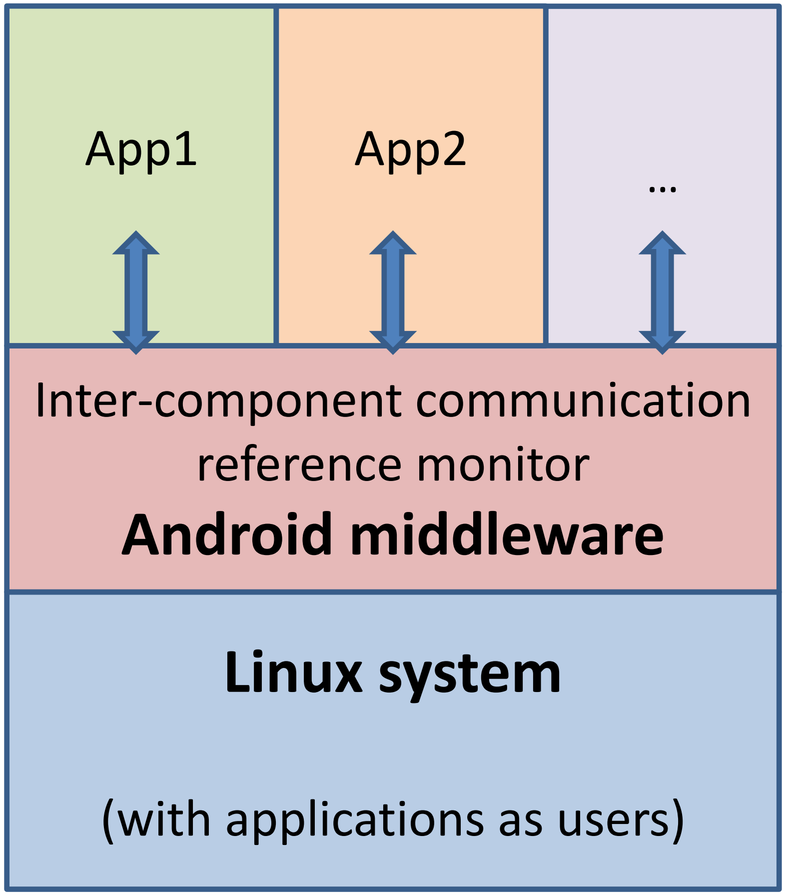
Low-level software security



App1

App2

…

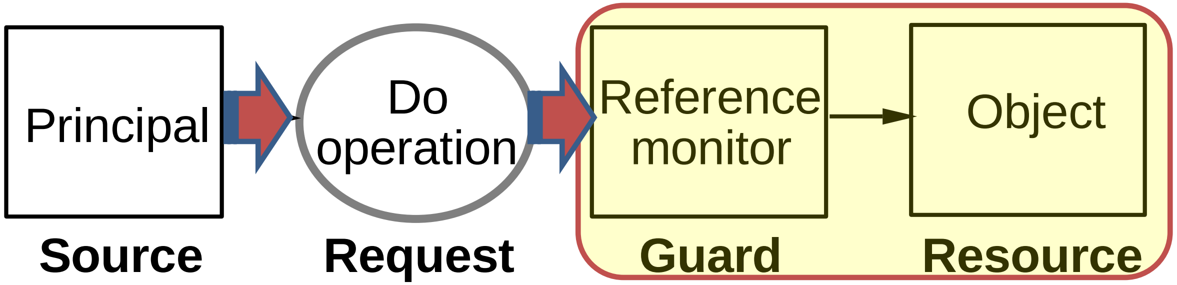
Inter-component communication reference monitor

Android middleware

Linux system

(with applications as users)

