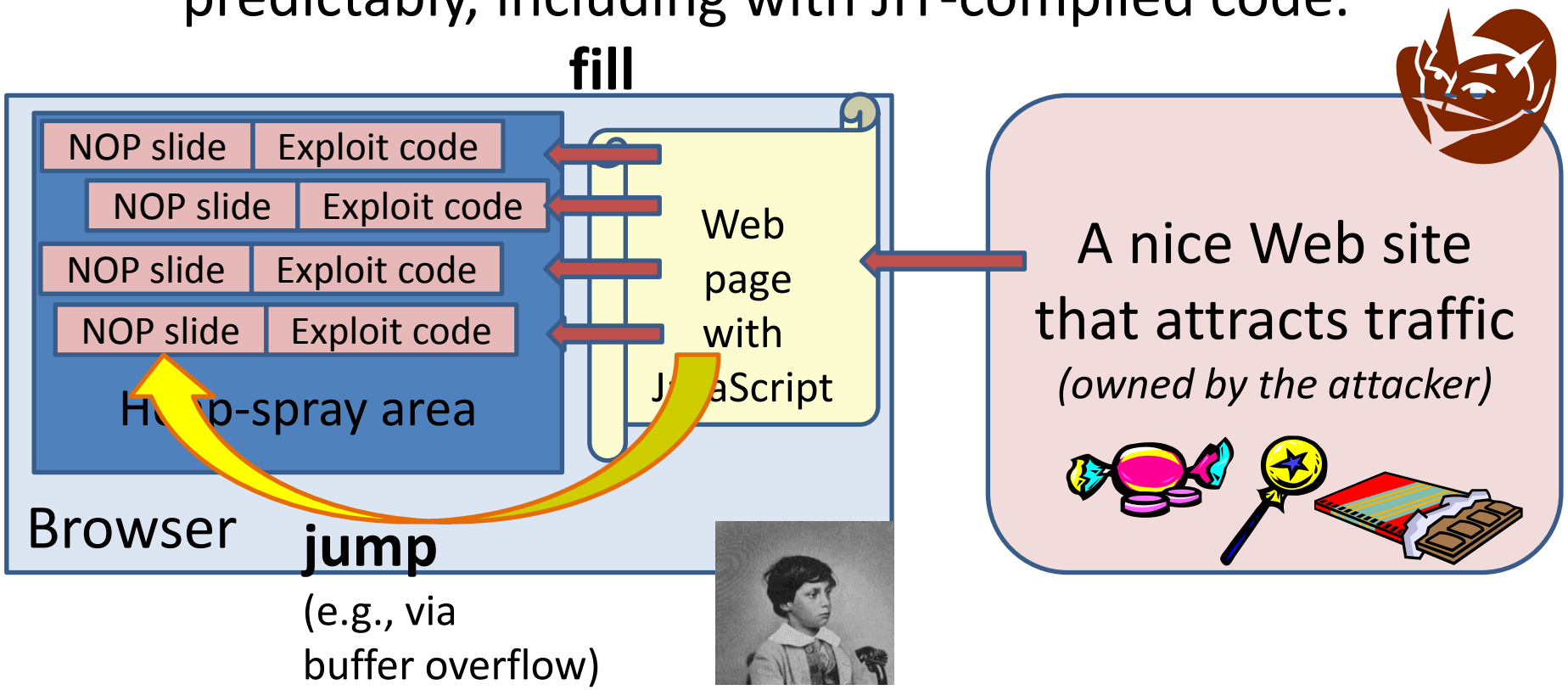
Layout randomization depends on secrecy, but…

- This secrecy is not always effective.
  - "Heap spraying" can fill parts of the address space predictably, including with JIT-compiled code.



Layout randomization depends on secrecy, but…

- This secrecy is not always effective.
  - "Heap spraying" can fill parts of the address space predictably, including with JIT-compiled code.



Layout randomization depends on secrecy, but…

- This secrecy is not always effective.
  - "Heap spraying" can fill parts of the address space predictably, including with JIT-compiled code.

Date	Browser	Description	milw0rm
11/2004	IE	IFRAME Tag BO	612
04/2005	IE	DHTML Objects Corruption	930
01/2005	IE	.ANI Remote Stack BO	753
07/2005	IE	`javaprxy.dll` COM Object	1079
03/2006	IE	`createTextRang` RE	1606
09/2006	IE	VML Remote BO	2408
03/2007	IE	ADODB Double Free	3577
09/2006	IE	WebViewFolderIcon `setSlice`	2448
09/2005	FF	0xAD Remote Heap BO	1224
12/2005	FF	`compareTo()` RE	1369
07/2006	FF	`Navigator` Object RE	2082
07/2008	Safari	Quicktime Content-Type BO	6013

Source: Ratanaworabhan, Livshits, and Zorn (2009)

Layout randomization depends on secrecy, but…

- This secrecy is not always effective.
  - “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.
  - “Heap feng shui” influences heap layout [Sotirov].
  - …

Layout randomization: status

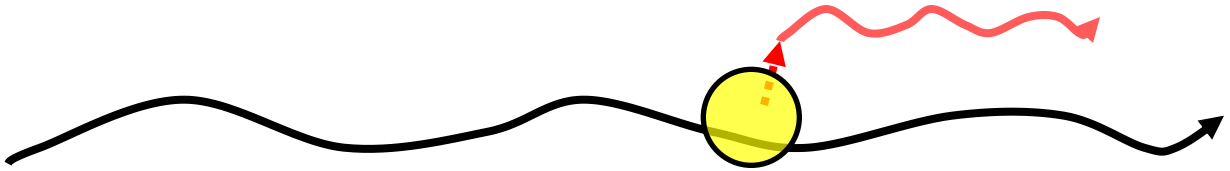
This is an active area, with

- variants and ongoing improvements to the randomization and its application,
- variants of the attacks,
- techniques detecting or mitigating the attacks.

Overall, randomization is widespread and seems quite effective but not a panacea.

Diverting control flow

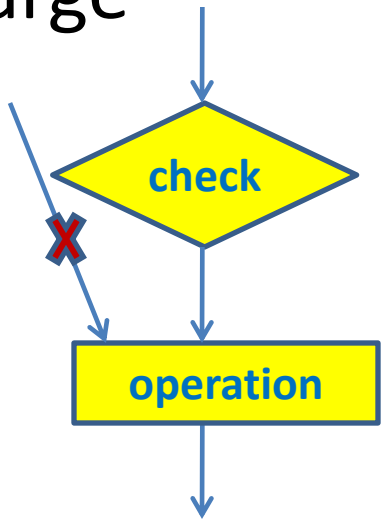
- Many attacks cause some sort of subversion of the expected control flow.



  - E.g., an argument that is “too large” can cause a function to jump to an unexpected place.
- Several techniques prevent or mitigate the effects of many control-flow subversions.
  - E.g., canaries help prevent some bad returns.

