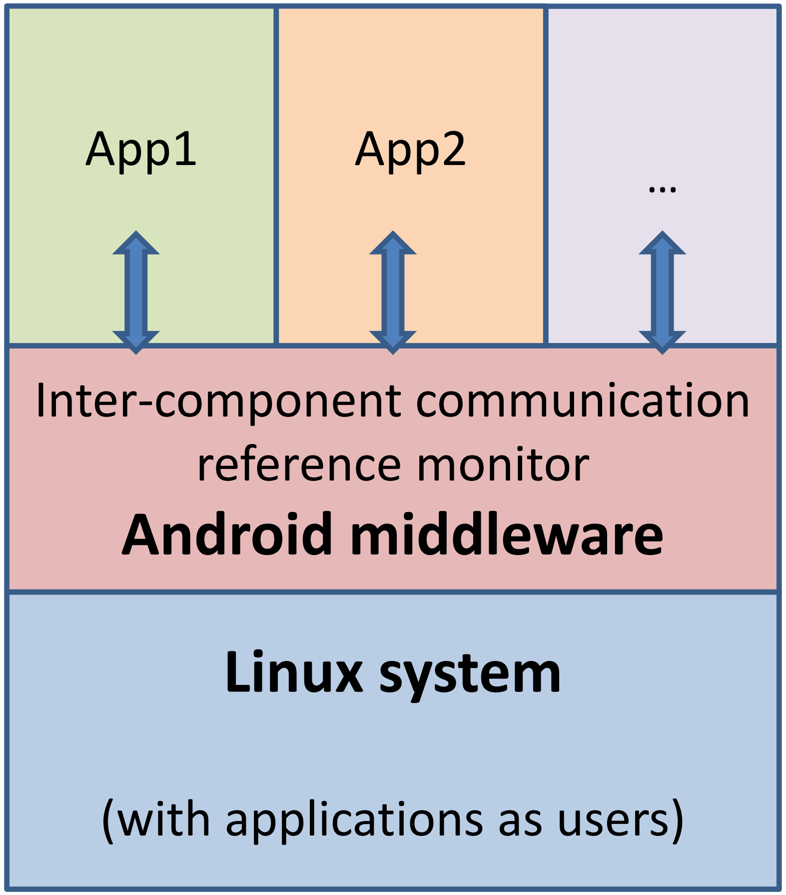
Software security

Chaire Informatique et sciences numériques
Collège de France, cours du 6 avril 2011

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



App1

App2

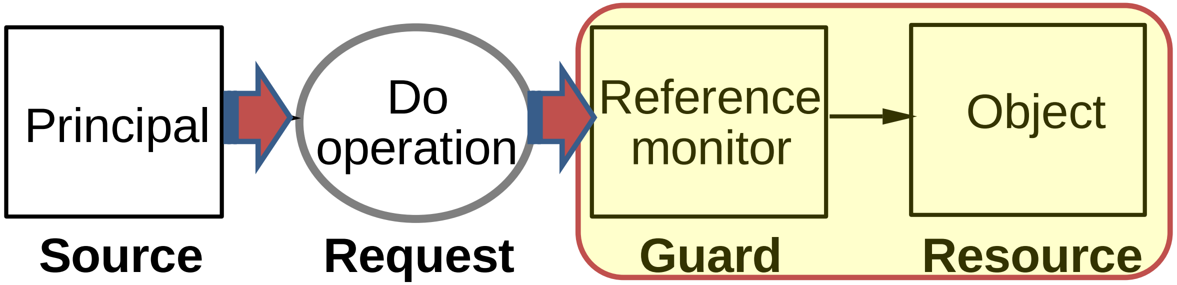
…

Inter-component communication reference monitor

Android middleware

Linux system

(with applications as users)



Principal

Source

Do operation

Request

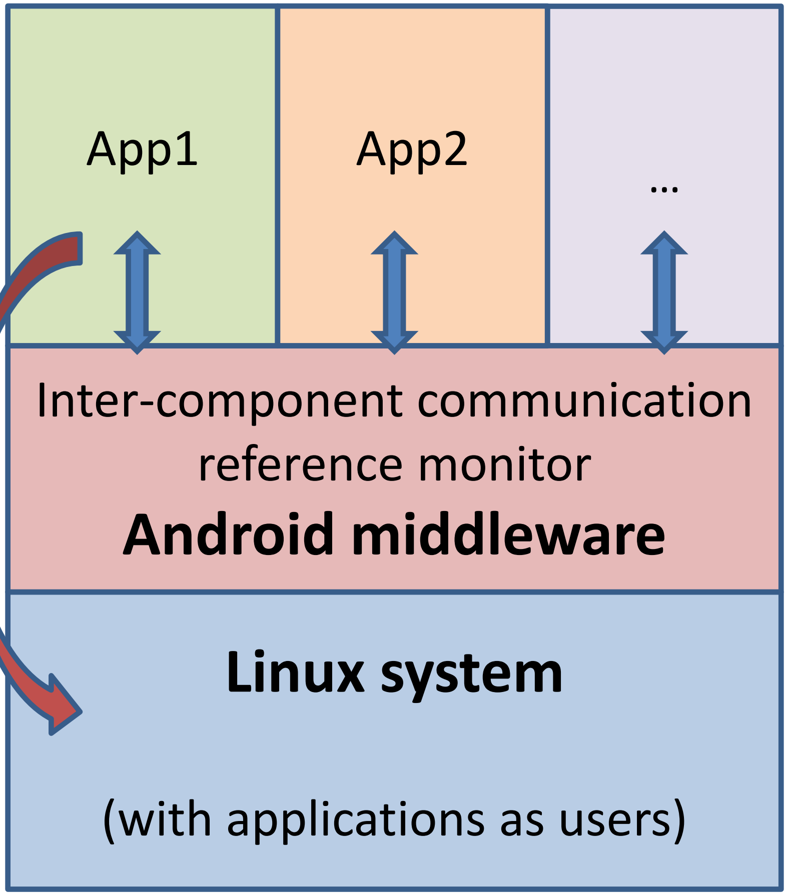
Reference monitor

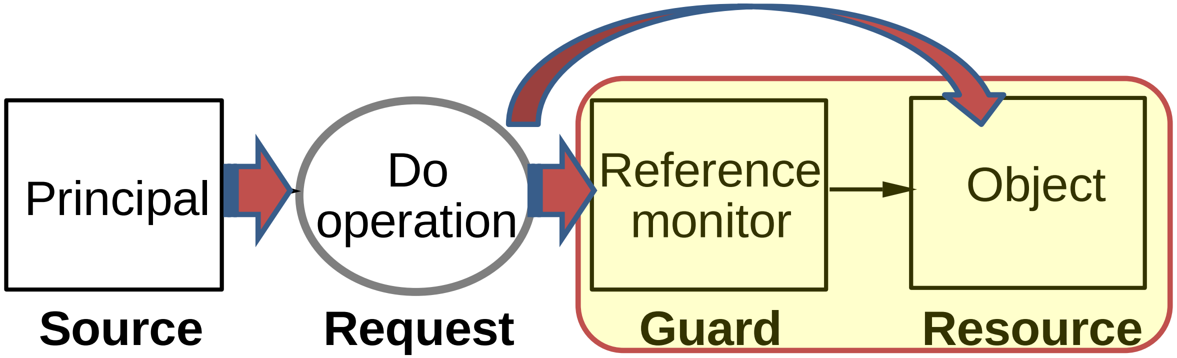
Guard

Object

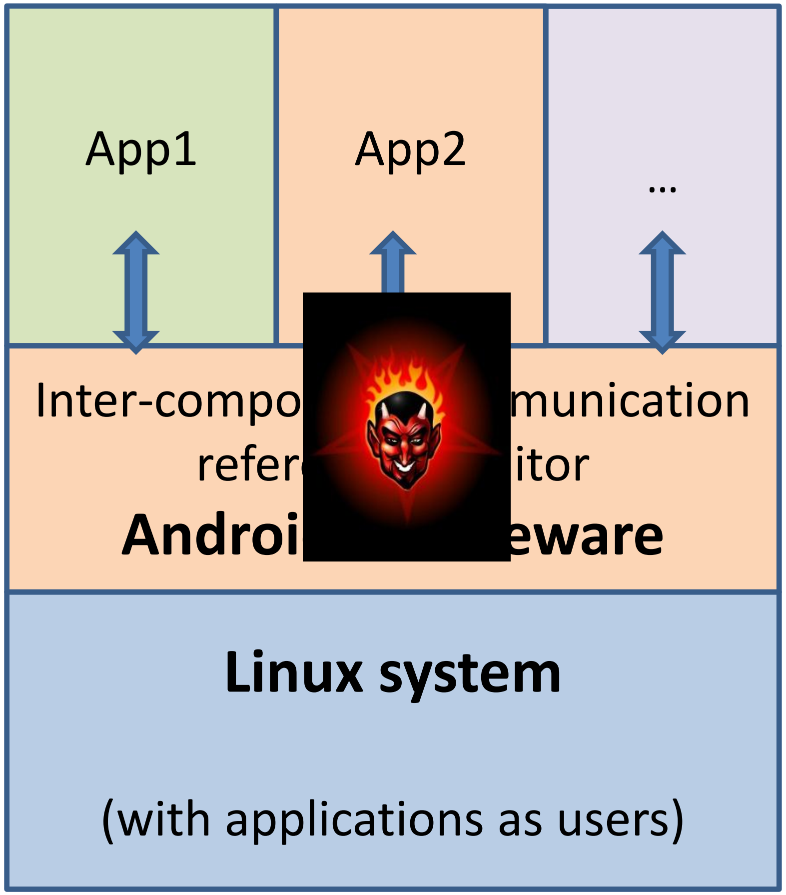
Resource

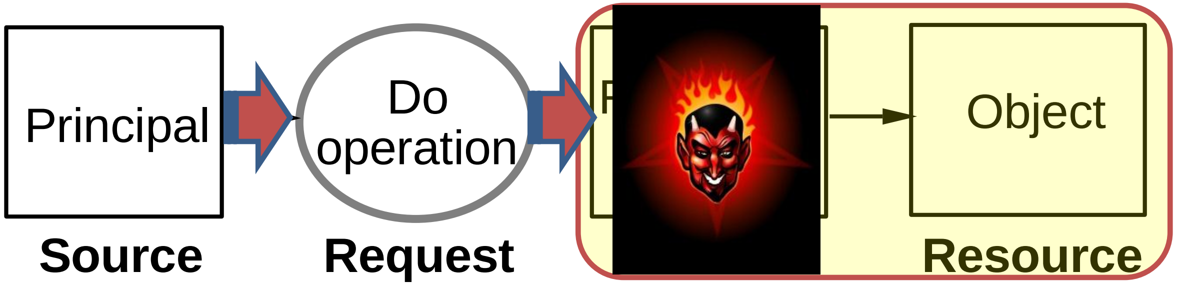
Pictures such as these ones make sense only if a component cannot circumvent or hijack other components.





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

```
// Les lignes qui commencent par des barres sont des commentaires.
// Nous définissons une fonction f à deux arguments :
//     un nombre entier x et un caractère y.
// La fonction donne un résultat entier.
int f(int x, char y)  {
    // La fonction a une variable locale :
    // un tableau t de taille 16 qui contient des caractères.
    char t[16] ;
    // Nous pouvons donner des valeurs initiales aux entrées du tableau.
    // initialize est une fonction dont les détails ne nous intéresseront pas.
    initialize(t) ;
    // Puis nous donnons la valeur y à l’entrée x de t.
   t[x] = y ;
    // Le résultat 0 indique juste que la fonction a bien tourné.
   return 0 ;
   }
```

An example

```
// Les lignes qui commencent par des barres sont des commentaires.
// Nous définissons une fonction f à deux arguments :
//     un nombre entier x et un caractère y.
// La fonction donne un résultat entier.
int f(int x, char y)  {
   // La fonction a une variable locale :
   // un tableau t de taille 16 qui contient des caractères.
   char t[16] ;
   // Nous pouvons donner des valeurs initiales aux entrées du tableau.
   // initialize est une fonction dont les détails ne nous intéresseront pas.
   initialize(t) ;
   // Puis nous donnons la valeur y à l'entrée x de t.
  t[x] = y ;
   // Le résultat 0 indique juste que la fonction a bien tourné.
  return 0 ;
  }
```

So what?

- Threat model: The attacker chooses inputs.

⇒ 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?

Two examples of low-level attacks

[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.***

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 run-time mitigation techniques.**
  - Defense in depth (e.g., use sacrificial machines).
  - Software updates.