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



Principal

Do operation

Reference monitor

Object

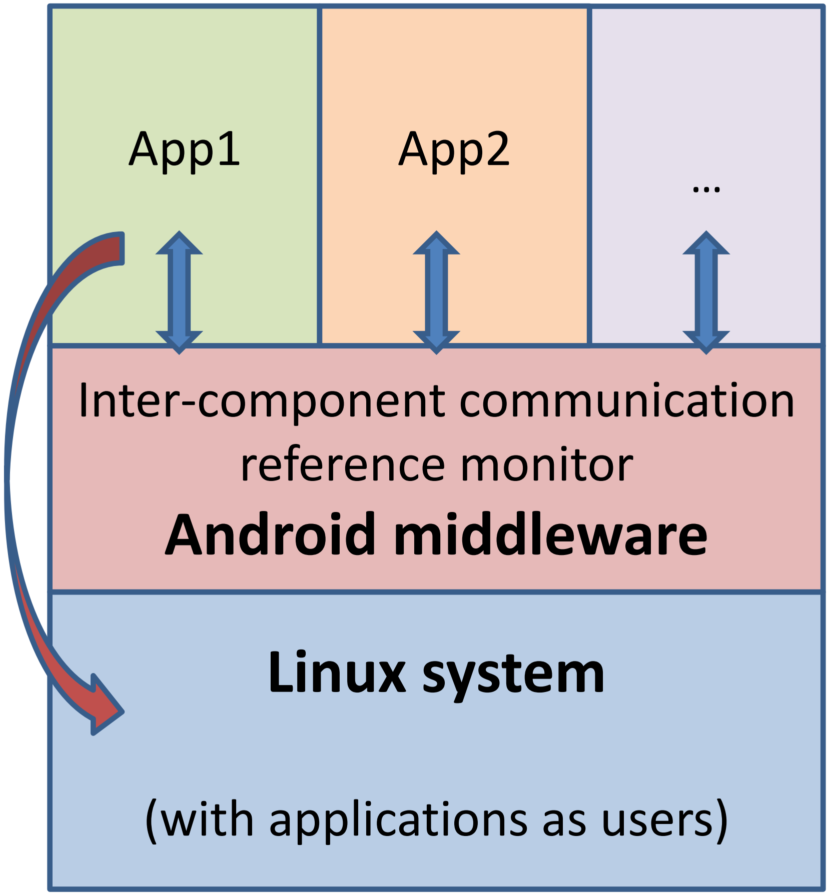
Source

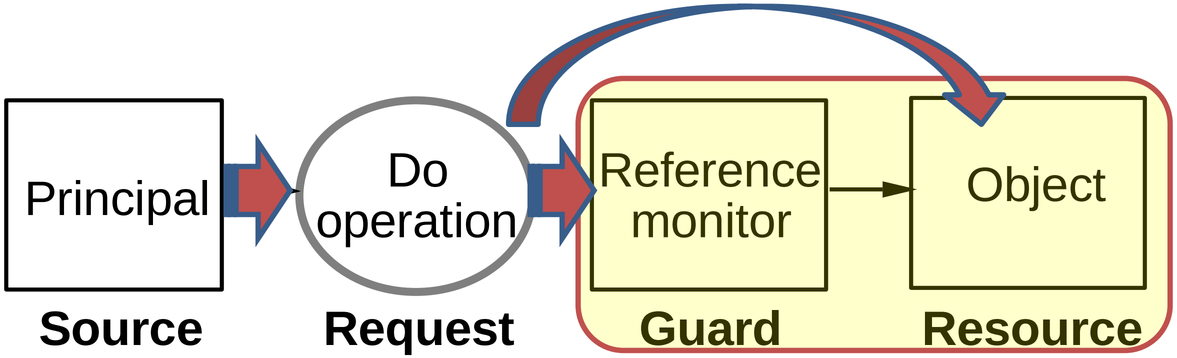
Request

Guard

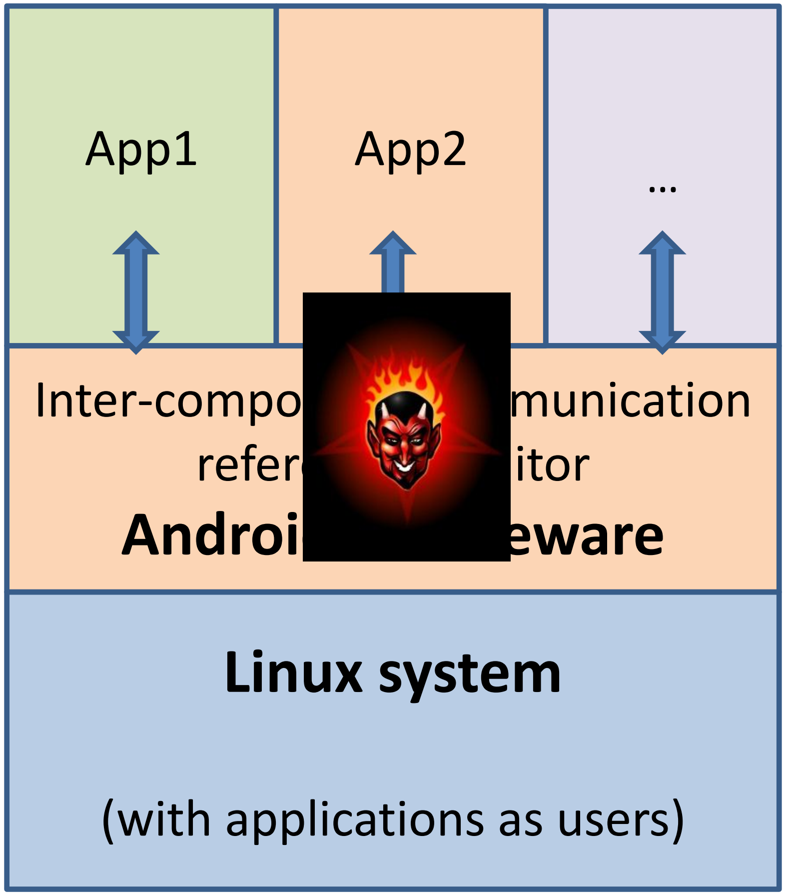
Resource

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.