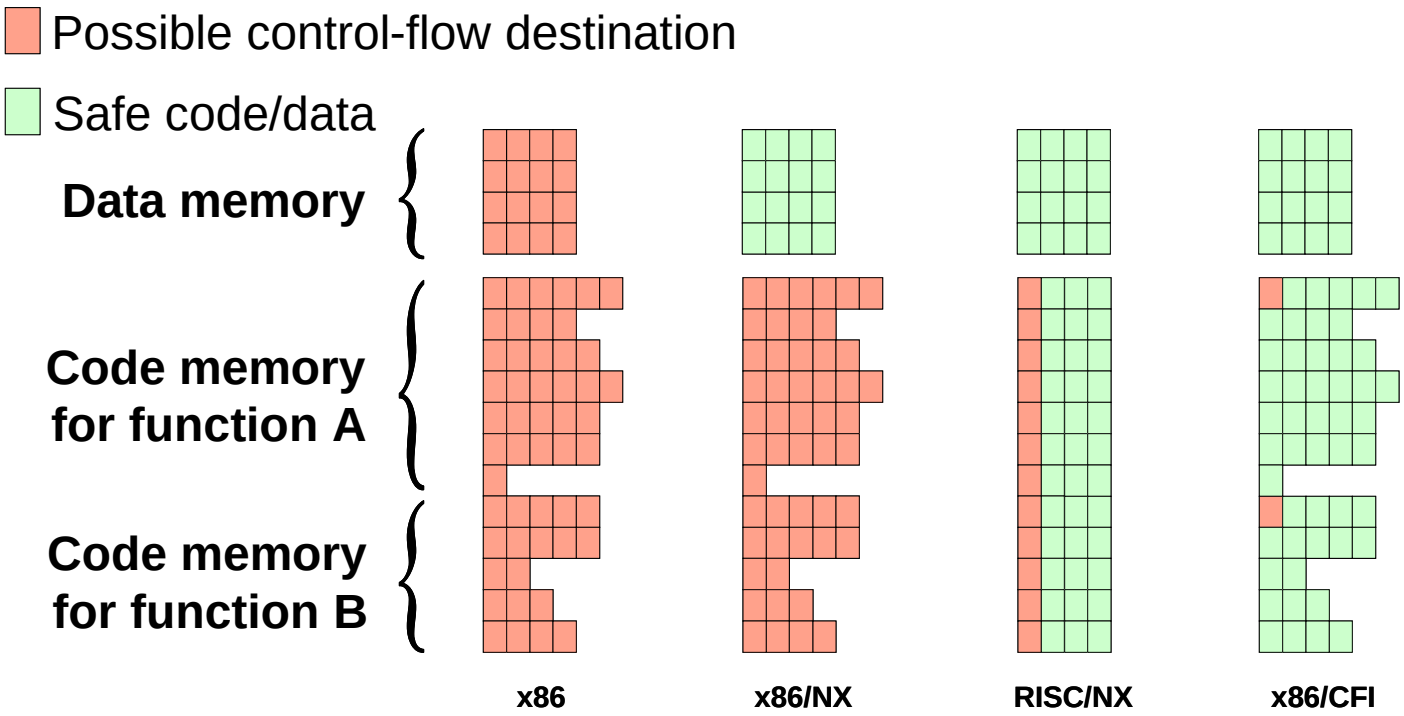
Control-flow integrity (CFI)

- CFI means that execution proceeds according to a specified control-flow graph (CFG).
- CFI is a basic property that thwarts a large class of attacks.



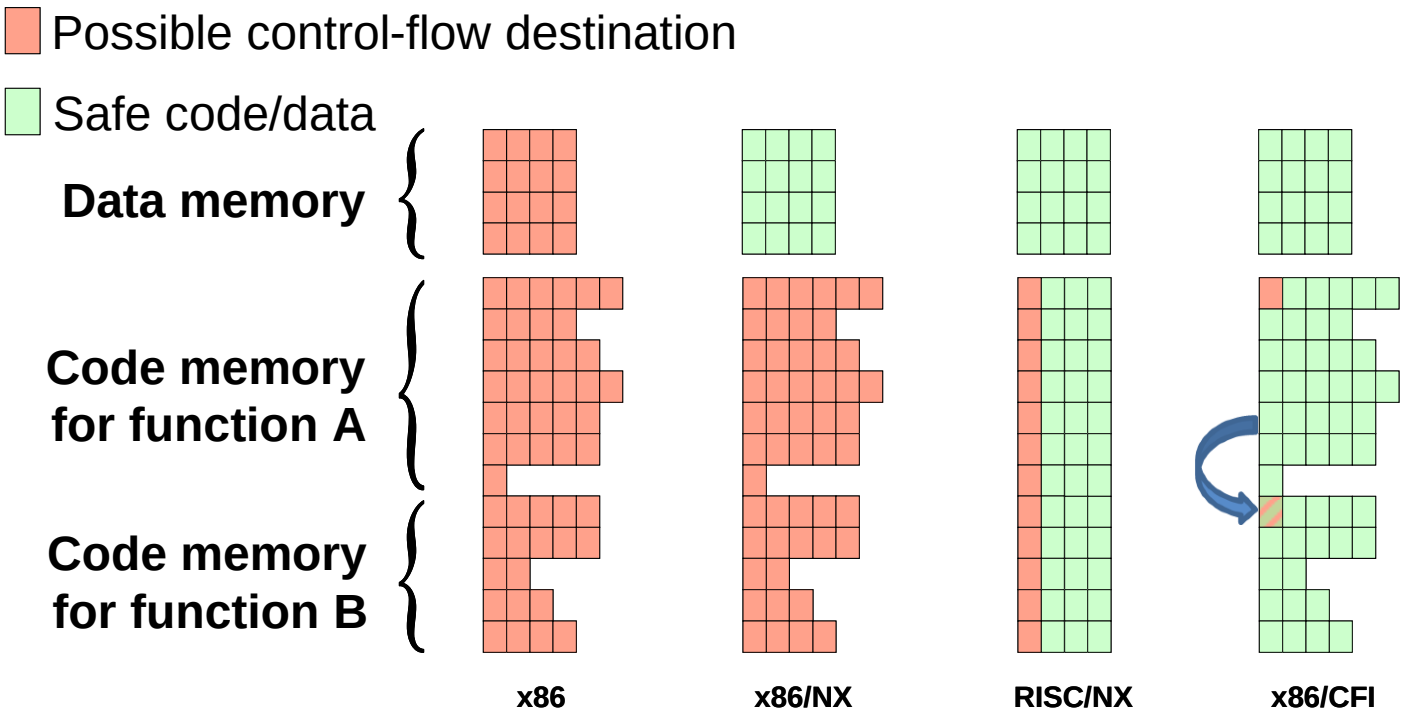
What bytes will the CPU interpret, with CFI?