Low-level attacks and defenses

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

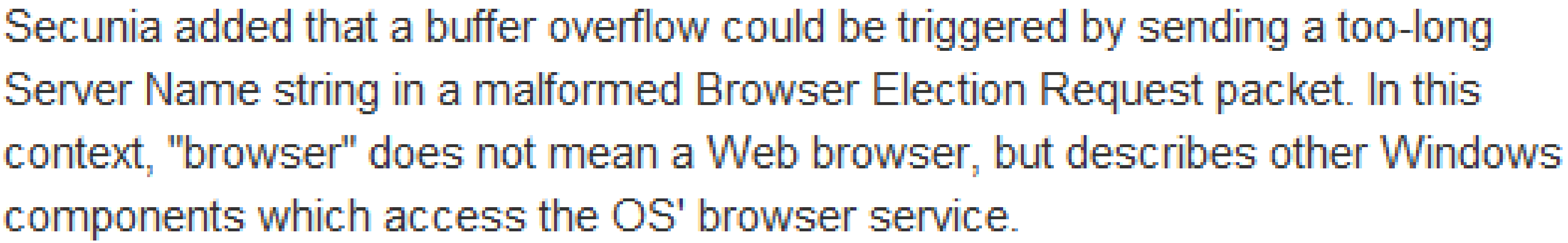
New Windows zero-day surfaces as researcher releases attack code

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:

		local variable t	return address	
First	**…**	**(nothing yet)**	**f’s caller address**	**…**

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:

		local variable t	return address	
First	**…**	**(nothing yet)**	**f’s caller address**	**…**
Later	**…**	**arg contents**	**f’s caller address**	**…**

A buffer overflow

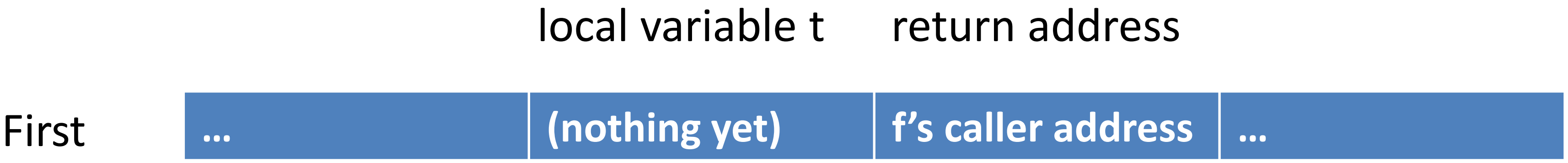
```
define function f(arg) =
    let t be a local variable of size n;
    copy contents of arg into t;
            …
```

- 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;
      …
```

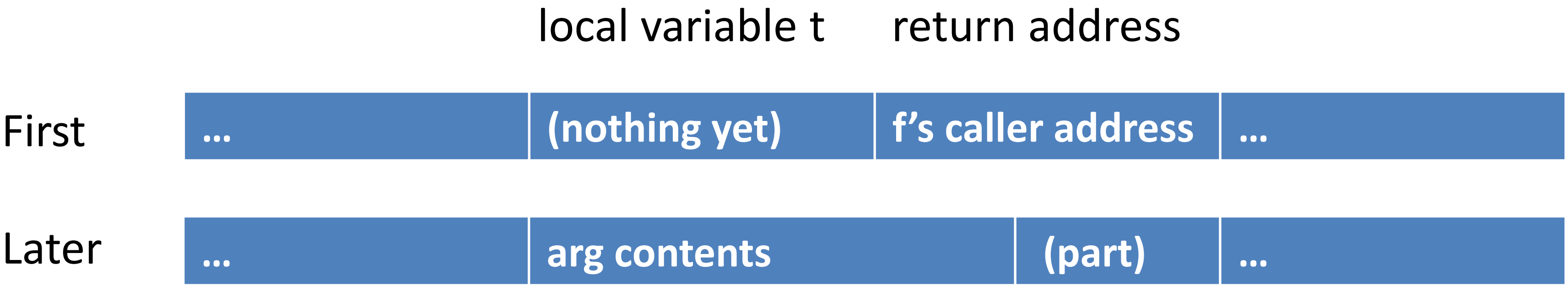
- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could have:



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:



local variable t return address

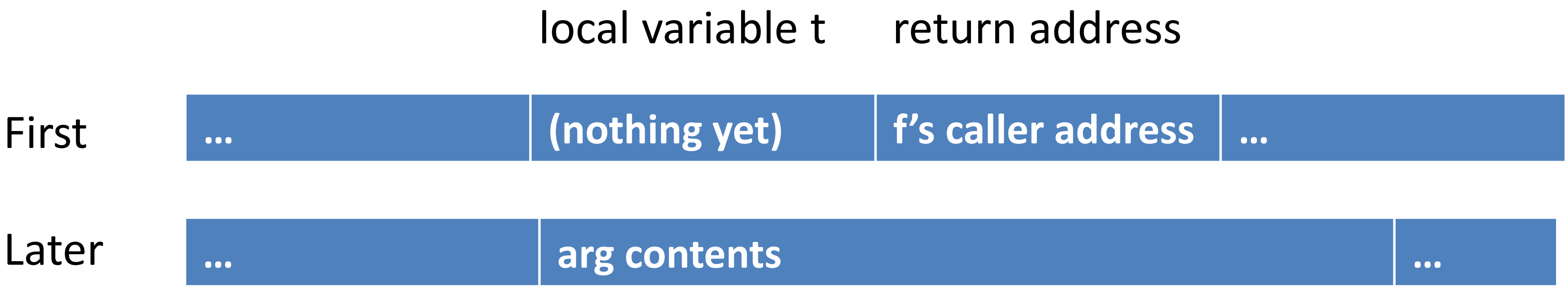
First … **(nothing yet)** **f's caller address** **…**

Later … **arg contents** **(part)** **…**

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:



local variable t
return address

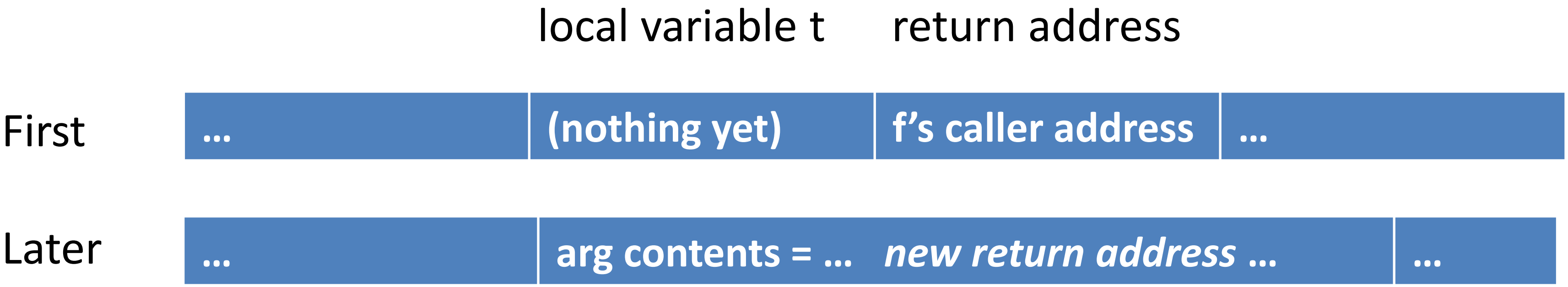
First
…
(nothing yet)
f's caller address
…

Later
…
arg contents
…

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:



local variable t return address

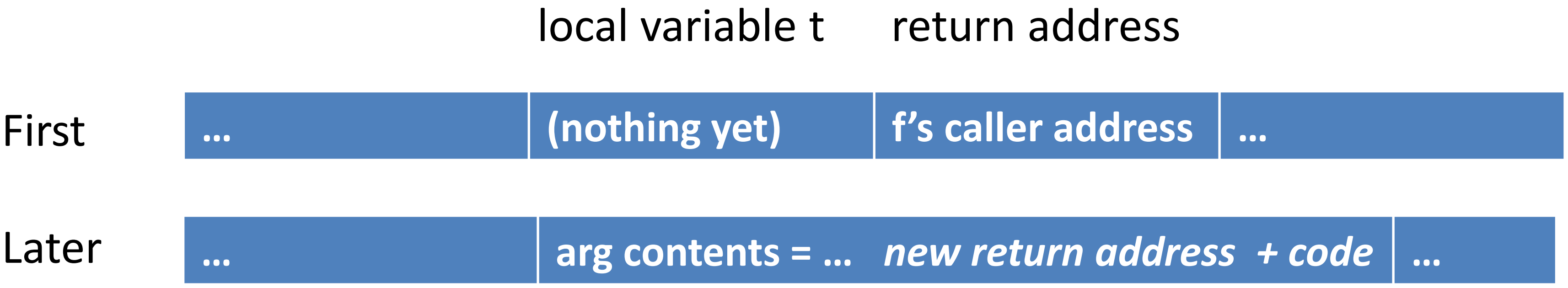
First … (nothing yet) f's caller address …

Later … arg contents = … *new return address …* …

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:

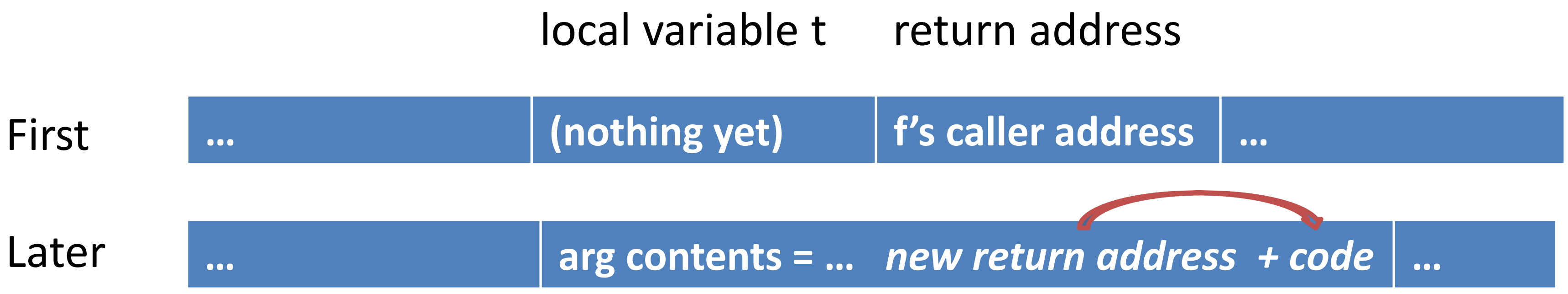


		local variable t	return address	
First	**…**	**(nothing yet)**	**f’s caller address**	**…**
Later	**…**	**arg contents = …** ***new return address + code***		**…**

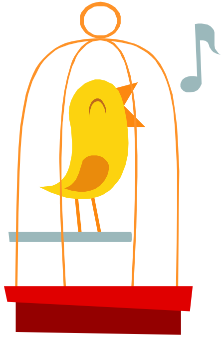
A buffer overflow

```
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;
        …
```

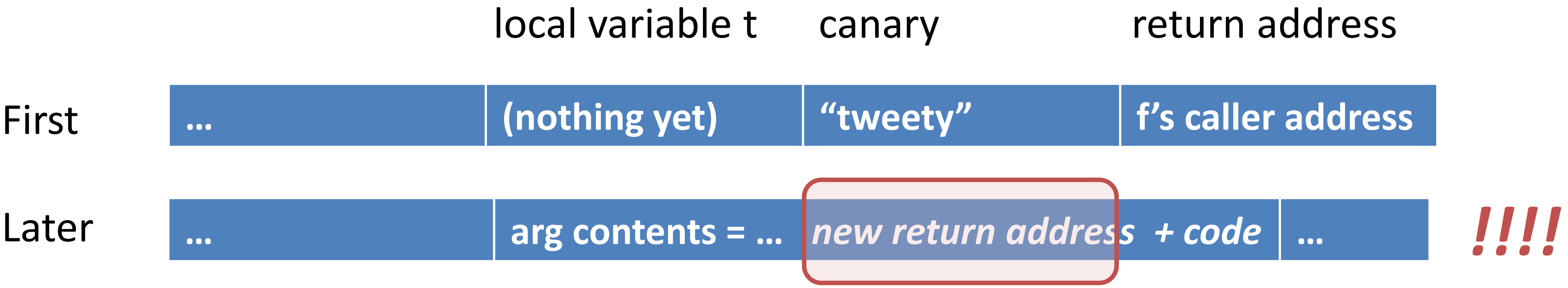
- A known quantity (fixed or random) can be inserted between the local variable and the return address so that any corruption can be detected.

		local variable t	canary	return address
First	**…**	**(nothing yet)**	**“tweety”**	**f’s caller address**

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.



		local variable t	canary	return address
First	…	(nothing yet)	"tweety"	f's caller address

Later: … | arg contents = … | *new return address + code* | …

!!!!

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.

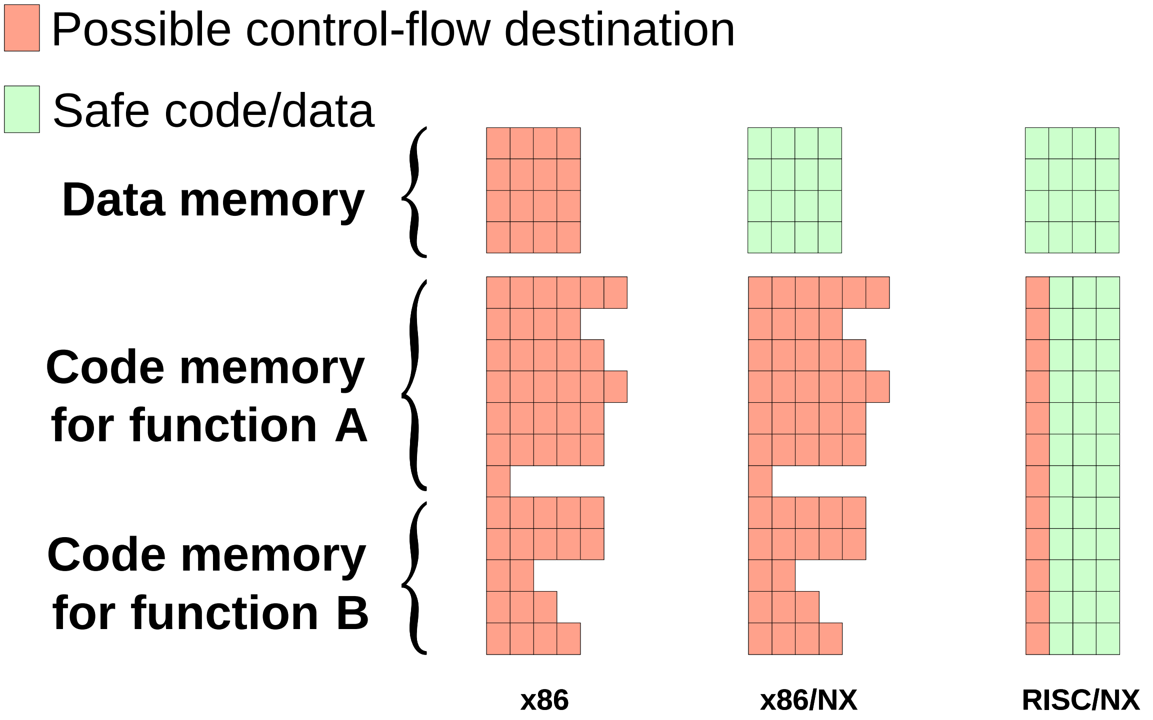
⇒ ***So a defense is to require that data that is writable cannot be executed.****

- 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:



🟥 Possible control-flow destination

🟩 Safe code/data

Data memory

Code memory for function A

Code memory for function B

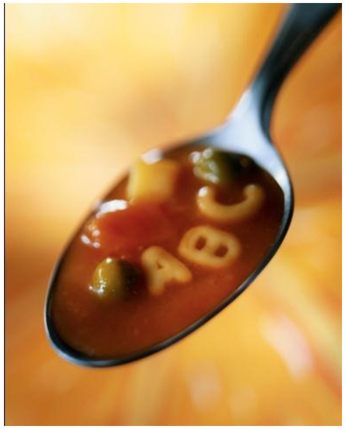
x86

x86/NX

RISC/NX

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

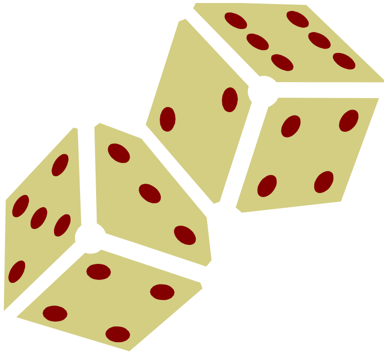
```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

Layout randomization

Attacks often depend on addresses.