App1

App2

…

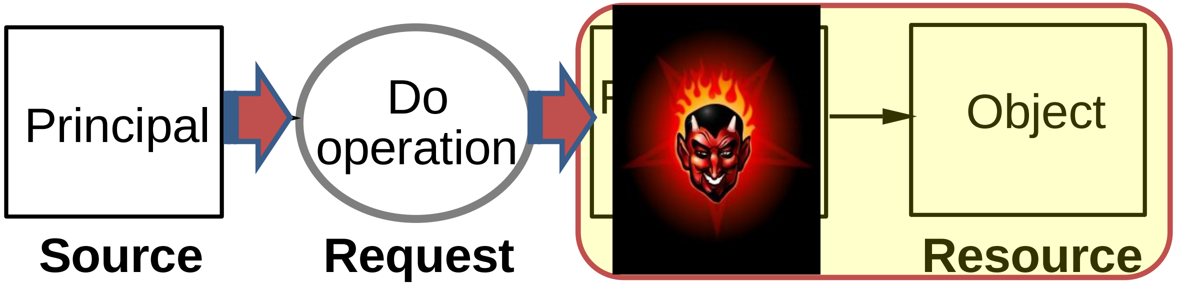
Inter-compo munication

refer itor

Androi eware

Linux system

(with applications as users)



Principal

Do operation

F

Object

Source

Request

Resource

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.

Software security

Software security is

- not only about implementation flaws,
- not only about low-level attacks and defenses,
- certainly not only about buffer overflows,

but low-level attacks and defenses

- remain important,
- illustrate themes and techniques that appear throughout software systems.

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_authenti
}
```

- 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

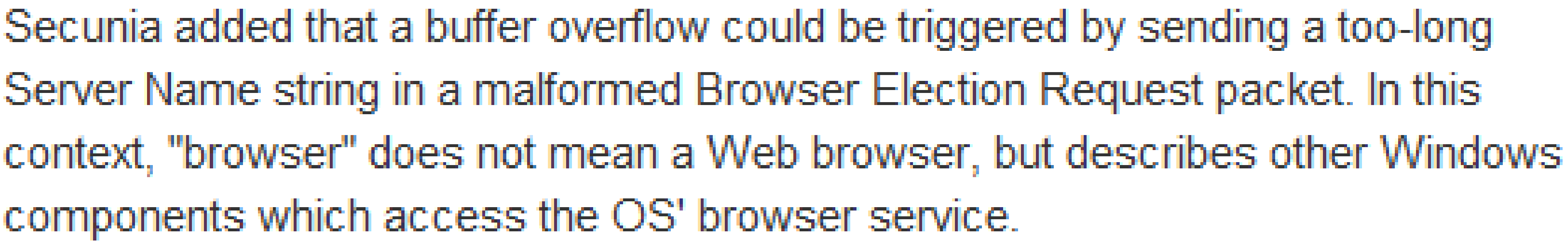
New Windows zero-day surfaces as researcher releases attack code

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;
            …
```

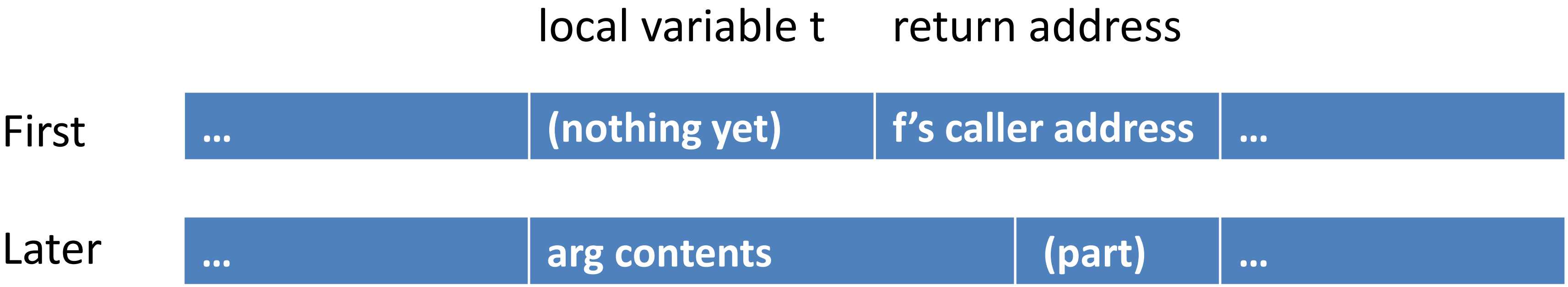
- If this size is too big and not checked (either statically or dynamically), there can be trouble.
- In memory, we could have:

		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;
        …
```

- 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

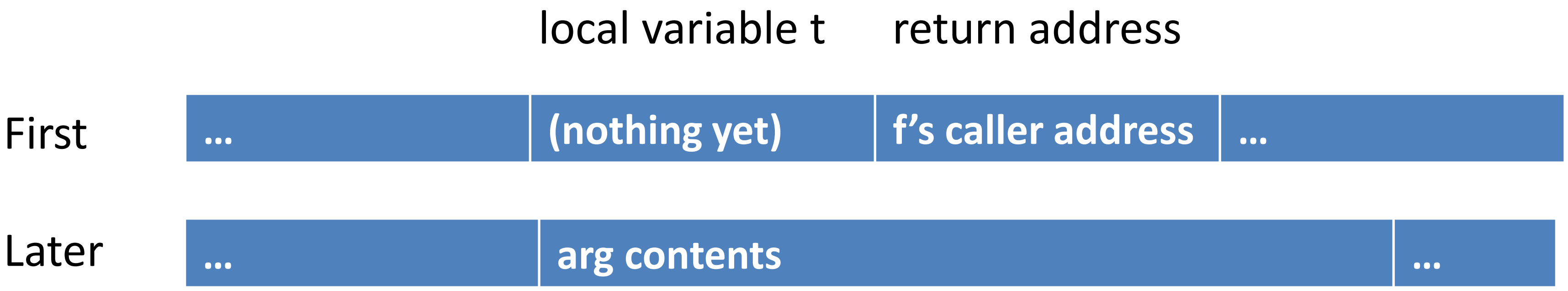
(part)

…

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** | **…**

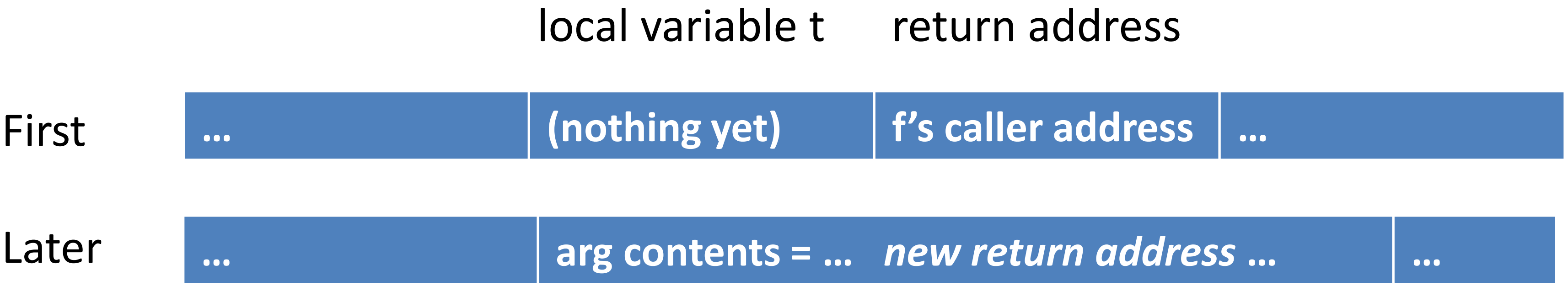
Later

… | **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:

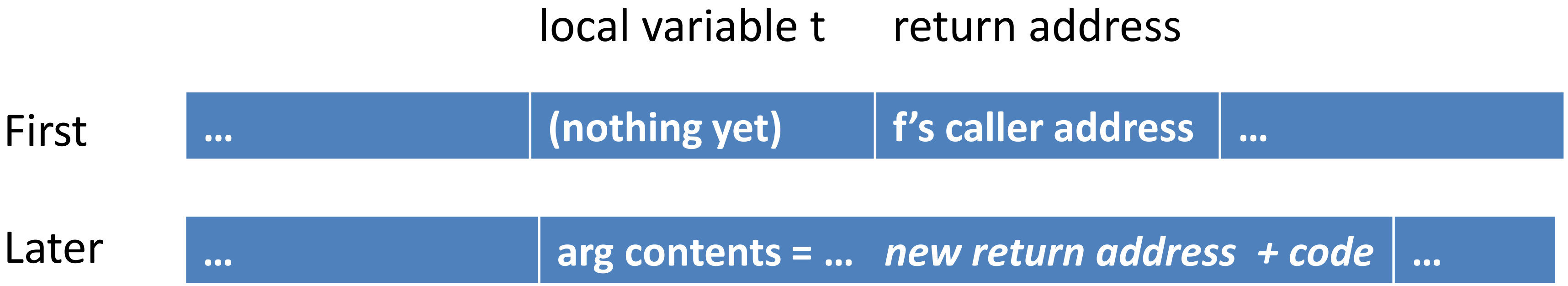


		local variable t	return address	
First	**…**	**(nothing yet)**	**f's caller address**	**…**
Later	**…**	**arg contents = …**	***new return address …***	**…**