- E.g., we may allow jumps to the start of any function (defined in a higher-level language):



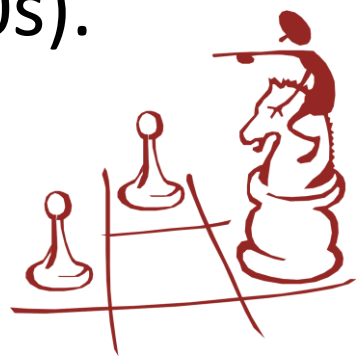
What bytes will the CPU interpret, with CFI? (cont.)

- Or we may allow jumps the start of B only from a particular call site in A:



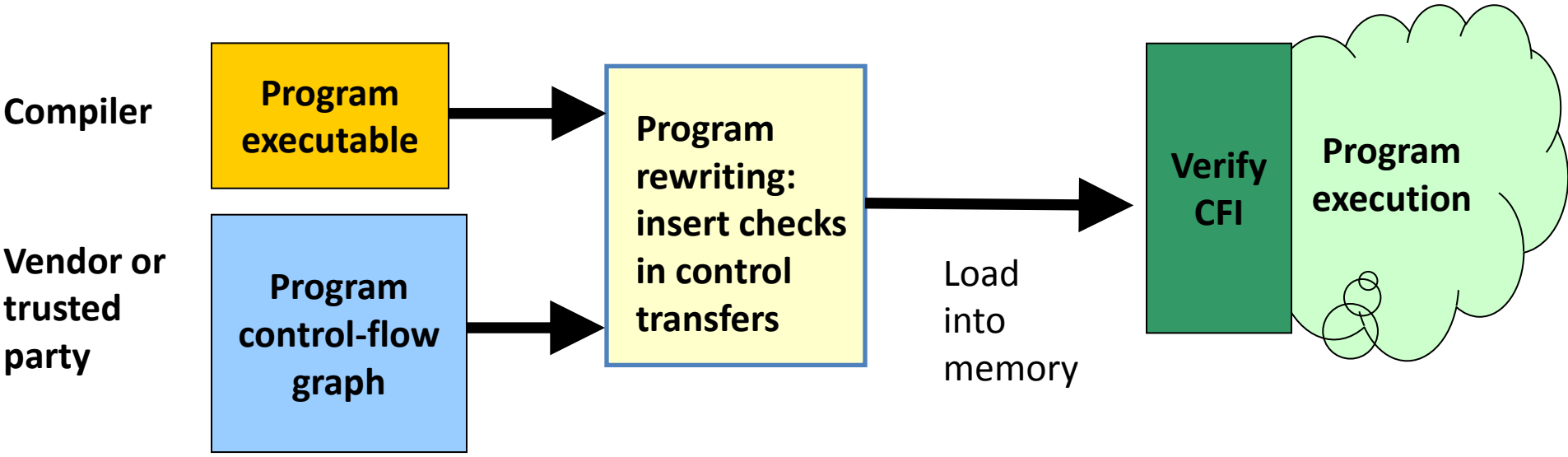
Some implementation strategies for CFI

1. A fast interpreter performs control-flow checks (“Program Shepherding”).
2. A compiler emits code with control-flow checks (as in WIT).
3. A code rewriter adds control-flow checks (as in PittSFIeld, where all control-flow targets are required to end with two 0s).



A rewriting-based system

[with Budiu, Erlingsson, Ligatti, Peinado, Necula, and Vrable]



- The rewriting inserts guards to be executed at run-time, before control transfers.
- It need not be trusted, because of the verifier.

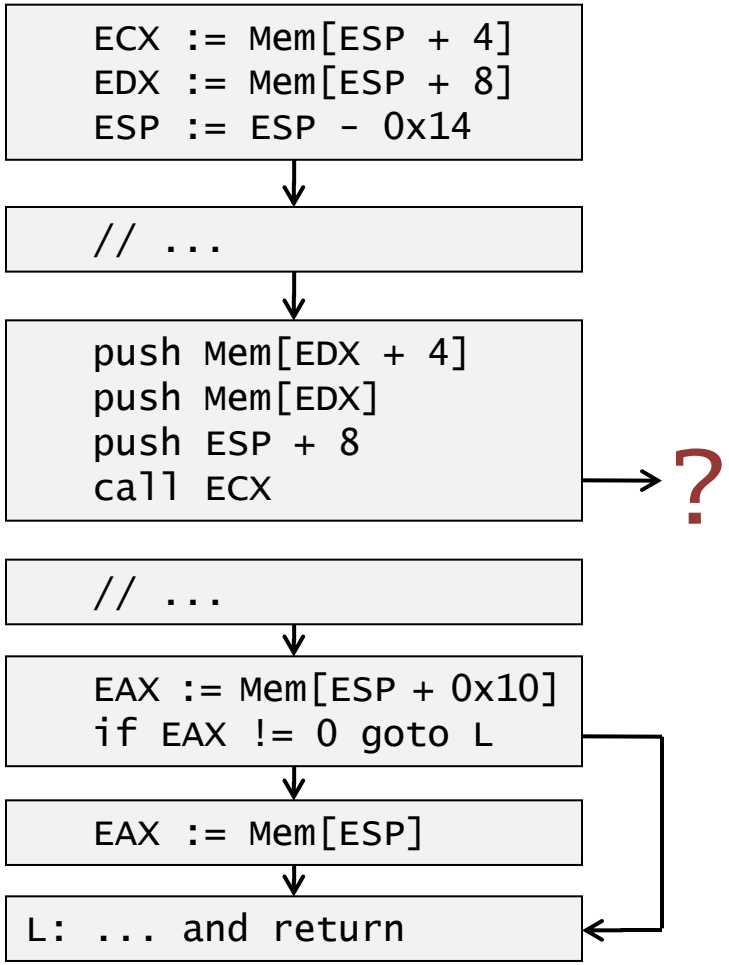
Example

- Code uses data and function pointers,
- susceptible to effects of memory corruption.