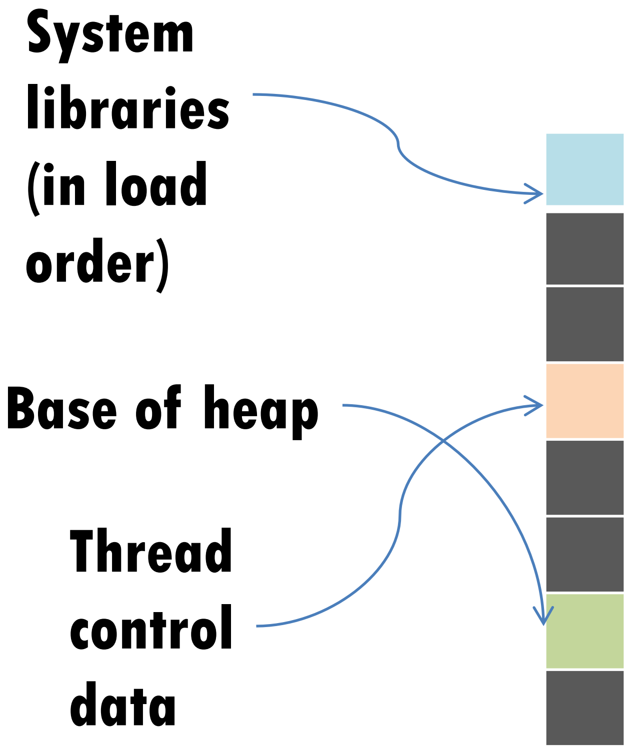
$\Rightarrow$ ***Let us randomize the addresses!***

- Considered for data at least since the rise of large virtual address spaces (e.g., [Druschel & Peterson, 1992] on fbufs).
- Present in Linux (PaX) and Windows (ASLR).



Implementations

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



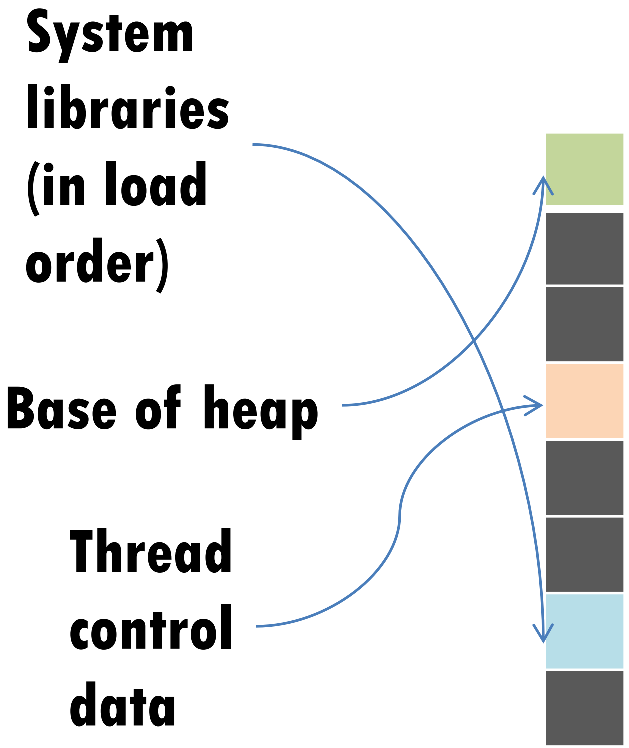
System libraries (in load order)

Base of heap

Thread control data

Implementations

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



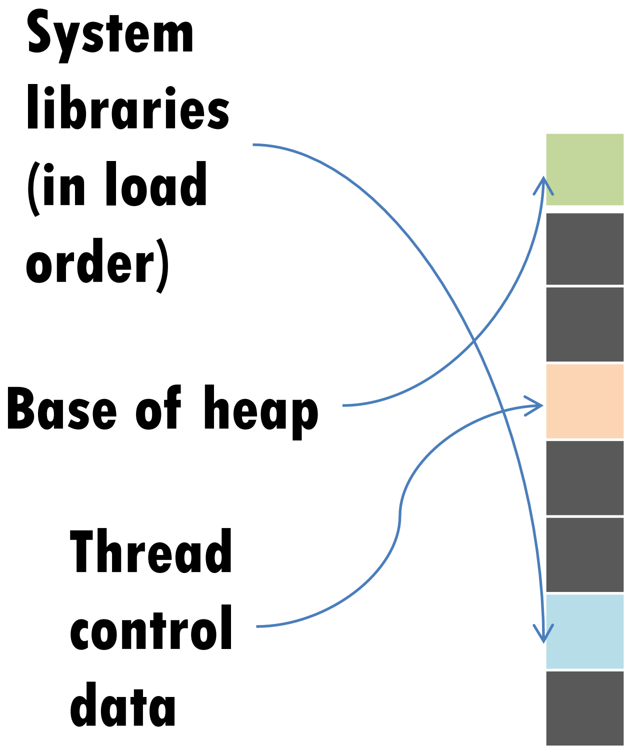
System libraries (in load order)

Base of heap

Thread control data

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.



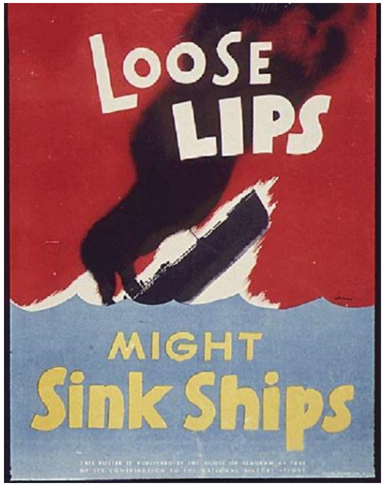
System libraries (in load order)

Base of heap

Thread control data

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.

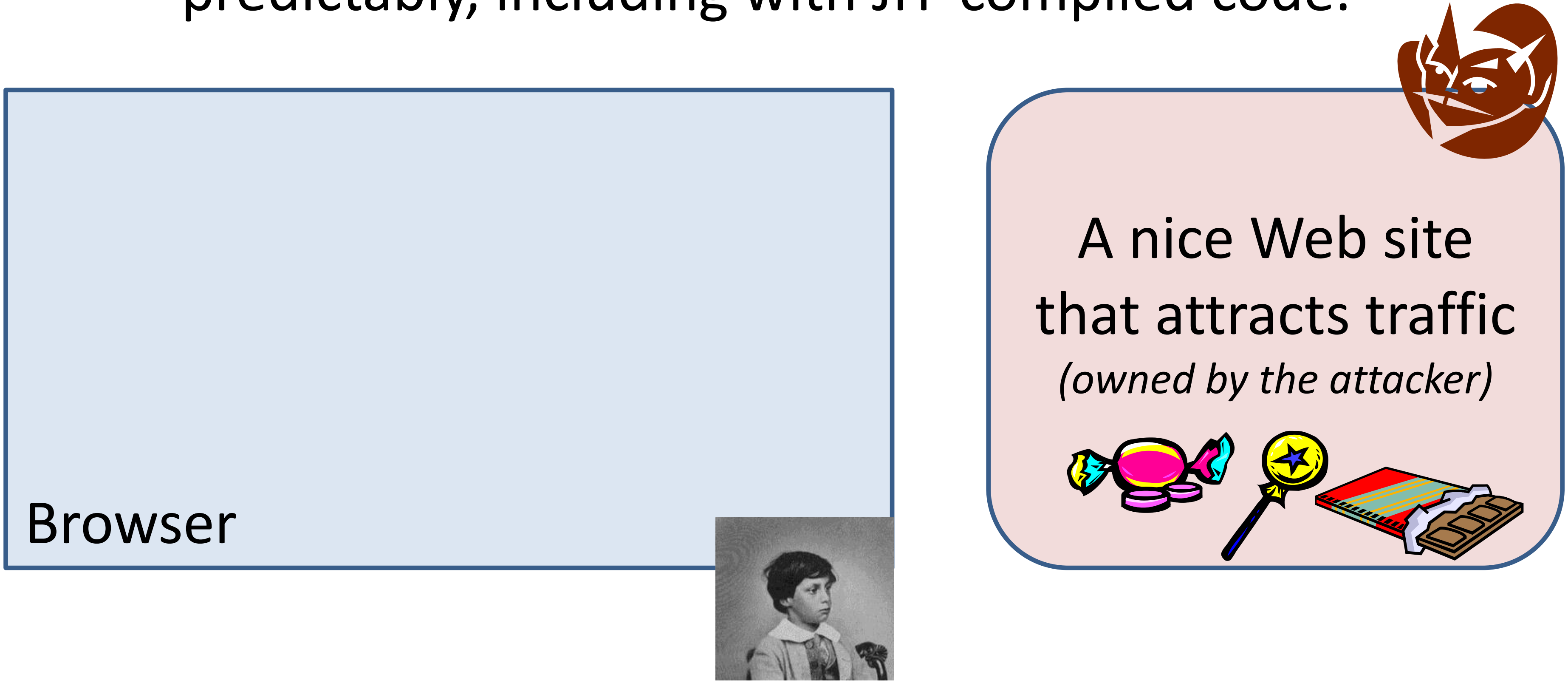
A nice Web site that attracts traffic

(owned by the attacker)

Browser

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.



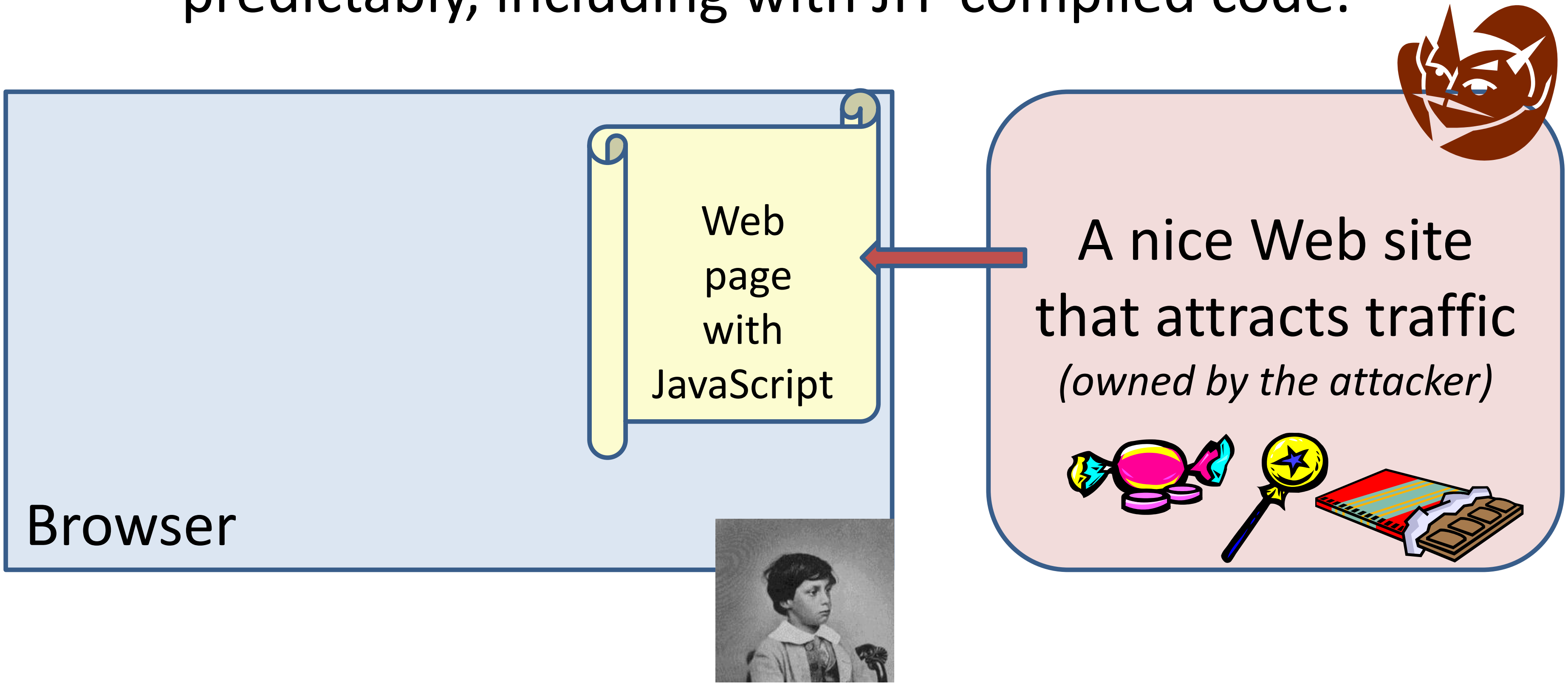
Browser

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.