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

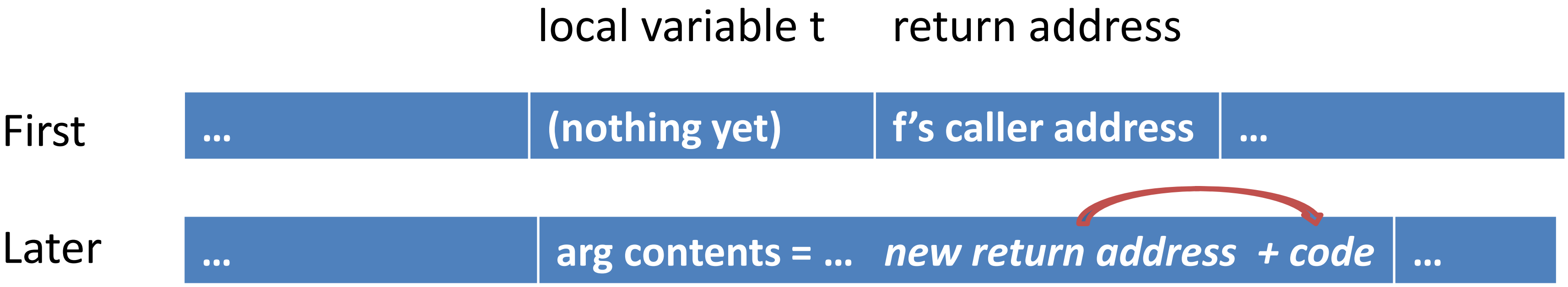
First
… | **(nothing yet)** | **f's caller address** | **…**

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:

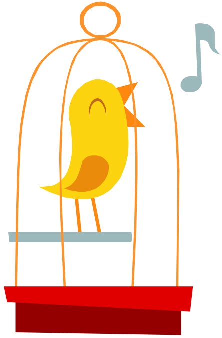


local variable t   return address

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

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

Stack canaries and cookies



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

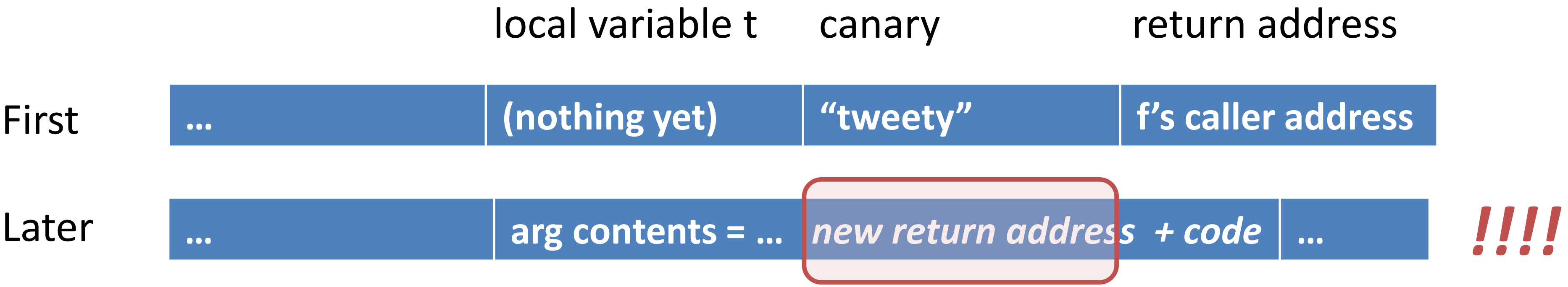