C source code

```
int foo(fptr pf, int* pm) {
  int err;
  int A[4];

  // ...

  pf(A, pm[0], pm[1]);

  // ...

  if( err ) return err;
  return A[0];
}
```

Machine-code basic blocks



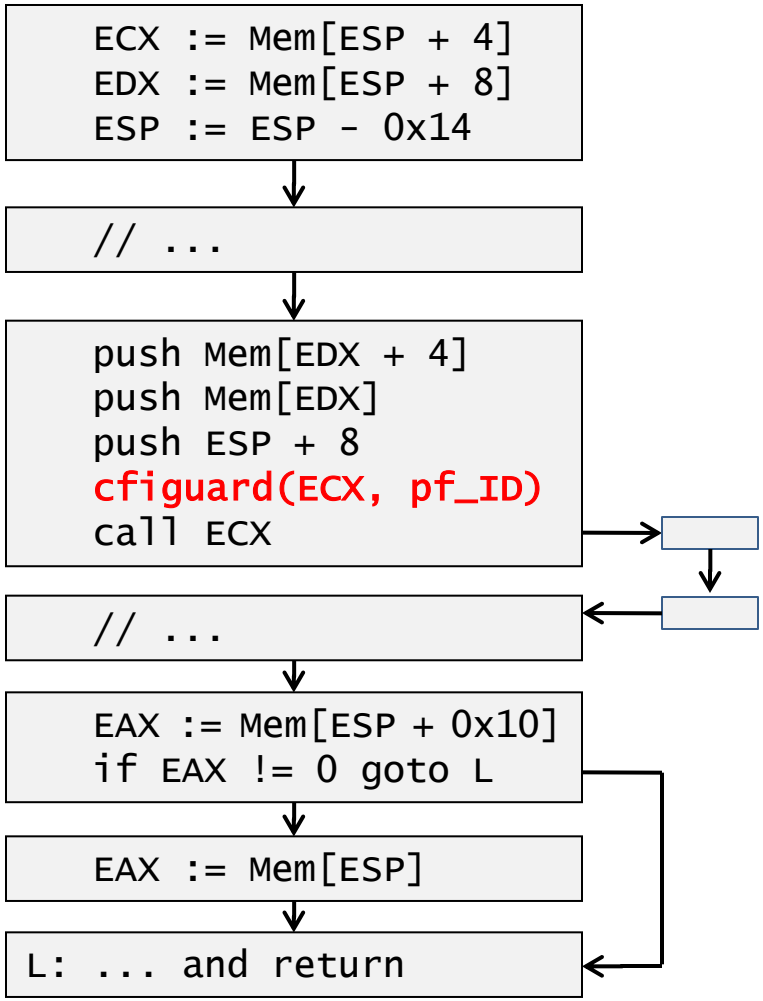
Example (cont.)

- We add guards for checking control transfers.
- These guards are “inline reference monitors”.

C source code

```
int foo(fptr pf, int* pm) {
  int err;
  int A[4];

  // ...

  pf(A, pm[0], pm[1]);

  // ...

  if( err ) return err;
  return A[0];
}
```

Machine-code basic blocks



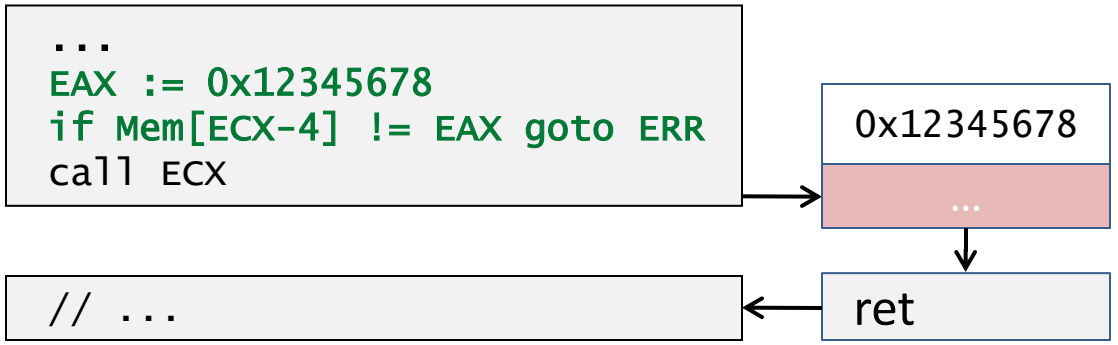
A CFI guard

(a simple variant)

- A CFI guard matches IDs at source and target.
  - IDs are constants embedded in machine code.
  - IDs are not secret, but must be unique.

```
pf(A, pm[0], pm[1]);
// ...
```

C source code



Machine code with 0x12345678 as CFI guard ID

Proving that CFI works

- Some of the recent systems come with (and were guided by) proofs of correctness.
- The basic steps may be:
  1. Define a machine language and its semantics.
  2. Define when a program has appropriate instrumentation, for a given control-flow graph.
  3. Prove that all executions of programs with appropriate instrumentation follow the prescribed control-flow graphs.

1. A small model of a machine

- Instructions: nop, $addi$, $movi$, bgt, jd, jmp, ld, st.
- States: each state is a tuple that includes
  - code memory M_c
  - data memory M_d
  - registers R
  - program counter pc
- Steps: transition relations define the possible state changes of the machine.

1. A small model of a machine

If $Dc(M_c(pc))=$	then $(M_c \mid M_d, R, pc) \rightarrow_n$
$nop\ w$	$(M_c \mid M_d, R, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$add\ r_d, r_s, r_t$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+R(r_t)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$addi\ r_d, r_s, w$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$movi\ r_d, w$	$(M_c \mid M_d, R\{r_d \mapsto w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$bgt\ r_s, r_t, w$	$(M_c \mid M_d, R, w)$, when $R(r_s) > R(r_t) \wedge w \in \mathrm{dom}(M_c)$ $(M_c \mid M_d, R, pc+1)$, when $R(r_s) \leq R(r_t) \wedge pc+1 \in \mathrm{dom}(M_c)$
$jd\ w$	$(M_c \mid M_d, R, w)$, when $w \in \mathrm{dom}(M_c)$
$jmp\ r_s$	$(M_c \mid M_d, R, R(r_s))$, when $R(r_s) \in \mathrm{dom}(M_c)$
$ld\ r_d, r_s(w)$	$(M_c \mid M_d, R\{r_d \mapsto M(R(r_s)+w)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$st\ r_d(w), r_s$	$(M_c \mid M_d\{R(r_d)+w \mapsto R(r_s)\}, R, pc+1)$, when $R(r_d)+w \in \mathrm{dom}(M_d) \wedge pc+1 \in \mathrm{dom}(M_c)$

1. A small model of a machine

If $Dc(M_c(pc))=$	then $(M_c \mid M_d, R, pc) \rightarrow_n$
$nop\ w$	$(M_c \mid M_d, R, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$add\ r_d, r_s, r_t$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+R(r_t)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$addi\ r_d, r_s, w$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$movi\ r_d, w$	$(M_c \mid M_d, R\{r_d \mapsto w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$bgt\ r_s, r_t, w$	$(M_c \mid M_d, R, w)$, when $R(r_s) > R(r_t) \wedge w \in \mathrm{dom}(M_c)$ $(M_c \mid M_d, R, pc+1)$, when $R(r_s) \leq R(r_t) \wedge pc+1 \in \mathrm{dom}(M_c)$
$jd\ w$	$(M_c \mid M_d, R, w)$, when $w \in \mathrm{dom}(M_c)$
$jmp\ r_s$	$(M_c \mid M_d, R, R(r_s))$, when $R(r_s) \in \mathrm{dom}(M_c)$
$ld\ r_d, r_s(w)$	$(M_c \mid M_d, R\{r_d \mapsto M(R(r_s)+w)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$st\ r_d(w), r_s$	$(M_c \mid M_d\{R(r_d)+w \mapsto R(r_s)\}, R, pc+1)$, when $R(r_d)+w \in \mathrm{dom}(M_d) \wedge pc+1 \in \mathrm{dom}(M_c)$