Web page with JavaScript

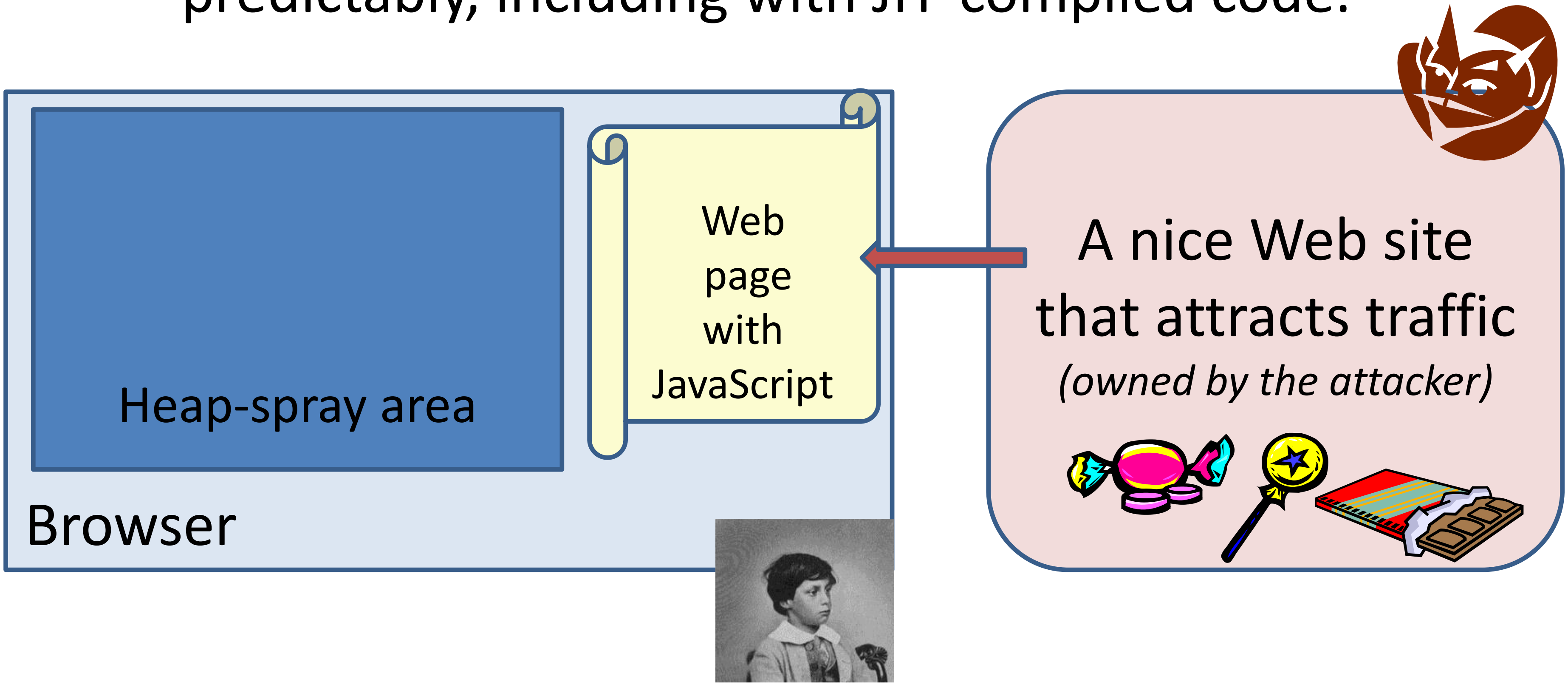
A nice Web site that attracts traffic

(owned by the attacker)

Browser

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-spray area

Web page with JavaScript

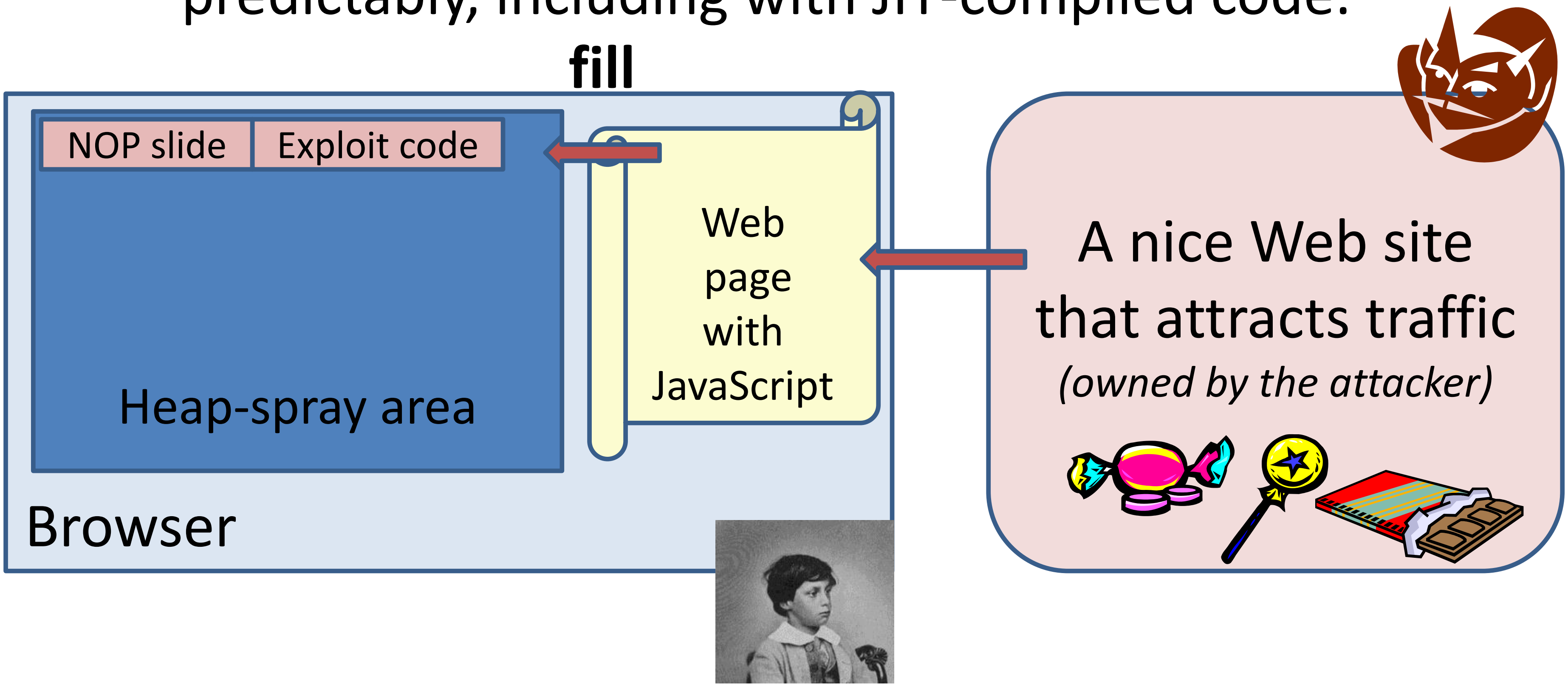
Browser

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.



fill

NOP slide

Exploit code

Heap-spray area

Web page with JavaScript

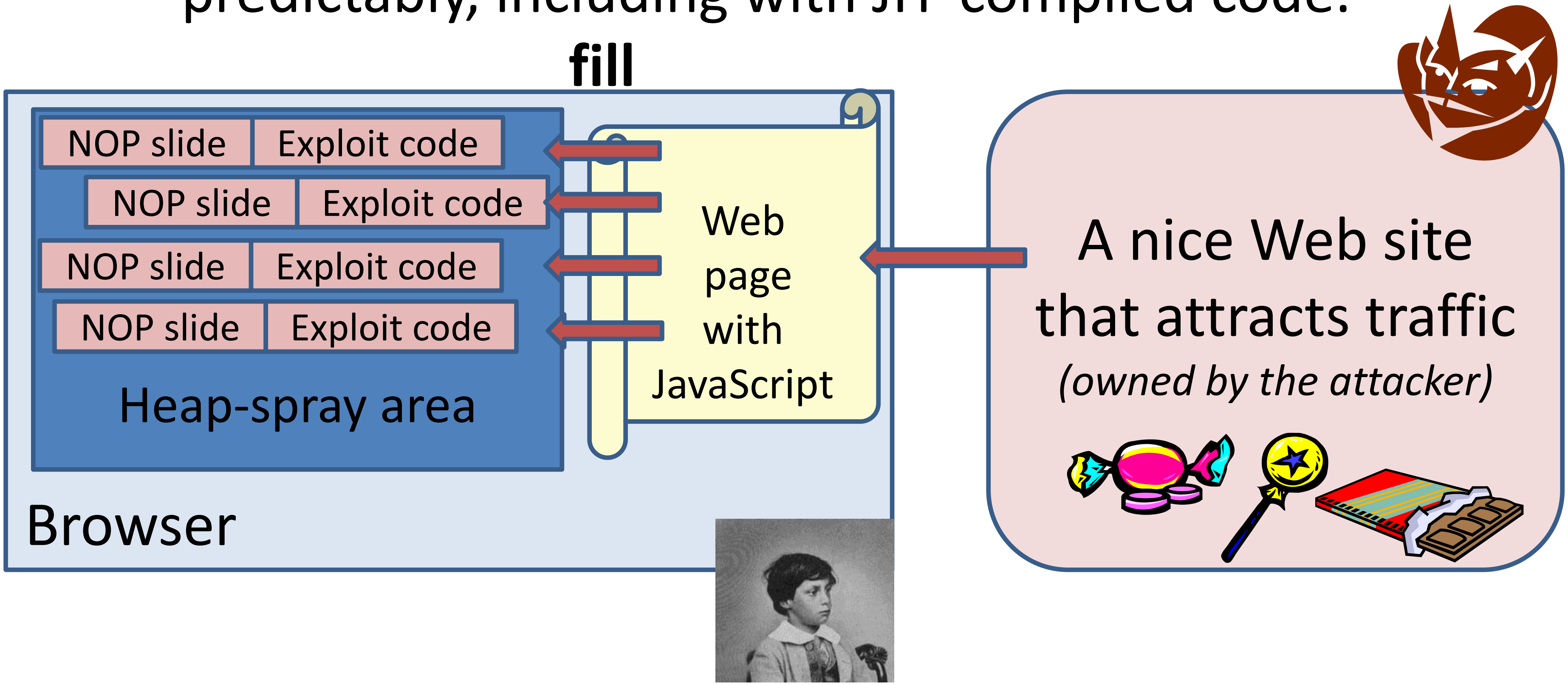
Browser

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.



fill

NOP slide | Exploit code

NOP slide | Exploit code

NOP slide | Exploit code

NOP slide | Exploit code

Heap-spray area

Web page with JavaScript

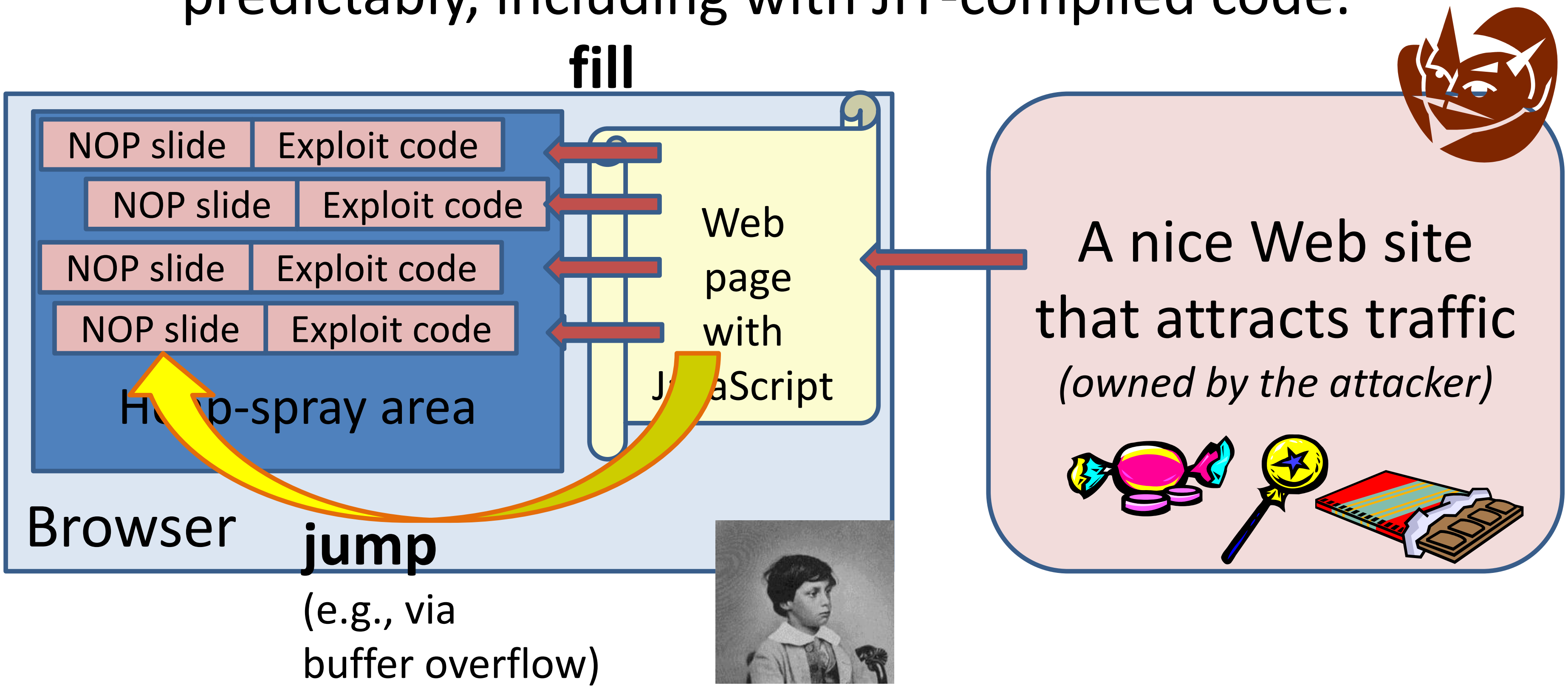
Browser

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.



fill

NOP slide | Exploit code

NOP slide | Exploit code

NOP slide | Exploit code

NOP slide | Exploit code

Heap-spray area

Web page with JavaScript

Browser

A nice Web site that attracts traffic

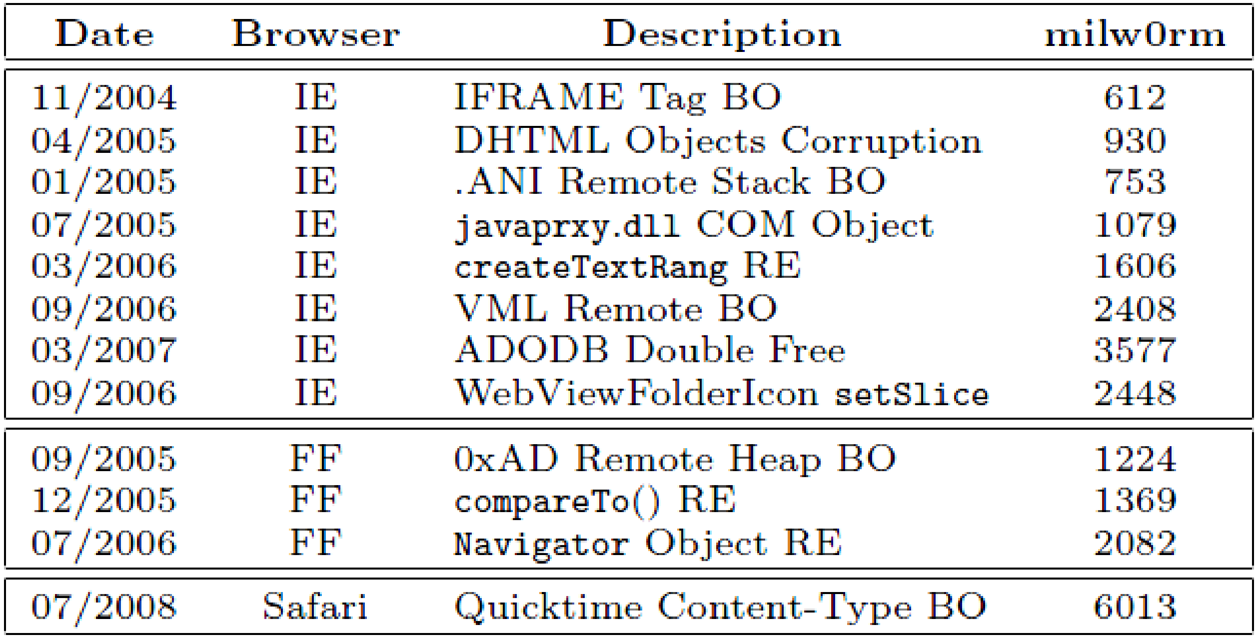
(owned by the attacker)

jump

(e.g., via buffer overflow)

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…

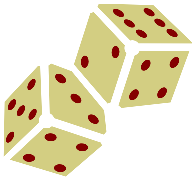
- This secrecy is not always effective.
  - “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.
  - “Heap feng shui” influences heap layout [Sotirov].
  - …

Layout randomization: status

This is an active area, with

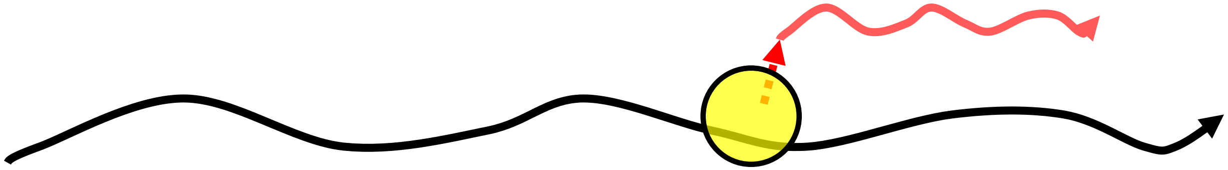
- variants and ongoing improvements to the randomization and its application,
- variants of the attacks,
- techniques detecting or mitigating the attacks.