- A known quantity (fixed or random) can be inserted between the local variable and the return address so that any corruption can be detected.

		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.

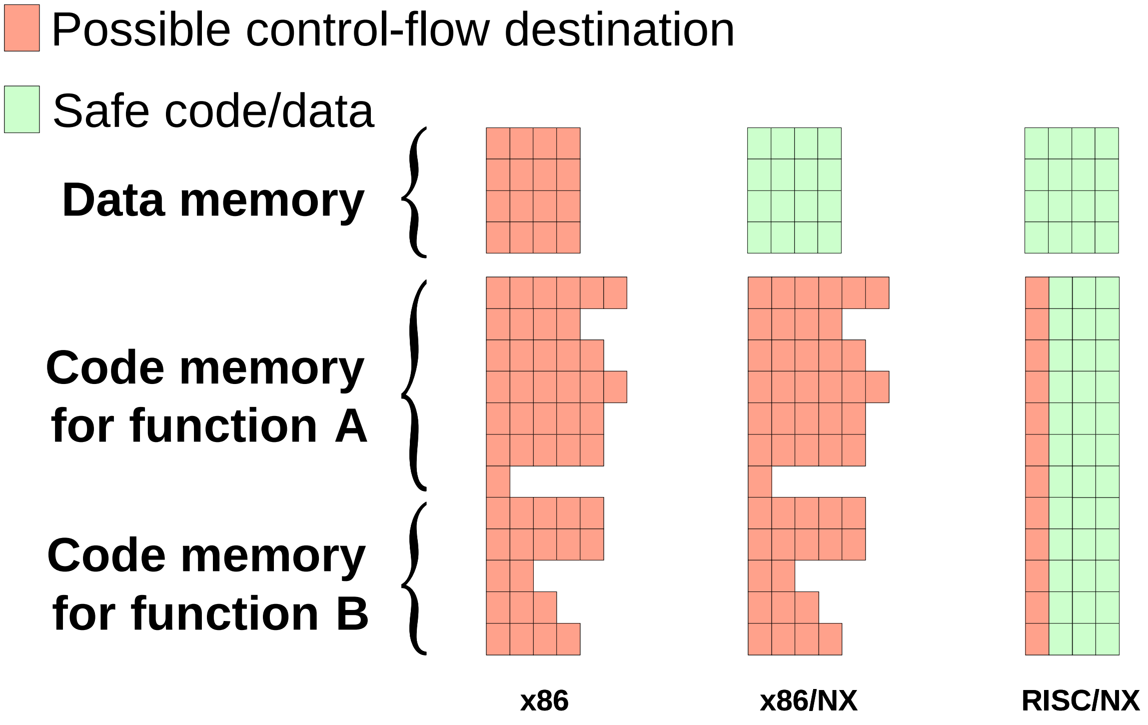
⇒ ***So a defense is to require that data that is writable cannot be executed.****

- 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

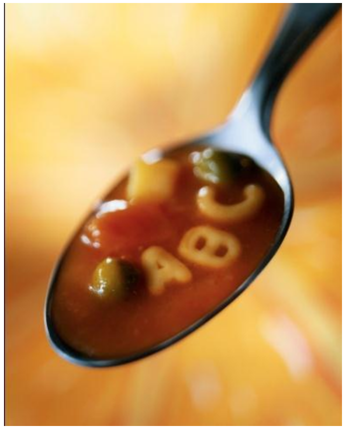
x86

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

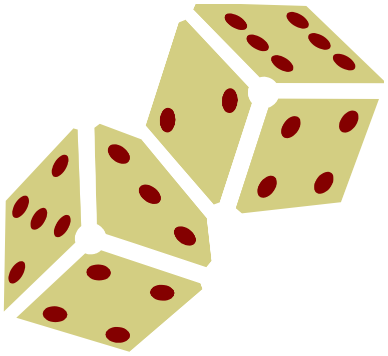
```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