Dc : instruction decoding function

1. A small model of a machine

If $Dc(M_c(pc))=$	then $(M_c \mid M_d, R, pc) \rightarrow_n$
$nop\ w$	$(M_c \mid M_d, R, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$add\ r_d, r_s, r_t$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+R(r_t)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$addi\ r_d, r_s, w$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$movi\ r_d, w$	$(M_c \mid M_d, R\{r_d \mapsto w\}, pc+1)$,
$bgt\ r_s, r_t, w$	
$jd\ w$	$(M_c \mid M_d, R, w)$, when $w \in \mathrm{dom}(M_c)$
$jmp\ r_s$	$(M_c \mid M_d, R, R(r_s))$, when $R(r_s) \in \mathrm{dom}(M_c)$
$ld\ r_d, r_s(w)$	$(M_c \mid M_d, R\{r_d \mapsto M(R(r_s)+w)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$st\ r_d(w), r_s$	$(M_c \mid M_d\{R(r_d)+w \mapsto R(r_s)\}, R, pc+1)$, when $R(r_d)+w \in \mathrm{dom}(M_d) \wedge pc+1 \in \mathrm{dom}(M_c)$

$$\frac{Dc(M_c(pc)) = jmp\ r_s \quad R(r_s) \in \mathrm{dom}(M_c)}{(M_c \mid M_d, R, pc) \rightarrow_n (M_c \mid M_d, R, R(r_s))}$$

1. A small model of a machine

If $Dc(M_c(pc))=$	then $(M_c \mid M_d, R, pc) \rightarrow_n$
$nop\ w$	$(M_c \mid M_d, R, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$add\ r_d, r_s, r_t$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+R(r_t)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$addi\ r_d, r_s, w$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$movi\ r_d, w$	$(M_c \mid M_d, R\{r_d \mapsto w\}, pc+1)$,
$bgt\ r_s, r_t, w$	
$jd\ w$	$(M_c \mid M_d, R, w)$, when $w \in \mathrm{dom}(M_c)$
$jmp\ r_s$	$(M_c \mid M_d, R, R(r_s))$, when $R(r_s) \in \mathrm{dom}(M_c)$
$ld\ r_d, r_s(w)$	$(M_c \mid M_d, R\{r_d \mapsto M(R(r_s)+w)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$st\ r_d(w), r_s$	$(M_c \mid M_d\{R(r_d)+w \mapsto R(r_s)\}, R, pc+1)$, when $R(r_d)+w \in \mathrm{dom}(M_d) \wedge pc+1 \in \mathrm{dom}(M_c)$

$$\frac{Dc(M_c(pc)) = jmp\ r_s \quad R(r_s) \in \mathrm{dom}(M_c)}{(M_c \mid M_d, R, pc) \rightarrow_n (M_c \mid M_d, R, R(r_s))}$$

+ M_d could change at any time (because of attacker actions).

2. Example condition on instrumentation

Computed jumps occur only in context of a specific instruction sequence:

$$
\begin{aligned}
&addi\ r_0, r_s, 0\\
&ld\ r_1, r_0(0)\\
&movi\ r_2, IMM\\
&bgt\ r_1, r_2, HALT\\
&bgt\ r_2, r_1, HALT\\
&jmp\ r_0
\end{aligned}
$$

2. Example condition on instrumentation

Computed jumps occur only in context of a specific instruction sequence:

$$
\begin{aligned}
&addi\ r_0, r_s, 0\\
&ld\ r_1, r_0(0)\\
&movi\ r_2, IMM\\
&bgt\ r_1, r_2, HALT\\
&bgt\ r_2, r_1, HALT\\
&jmp\ r_0
\end{aligned}
$$

HALT is the address of a halt instruction.

IMM is a constant that encodes the allowed label at the jump target.

3. A result

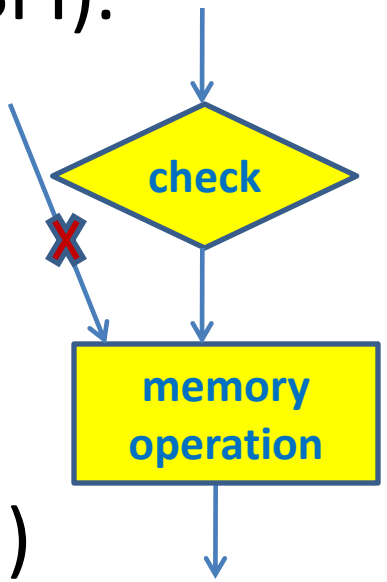
Let S_0 be a state with pc = 0 and code memory M_c that satisfies the instrumentation condition for a given CFG.

Suppose $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$
where each $\rightarrow$ transition is either a normal $\rightarrow_n$ step or an attacker step that changes only data memory.