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



Diverting control flow

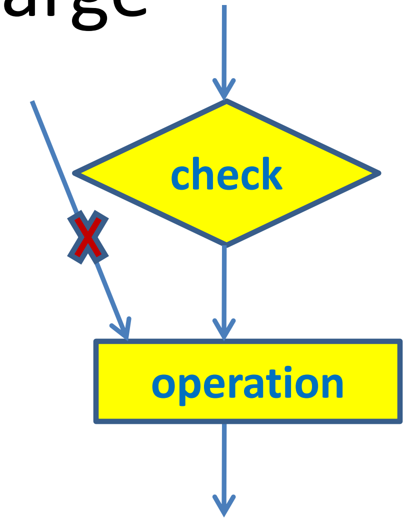
- Many attacks cause some sort of subversion of the expected control flow.



  - 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.

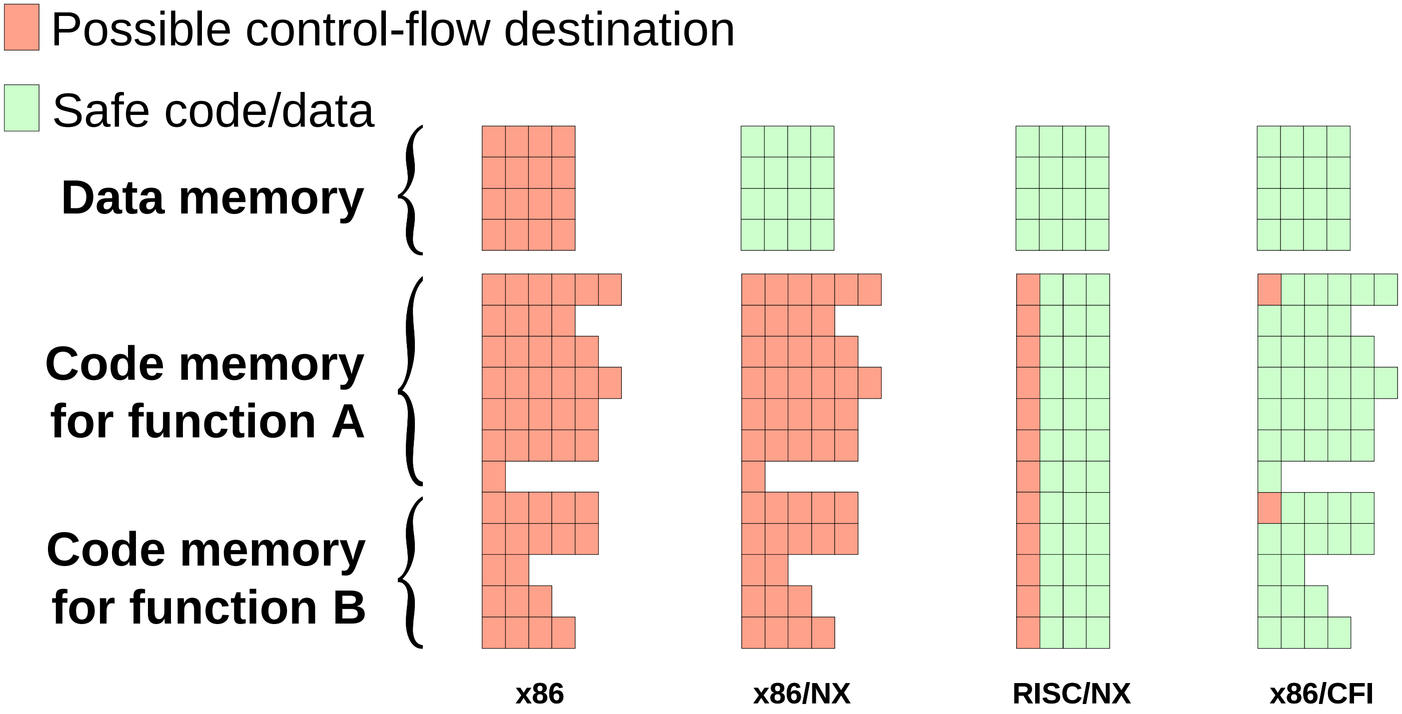
Control-flow integrity (CFI)

- 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):



🟥 Possible control-flow destination

🟩 Safe code/data

Data memory

Code memory for function A

Code memory for function B

x86

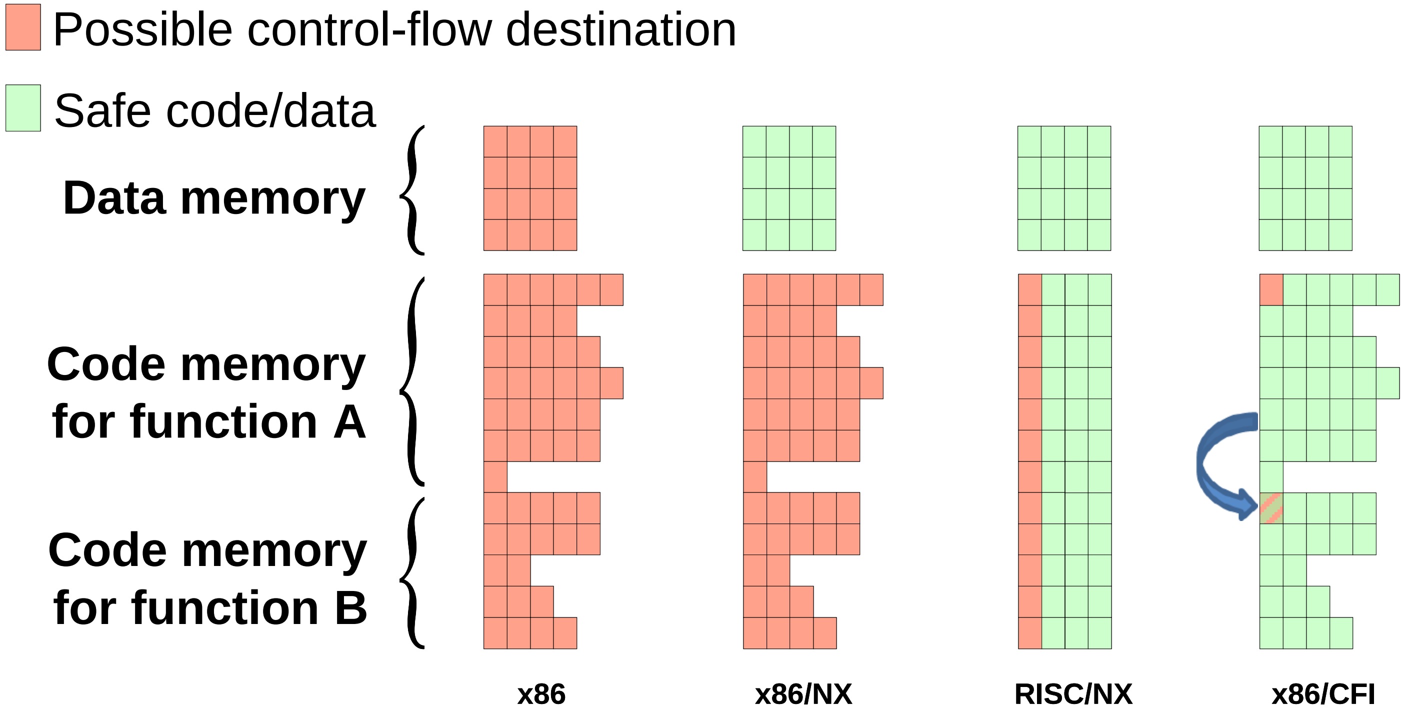
x86/NX

RISC/NX

x86/CFI

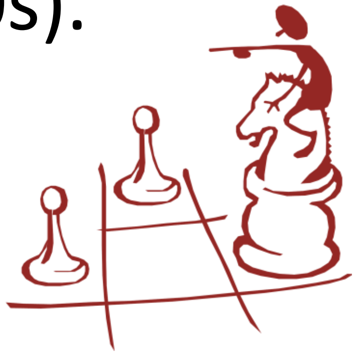
What bytes will the CPU interpret, with CFI? (cont.)

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



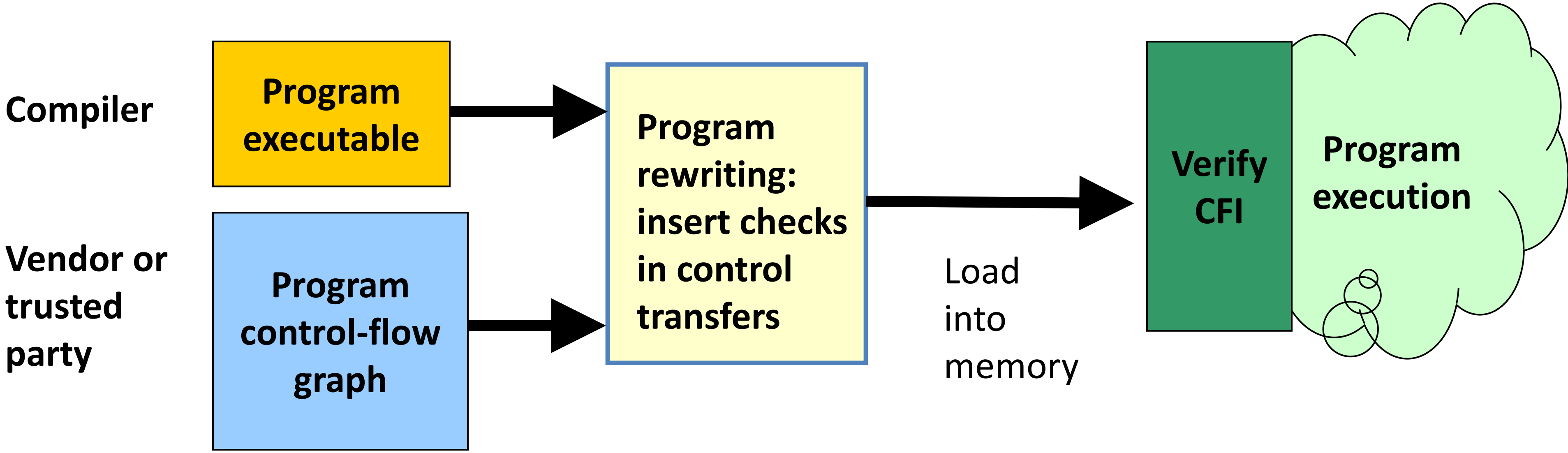
Some implementation strategies for CFI

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.

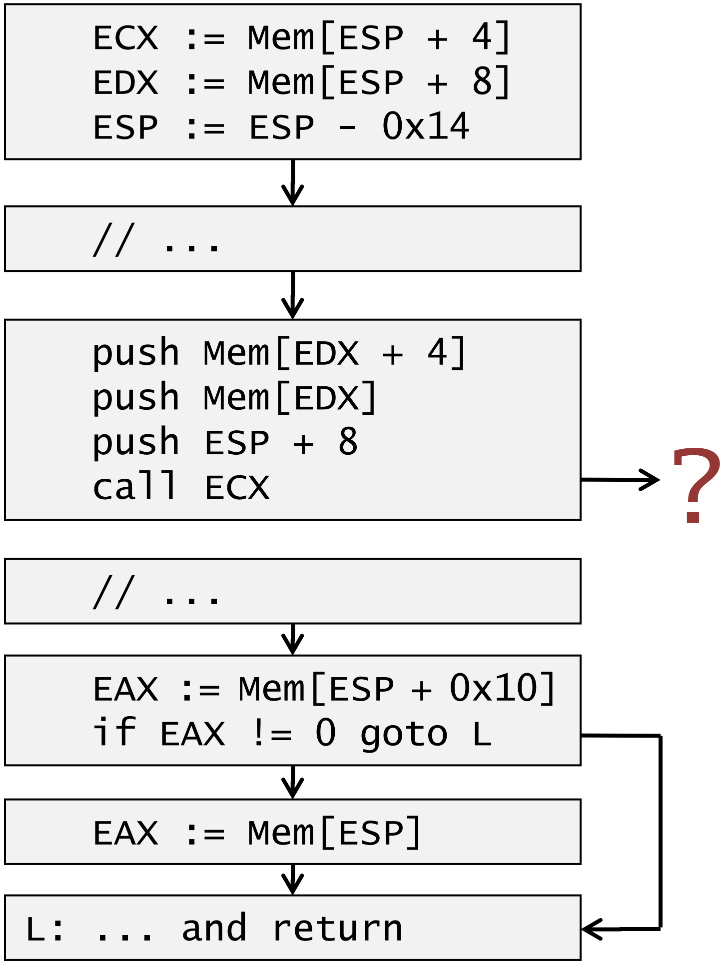
Example

- 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



```
ECX := Mem[ESP + 4]
EDX := Mem[ESP + 8]
ESP := ESP - 0x14
```

```
// ...
```

```
push Mem[EDX + 4]
push Mem[EDX]
push ESP + 8
call ECX
```

?

```
// ...
```

```
EAX := Mem[ESP + 0x10]
if EAX != 0 goto L
```

```
EAX := Mem[ESP]
```

```
L: ... and return
```

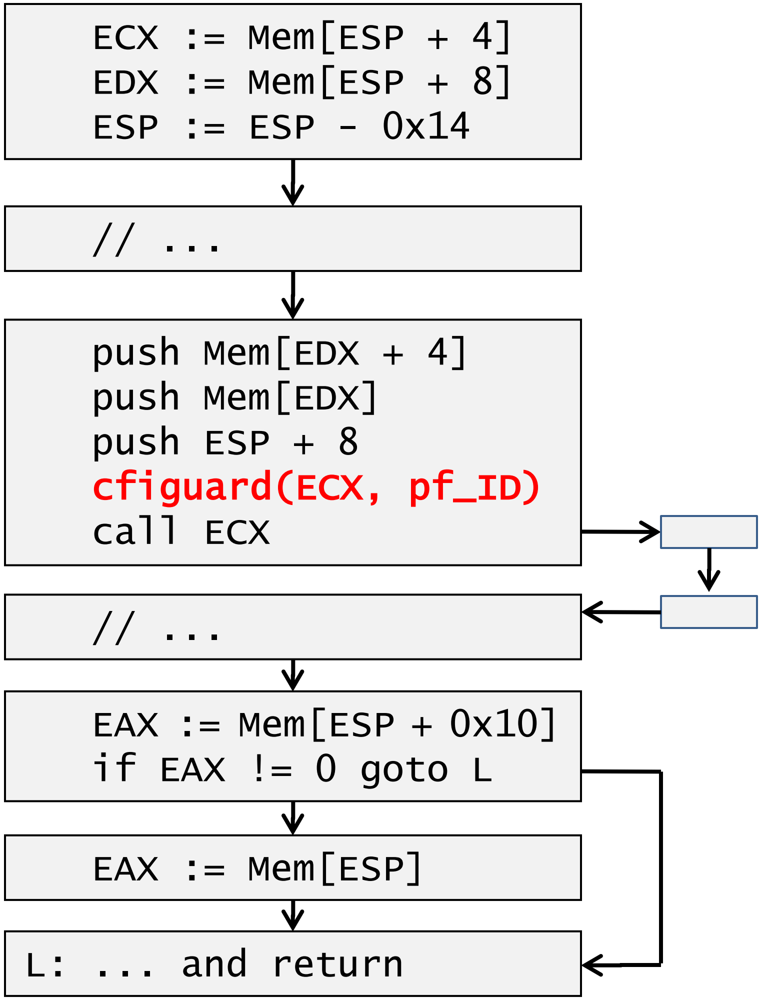
Example (cont.)