Layout randomization

Attacks often depend on addresses.

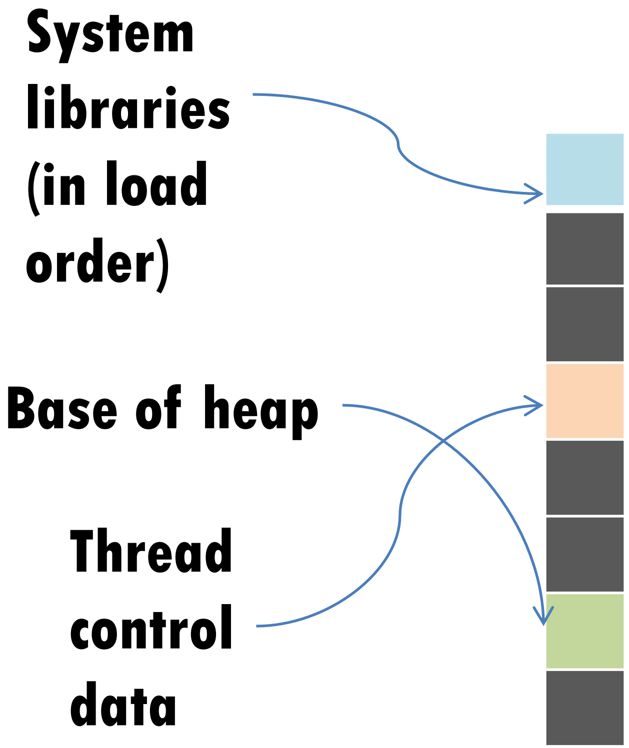
⇒ ***Let us randomize the addresses!***

- 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.



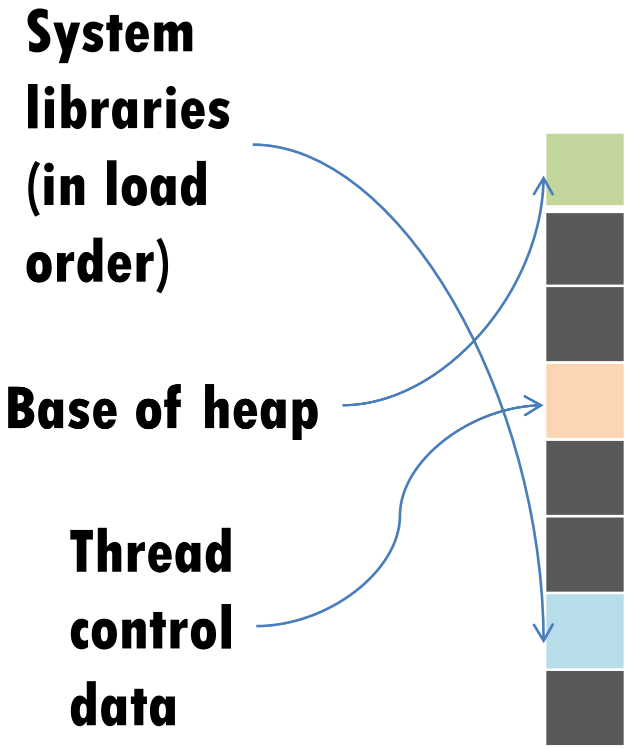
System libraries (in load order)

Base of heap

Thread control data

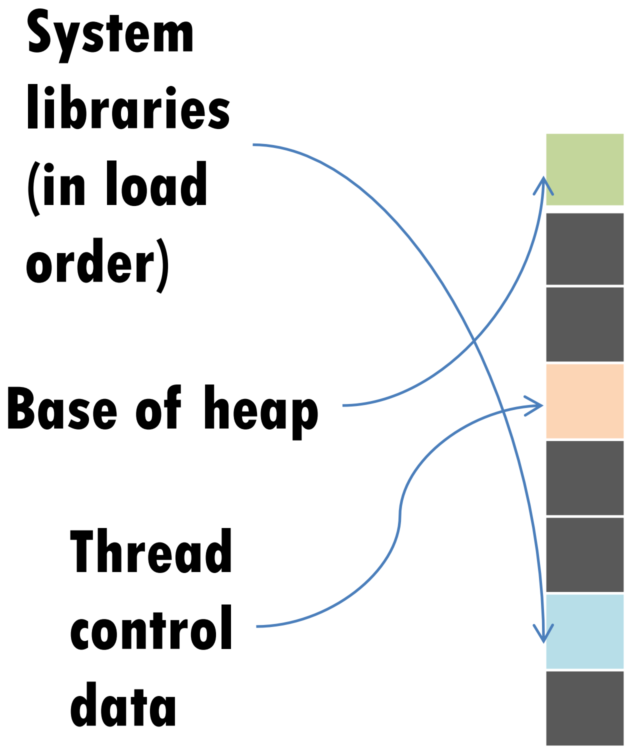
Implementations

- 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.