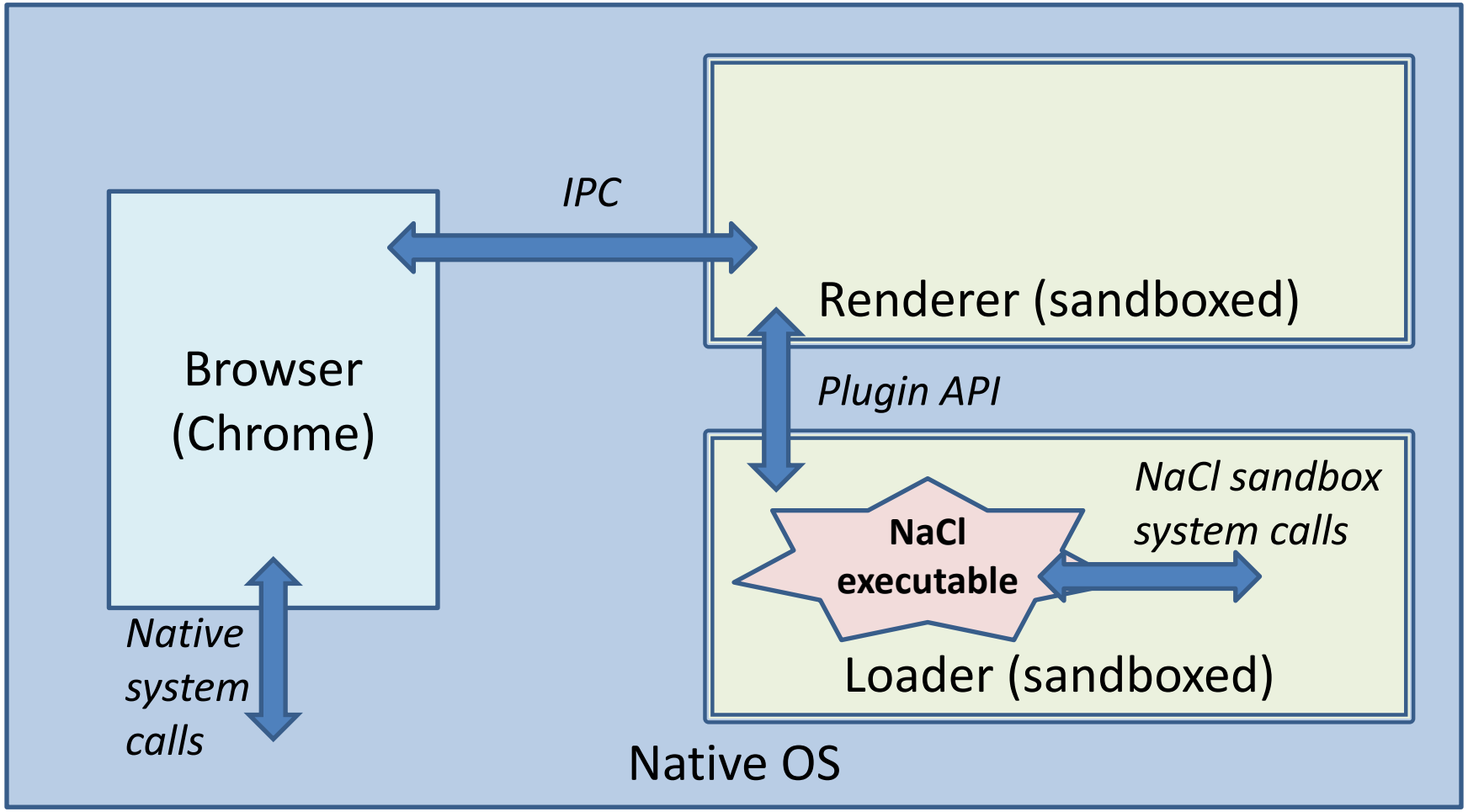
For each i, if $S_i \rightarrow_n S_{i+1}$ then pc at S_{i+1} is one of the allowed successors of pc at S_i according to the CFG.

Software-based fault isolation

- CFI does not assume memory protection.
- But it enables memory protection, i.e., “software-based fault isolation” (SFI).
- Again, there are several possible implementations of SFI.
  - E.g., by code rewriting, with guards on memory operations.
- Recent systems (XFI, BGI, LXFI, NaCl, …) explore several variants and extensions.



A recent system: Native Client (NaCl) [Yee et al.]



A recent SFI tool: RockSalt

[Morrisett et al.]

- RockSalt is an SFI checker
  - for the NaCl sandbox policy,
  - ~80 lines of Coq code, manually translated into C.
- A formal argument shows that, if RockSalt accepts a string of bytes *B*, then *B*'s execution on x86 will respect the sandbox policy.
  - The argument is based on a sophisticated Coq model of x86 integer instructions.
  - More work remains, in several directions: models, proofs, policies.

Security in programming languages

Security in programming languages

- Languages have long been related to security.
- Modern languages should contribute to security:
  - Constructs for protection (e.g., objects).
  - Techniques for static analysis,
    in particular for ensuring safety by typechecking.
  - A tractable theory, with sophisticated methods.
- Several security techniques rely on language ideas, with static and dynamic checks.

See Morris's "Protection in Programming Languages".

A class with a secret field

```
class C {
    // the field
    private int x;
    // a constructor
    public C(int v) { x = v; }
}

// two instances of C
C c1 = new C(17);
C c2 = new C(28);
```

- A possible conjecture: *Any two instances of this class are observationally equivalent (that is, they cannot be distinguished within the language).*
- More realistic examples use constructs similarly.
- Objects are unforgeable. E.g., integers cannot be cast into objects.

Mediated access [example from A. Kennedy]

```
class Widget {// No checking of argument
  virtual void Operation(string s) {…};
}
class SecureWidget : Widget {
  // Validate argument and pass on
  // Could also authenticate the caller
  override void Operation(string s) {
    Validate(s);
    base.Operation(s);
  }
}
…
SecureWidget sw = new SecureWidget();
sw.Operation(“Nice string”);
// Can’t avoid validation of argument
```

Caveats

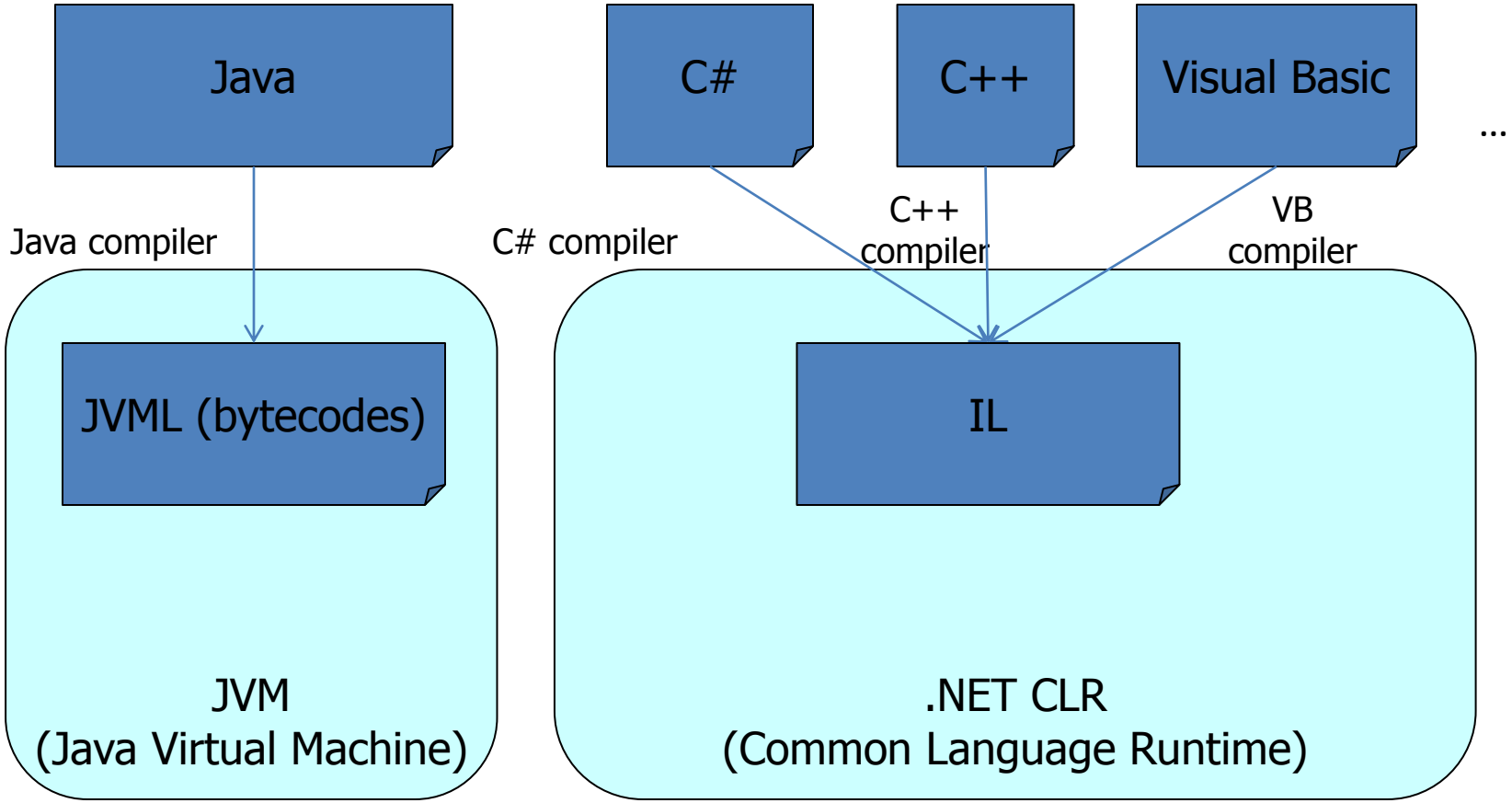
Mismatch in characteristics:

- Security requires simplicity and minimality.
- Common programming languages are complex.

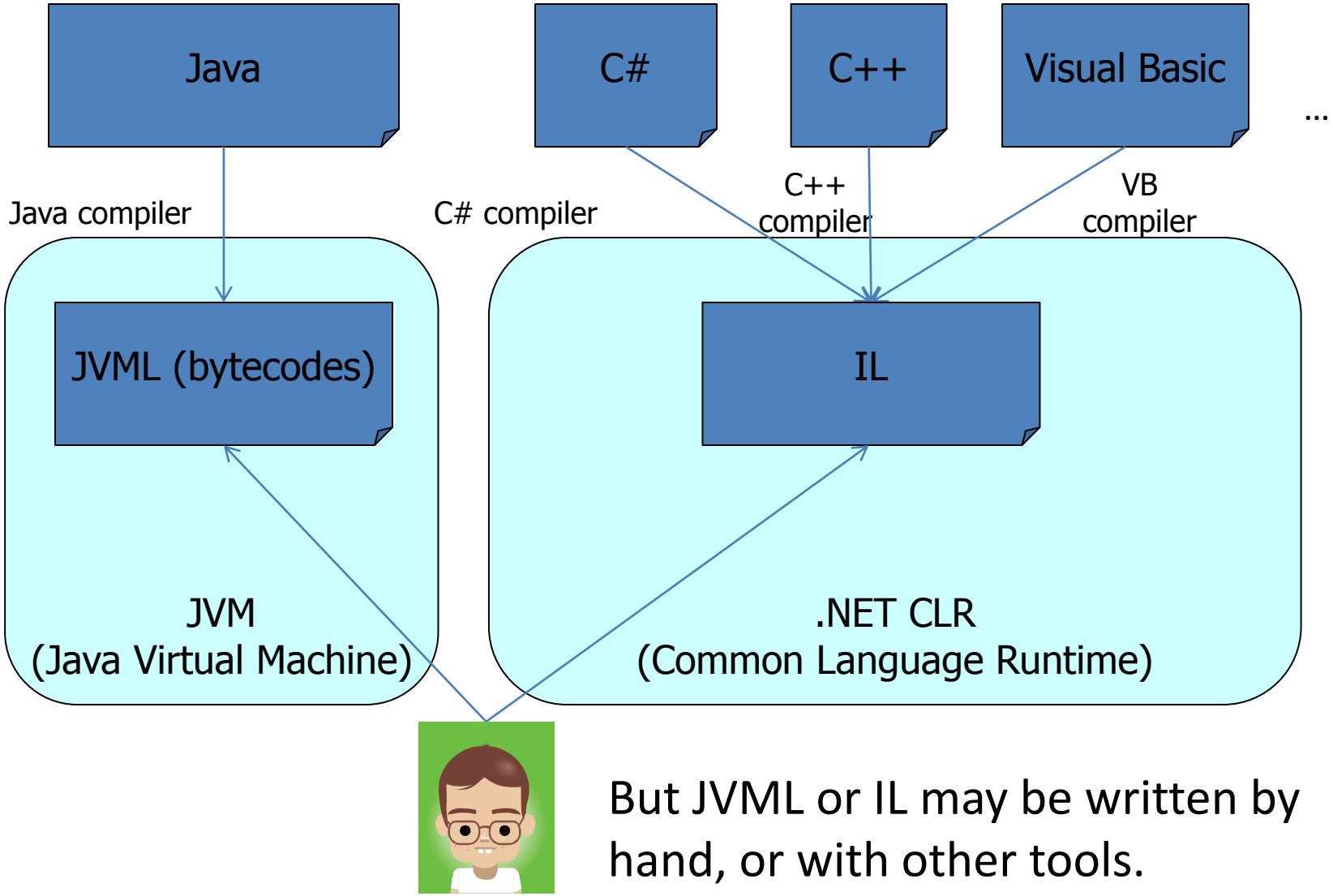
Mismatch in scope:

- Language descriptions rarely specify security. Implementations may or may not be secure.
- Security is a property of systems (not languages). Systems typically include much security machinery beyond what is given in language definitions.