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

C source code

```c
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



```
ECX  := Mem[ESP + 4]
EDX  := Mem[ESP + 8]
ESP  := ESP - 0x14
```

```
// ...
```

```
push Mem[EDX + 4]
push Mem[EDX]
push ESP + 8
cfiguard(ECX, pf_ID)
call ECX
```

```
// ...
```

```
EAX := Mem[ESP + 0x10]
if EAX != 0 goto L
```

```
EAX  := Mem[ESP]
```

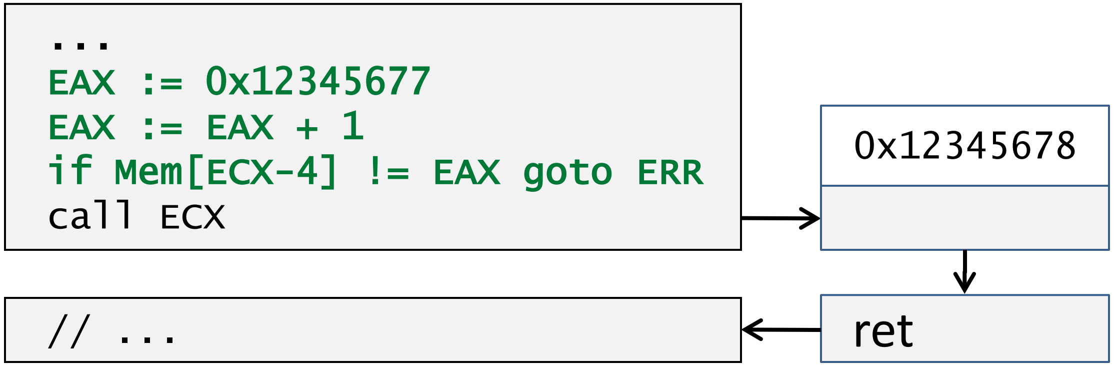
```
L: ... and return
```

A CFI guard

- 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



```
...
EAX := 0x12345677
EAX := EAX + 1
if Mem[ECX-4] != EAX goto ERR
call ECX
```

```
0x12345678
```

```
ret
```

```
// ...
```

Machine code with 0x12345678 as CFI guard ID

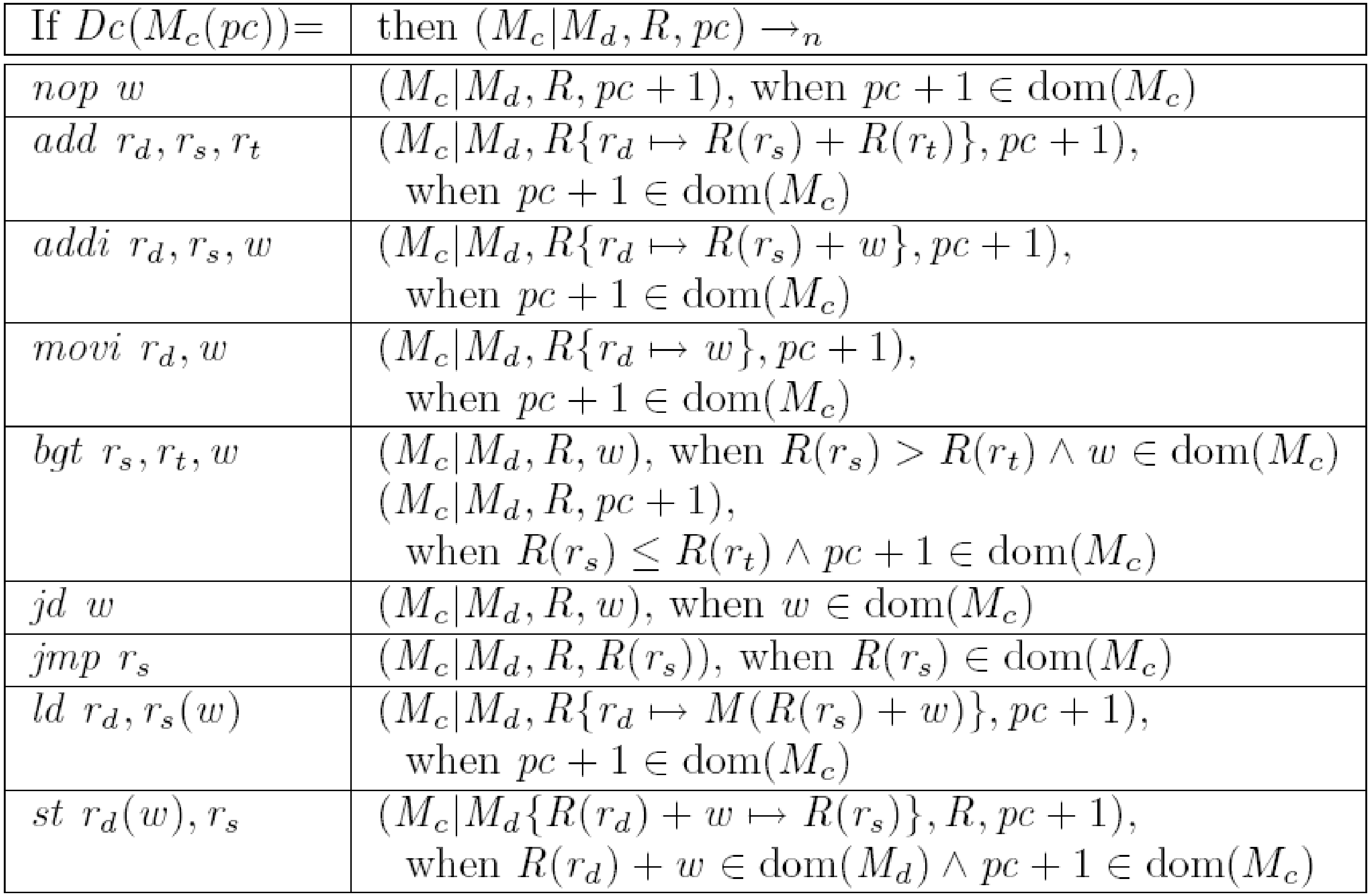
Proving that CFI works

- 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

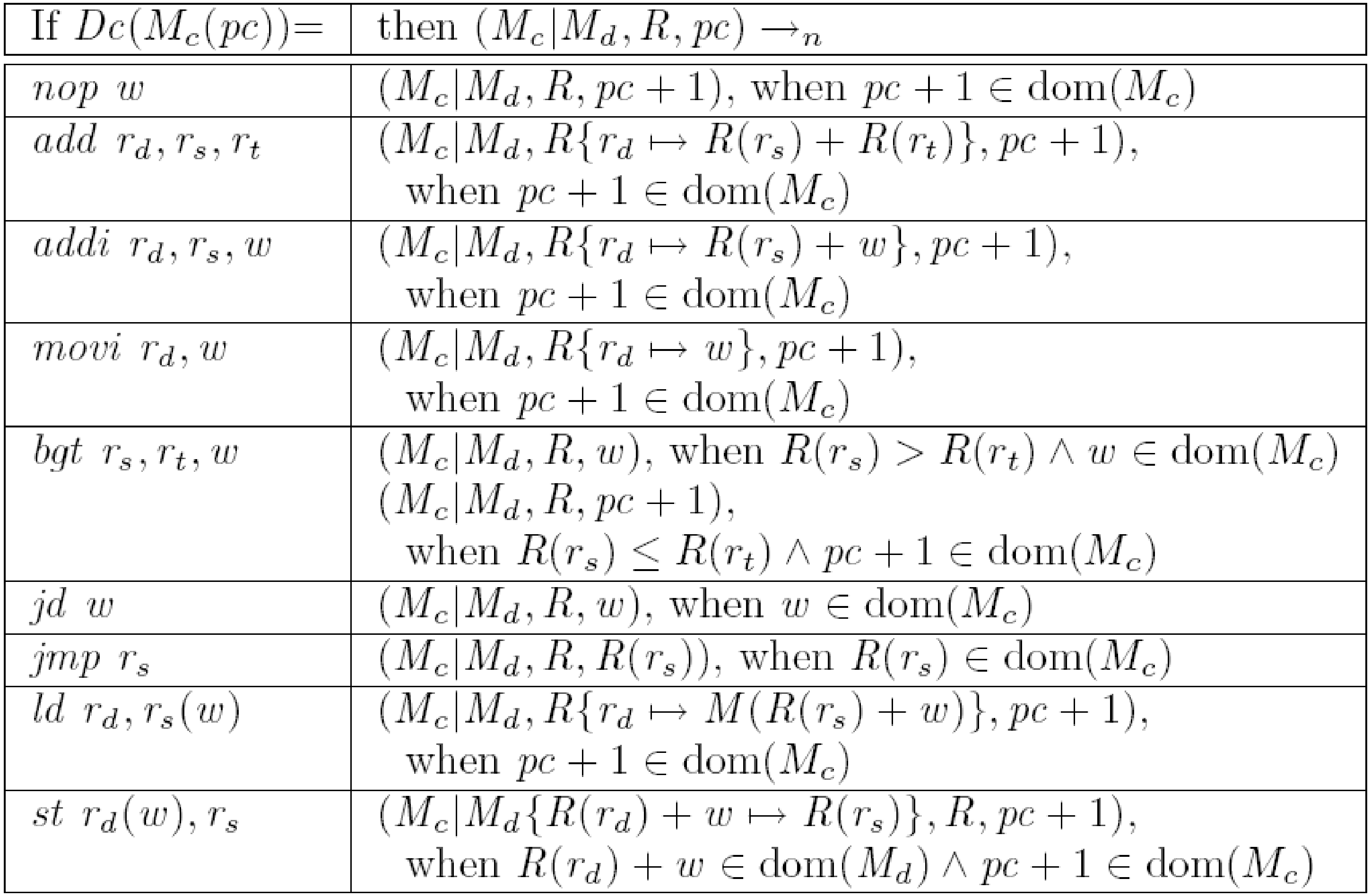
- Instructions: nop, $addi$, $movi$, bgt, jd, jmp, ld, st.
- States: each state is a tuple that includes
  - code memory M_c
  - data memory M_d
  - registers R
  - program counter pc
- Steps: transition relations define the possible state changes of the machine.

1. A small model of a machine



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

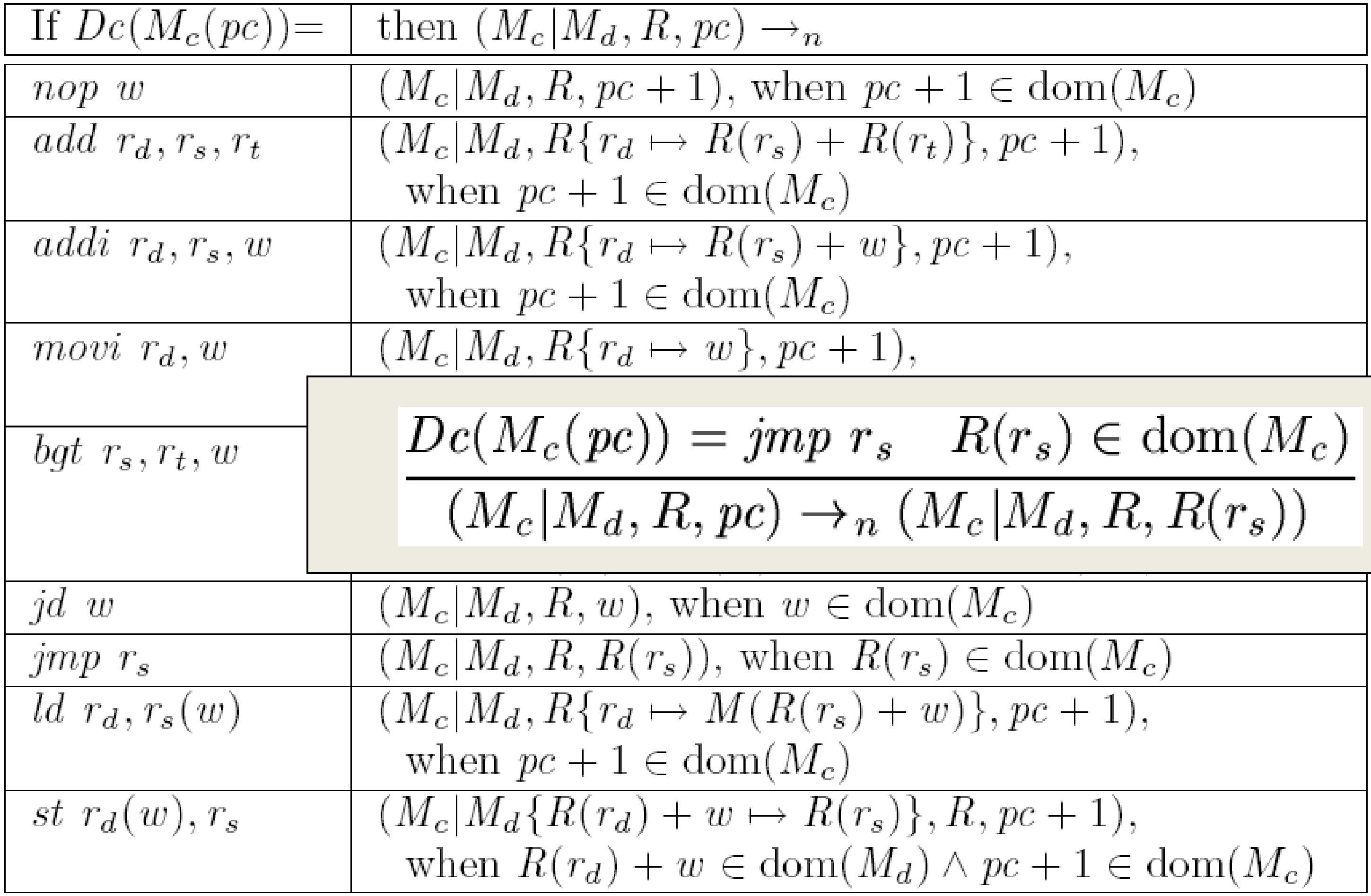
1. A small model of a machine



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

Dc : instruction decoding function

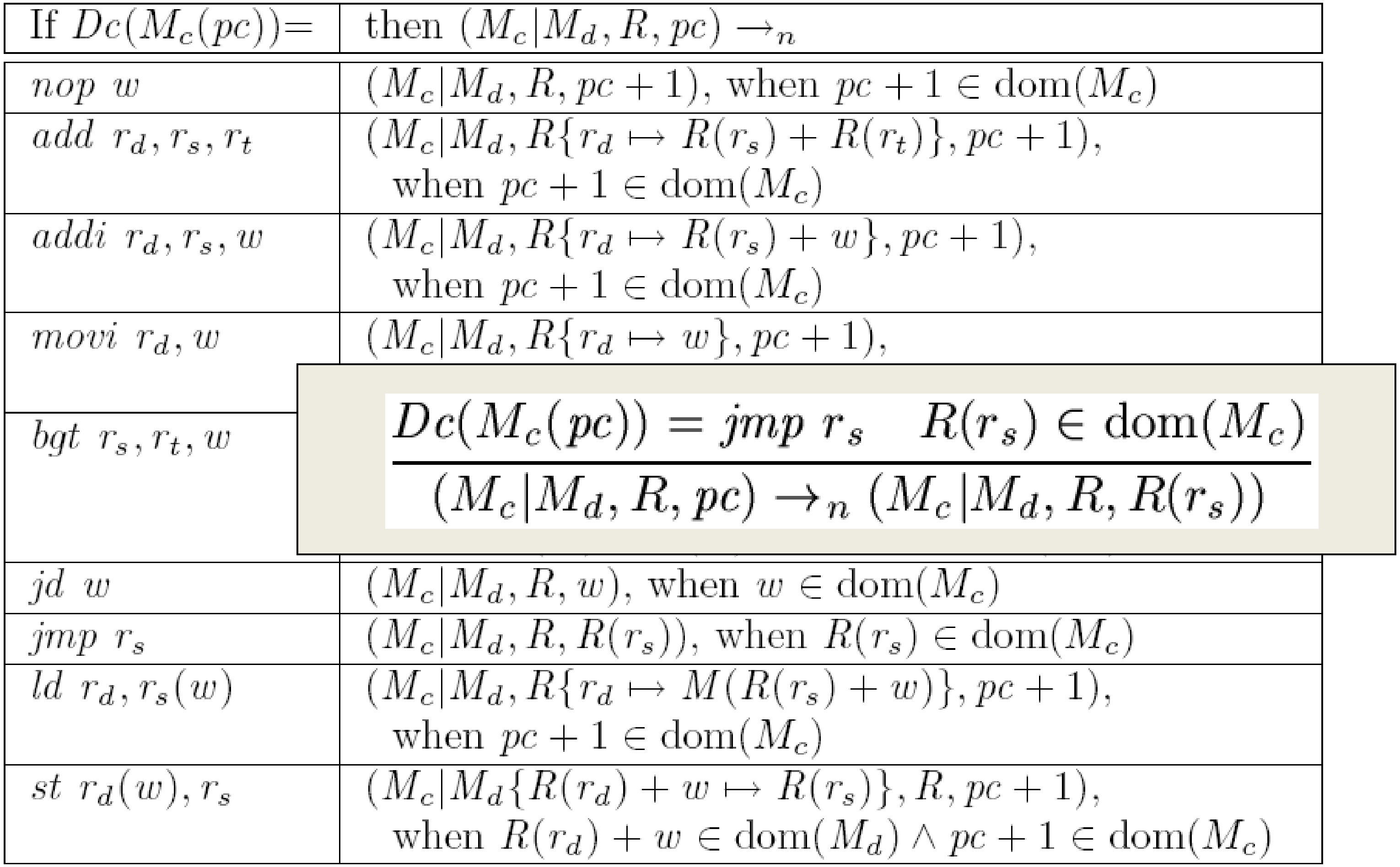
1. A small model of a machine



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

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

1. A small model of a machine



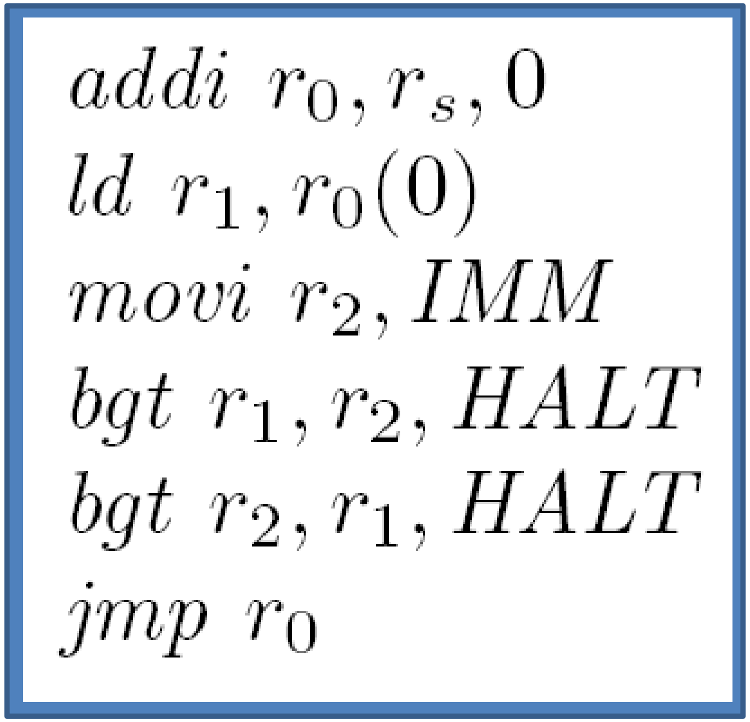
If $Dc(M_c(pc))=$	then $(M_c \mid M_d, R, pc) \to_n$
$nop\ w$	$(M_c \mid M_d, R, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$add\ r_d, r_s, r_t$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+R(r_t)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$addi\ r_d, r_s, w$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+w\}, pc+1)$, 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 $R(r_d)+w \in \mathrm{dom}(M_d) \wedge pc+1 \in \mathrm{dom}(M_c)$

$$\frac{Dc(M_c(pc)) = jmp\ r_s \qquad R(r_s) \in \mathrm{dom}(M_c)}{(M_c|M_d, R, pc) \to_n (M_c|M_d, R, R(r_s))}$$

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

2. Example condition on instrumentation

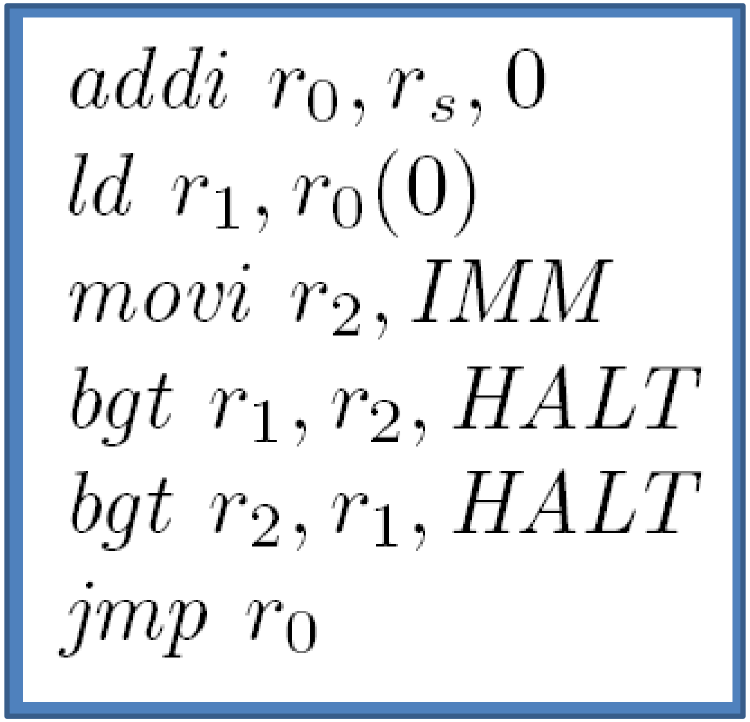
Computed jumps occur only in context of a specific instruction sequence:



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

2. Example condition on instrumentation

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



$addi\ r_0, r_s, 0$

$ld\ r_1, r_0(0)$

$movi\ r_2, IMM$

$bgt\ r_1, r_2, HALT$

$bgt\ r_2, r_1, HALT$

$jmp\ r_0$

HALT is the address of a halt instruction.

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

(For this simple model, we do not need to add 1.)

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.

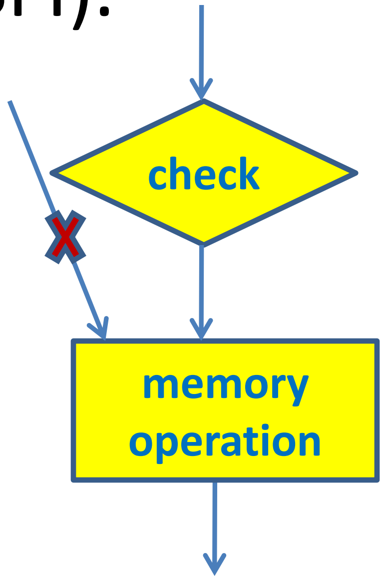
Suppose $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$