“Secure” programming platforms

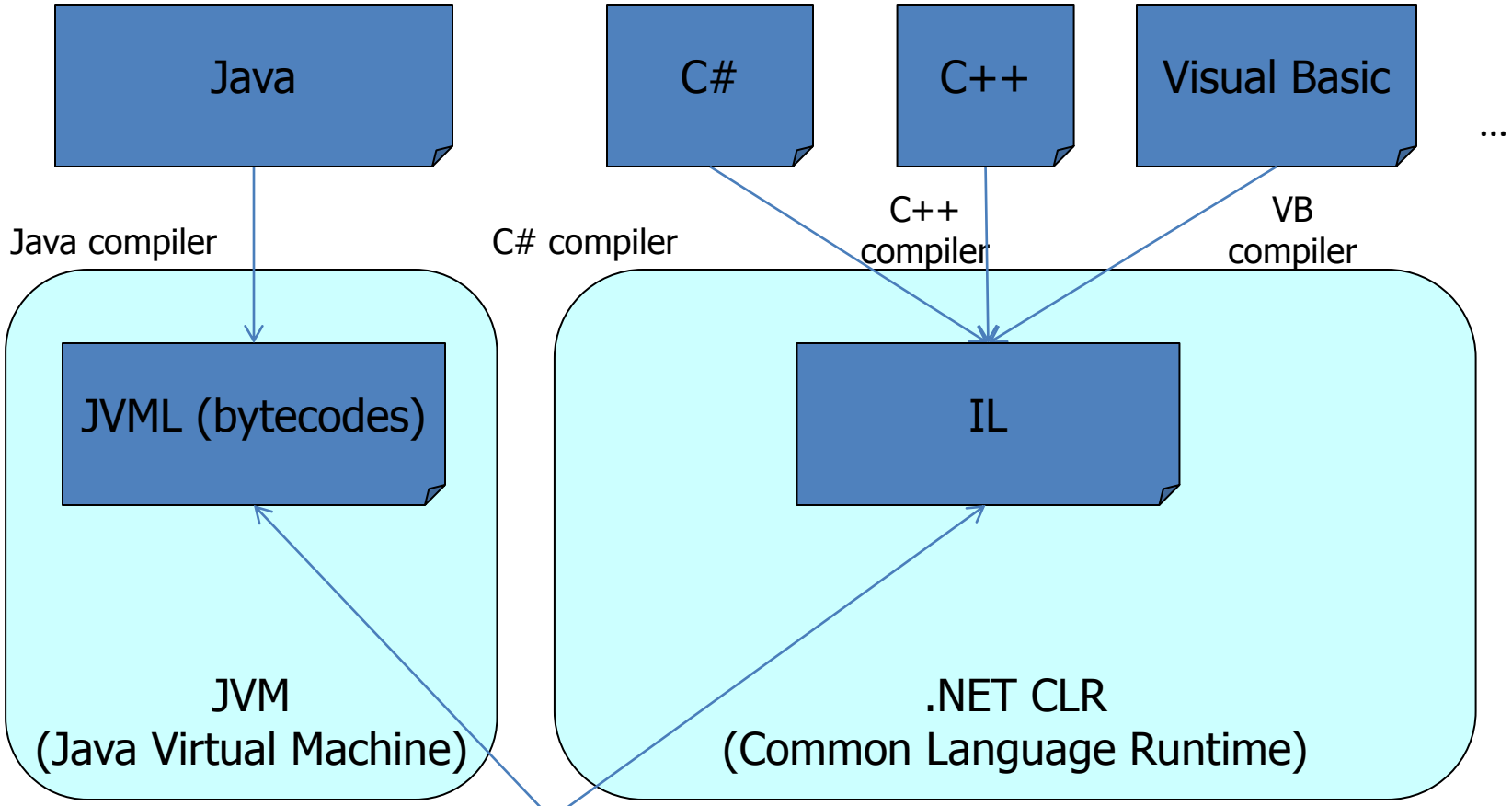


“Secure” programming platforms



But JVML or IL may be written by hand, or with other tools.

“Secure” programming platforms



But JVML or IL may be written by hand, or with other tools.

Mediated access

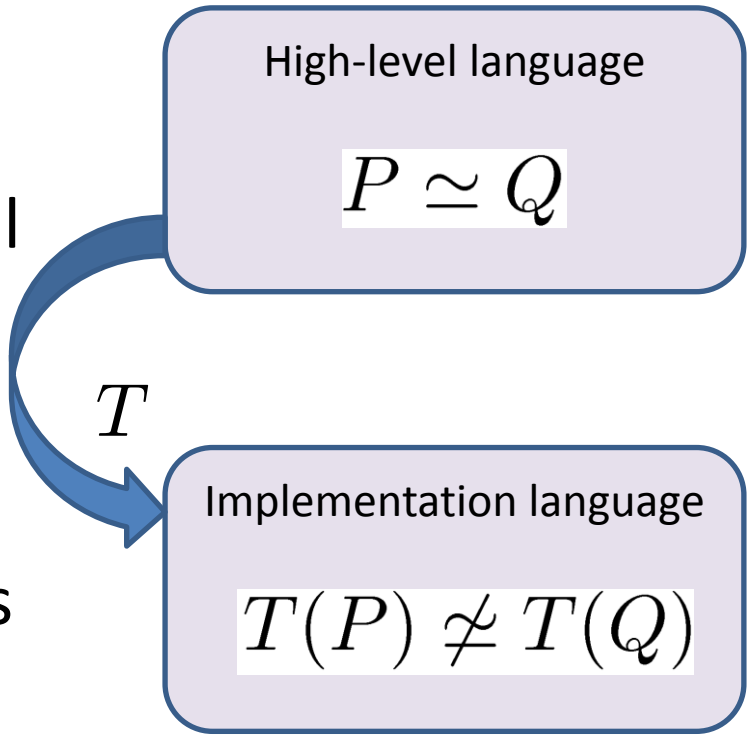
```
class Widget {// No checking of argument
  virtual void Operation(string s) {…};
}
class SecureWidget : Widget {
  // Validate argument and pass on
  // Could also authenticate the caller
  override void Operation(string s) {
    Validate(s);
    base.Operation(s);
```

```
// In IL (pre-2.0), make a direct call
// on the superclass:
ldloc sw
ldstr “Invalid string”
call void Widget::Operation(string)
```

Other examples

There are many more examples, for Java, C#, and other languages.

- In each case, some observational equivalence that holds in the source language does not hold in implementations.
- We may say that the translations are not ***fully abstract***.
- Typechecking helps, but it does not suffice.



Alternatives

- One may ignore the security of translations
  - when low-level code is signed by a trusted party,
  - if one analyzes low-level code.

  These alternatives are not always satisfactory.
- In other cases, translations should preserve at least some security properties; for example:
  - limited versions of full abstraction (e.g., for certain programming idioms),
  - the secrecy of pieces of data labelled as secret,
  - fundamental guarantees about control flow.

Closing comments

Some themes

Some themes

- Inventive attackers, with deep, detailed understanding of their targets.

Some themes

- Inventive attackers, with deep, detailed understanding of their targets.
- The malleability of software:
  - enables sophisticated architectures and methods for protection,
  - benefits from looseness in systems constraints ("*our goal is not to preserve semantics, but to improve it*"),
  - costs in compatibility and run-time efficiency.

Abstractions and security

Abstractions are common in computing, e.g.:

- function calls,
- objects with private components,
- secure channels.

Clever implementation techniques abound too:

- stacks,
- static and dynamic access checks,
- cryptography.

Abstractions and security

Abstractions are common in computing, e.g.:

- function calls,
- objects with private components,
- secure channels.

Clever implementation techniques abound too:

- stacks,
- static and dynamic access checks,
- cryptography.

Implementations often need to work in interaction with (malicious?) systems that do not use the abstractions.

Abstractions and security

Abstractions are common in computing, e.g.:

- function calls,
- objects with private components,
- secure channels.

Clever implementation techniques abound too:

- stacks,
- static and dynamic access checks,
- cryptography.

Implementations often need to work in interaction with (malicious?) systems that do not use the abstractions.

This holds even for low-level code, and ideas originally developed in high-level languages are useful there too.