where each $\rightarrow$ transition is either a normal $\rightarrow_n$ step or an attacker step that changes only data memory.

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.

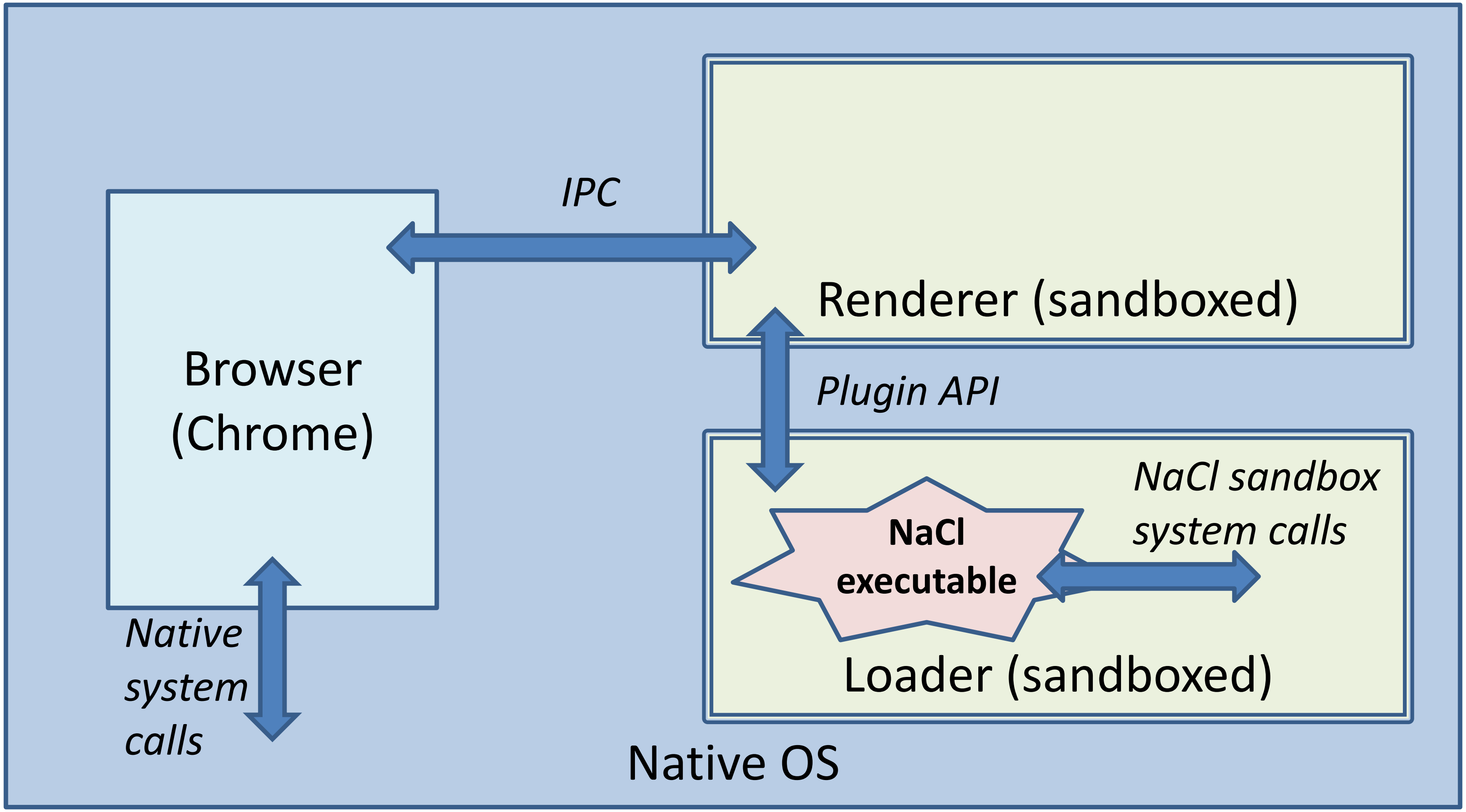
Proof: by a tedious induction.

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.



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



IPC

Browser (Chrome)

Renderer (sandboxed)

Plugin API

NaCl sandbox system calls

NaCl executable

Loader (sandboxed)

Native system calls

Native OS

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.

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

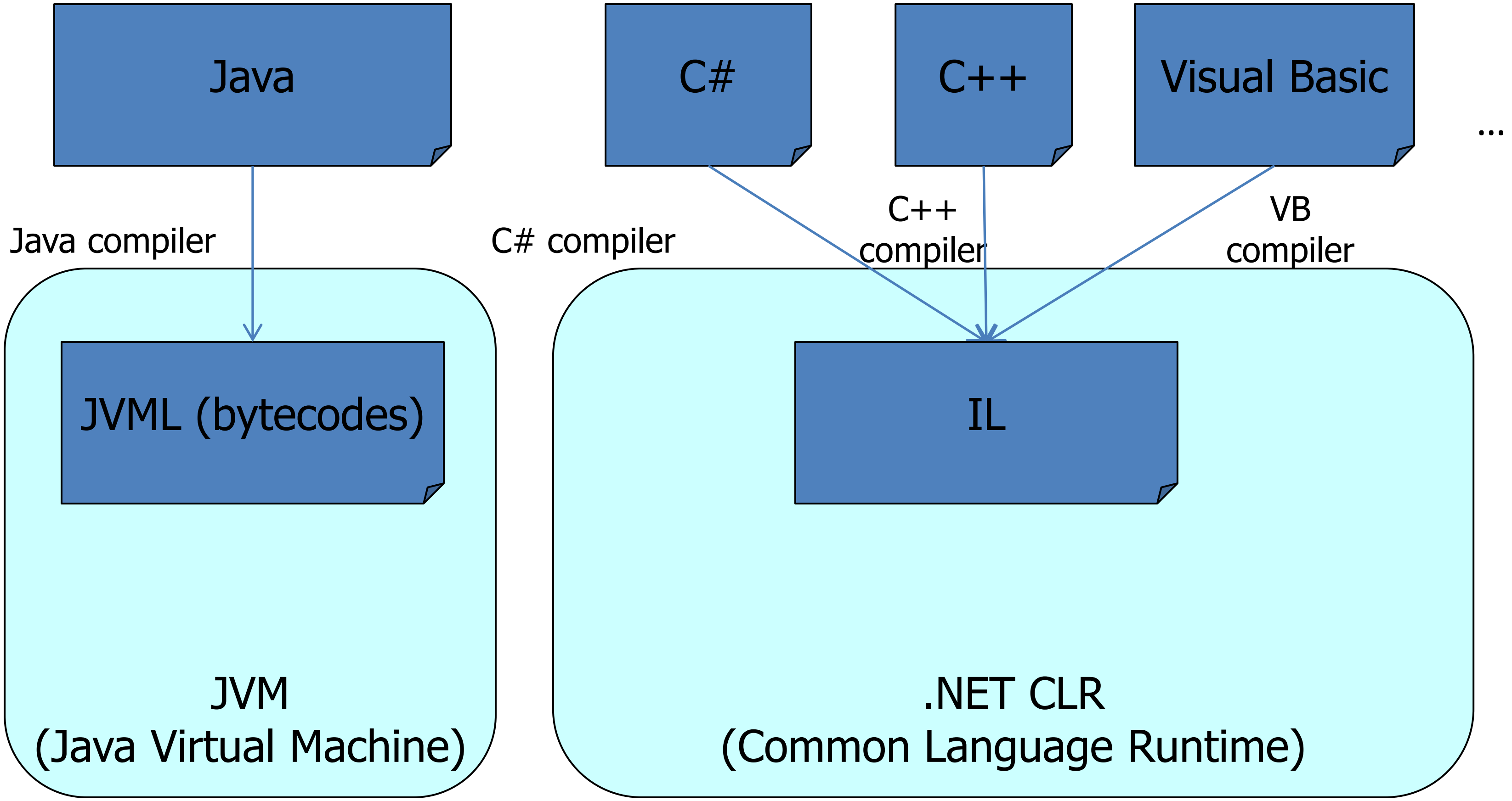
Mismatch in characteristics:

- 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



Java

C#

C++

Visual Basic

…

Java compiler

C# compiler

C++ compiler

VB compiler

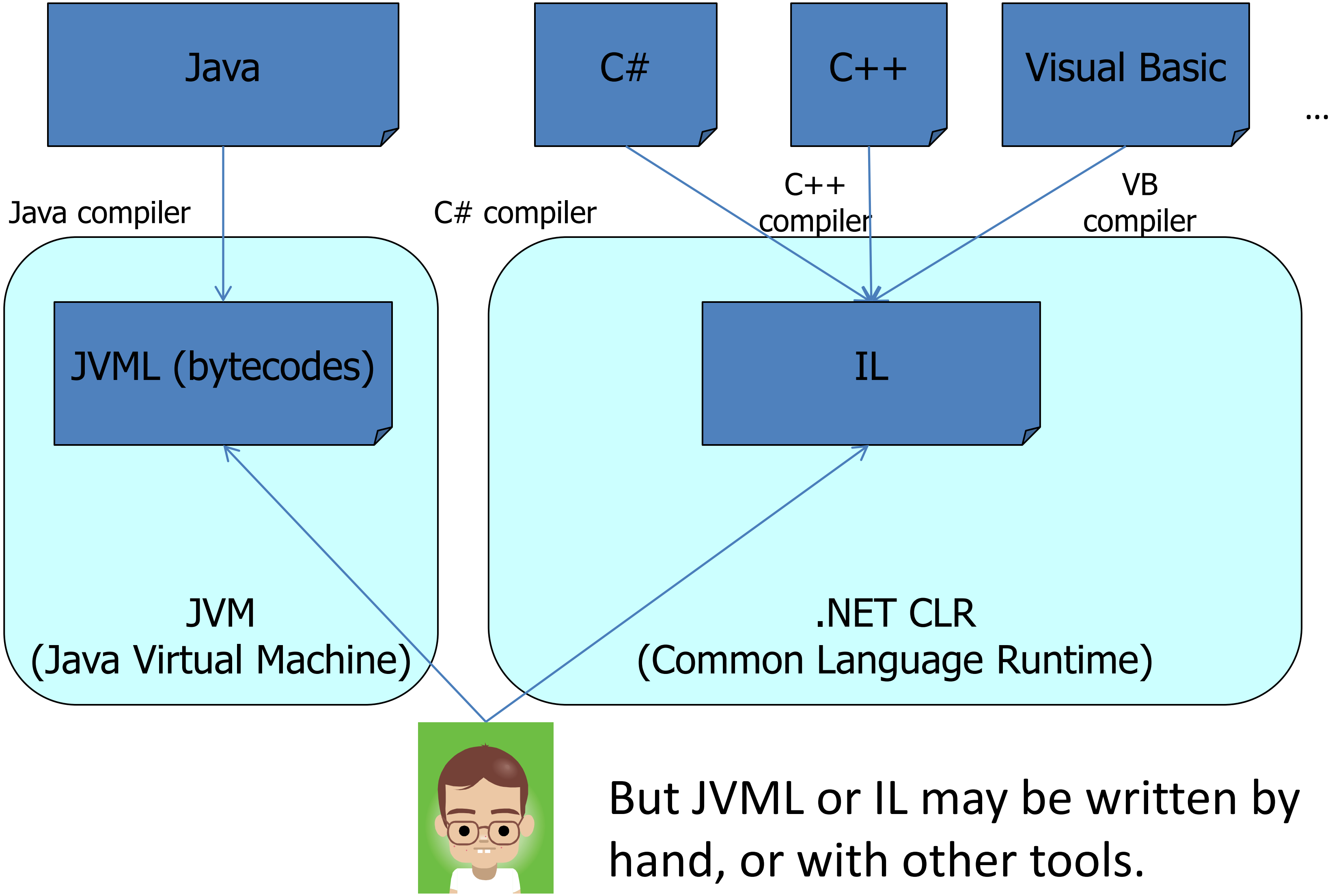
JVML (bytecodes)

IL

JVM
(Java Virtual Machine)

.NET CLR
(Common Language Runtime)

“Secure” programming platforms



Java

C#

C++

Visual Basic

…

Java compiler

C# compiler

C++ compiler

VB compiler

JVML (bytecodes)

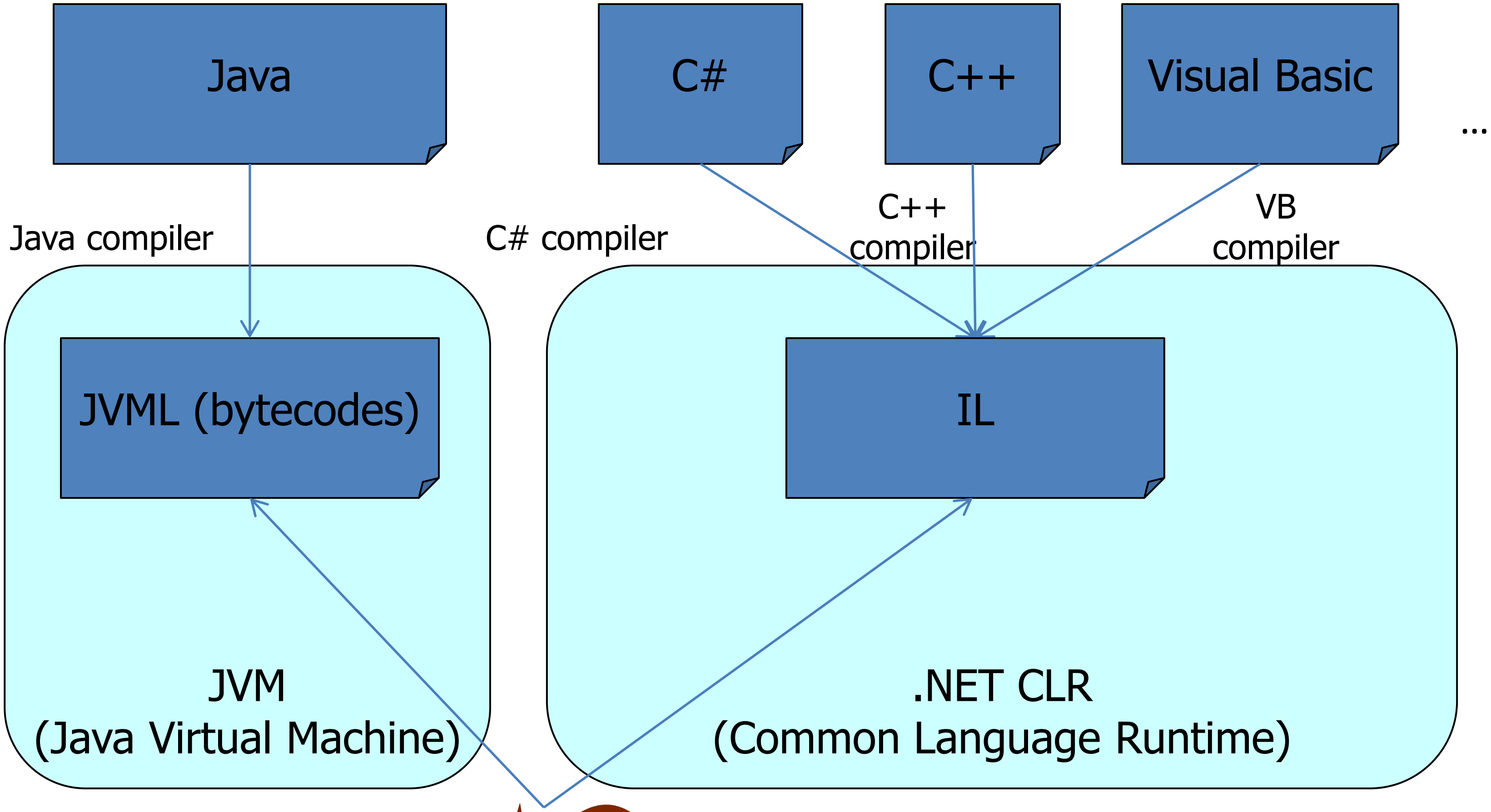
IL

JVM
(Java Virtual Machine)

.NET CLR
(Common Language Runtime)

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

“Secure” programming platforms



Java

C#

C++

Visual Basic

…

Java compiler

C# compiler

C++ compiler

VB compiler

JVML (bytecodes)

IL

JVM
(Java Virtual Machine)

.NET CLR
(Common Language Runtime)

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

Mediated access

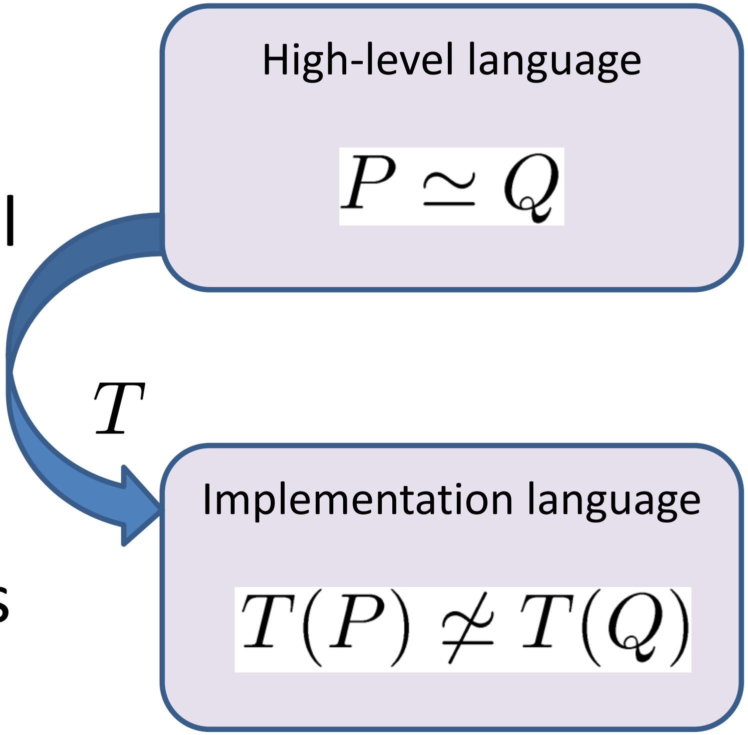
```
class Widget {// No checking of argument
  virtual void Operation(string s) {…};
}
class SecureWidget : Widget {
  // Validate argument and pass on
  // Could also authenticate the caller
  override void Operation(string s) {
    Validate(s);
    base Operation(s);
```

```
// 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.



High-level language

$P \simeq Q$

T

Implementation language

$T(P) \not\simeq T(Q)$

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

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.

Some reading

- Úlfar Erlingsson’s tutorial paper “Low-level Software Security: Attacks and Defenses” (2007).
- “Protection in Programming Languages”, by Jim Morris (1973).
- “Securing the .NET Programming Model”, by Andrew Kennedy (2006).