System libraries (in load order)

Base of heap

Thread control data

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:

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

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:

$$\begin{aligned} &l : \texttt{loc} \ (l \in \mathrm{Loc}) \\ &!_{\texttt{loc}} : \texttt{loc} \to \texttt{nat} \\ &:=_{\texttt{loc}} : \texttt{loc} \times \texttt{nat} \to \texttt{unit} \\ &\texttt{run}_{\texttt{loc}} : \texttt{loc} \to \texttt{unit} \end{aligned}$$

- Some semantics:

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

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.

$$l : \texttt{nat} \qquad (\text{for } l \in \text{Loc})$$

$$!_{\texttt{nat}} : \texttt{nat} \to \texttt{nat}$$

$$:=_{\texttt{nat}} : \texttt{nat} \times \texttt{nat} \to \texttt{unit}$$

$$\texttt{run}_{\texttt{nat}} : \texttt{nat} \to \texttt{unit}$$

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 $\sigma \to \mathtt{bool}$
(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

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

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

Preserving equivalences

(“full abstraction”)

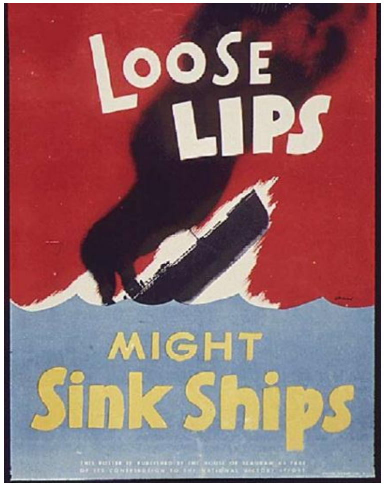
With each high-level program M
we associate a low-level program $M^{\downarrow}$.

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.

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.

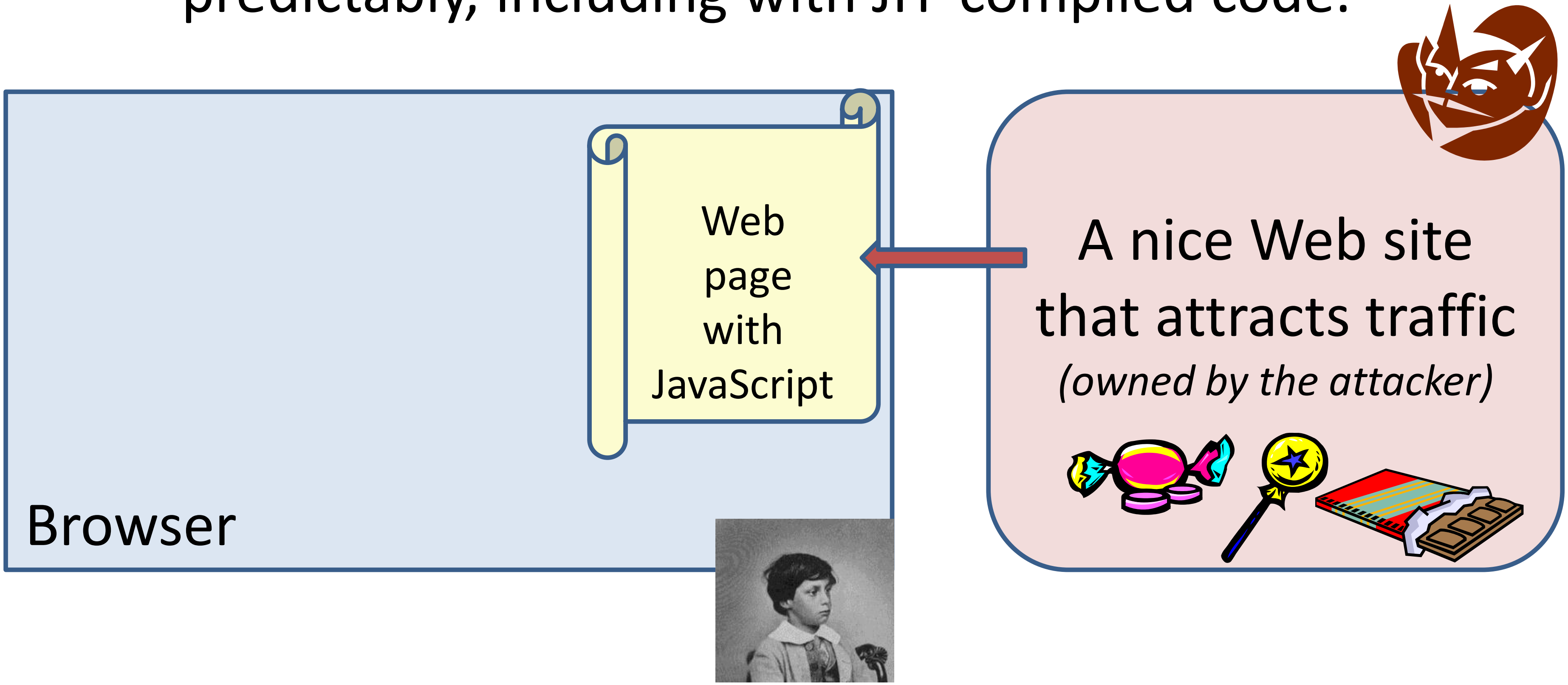
Browser

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

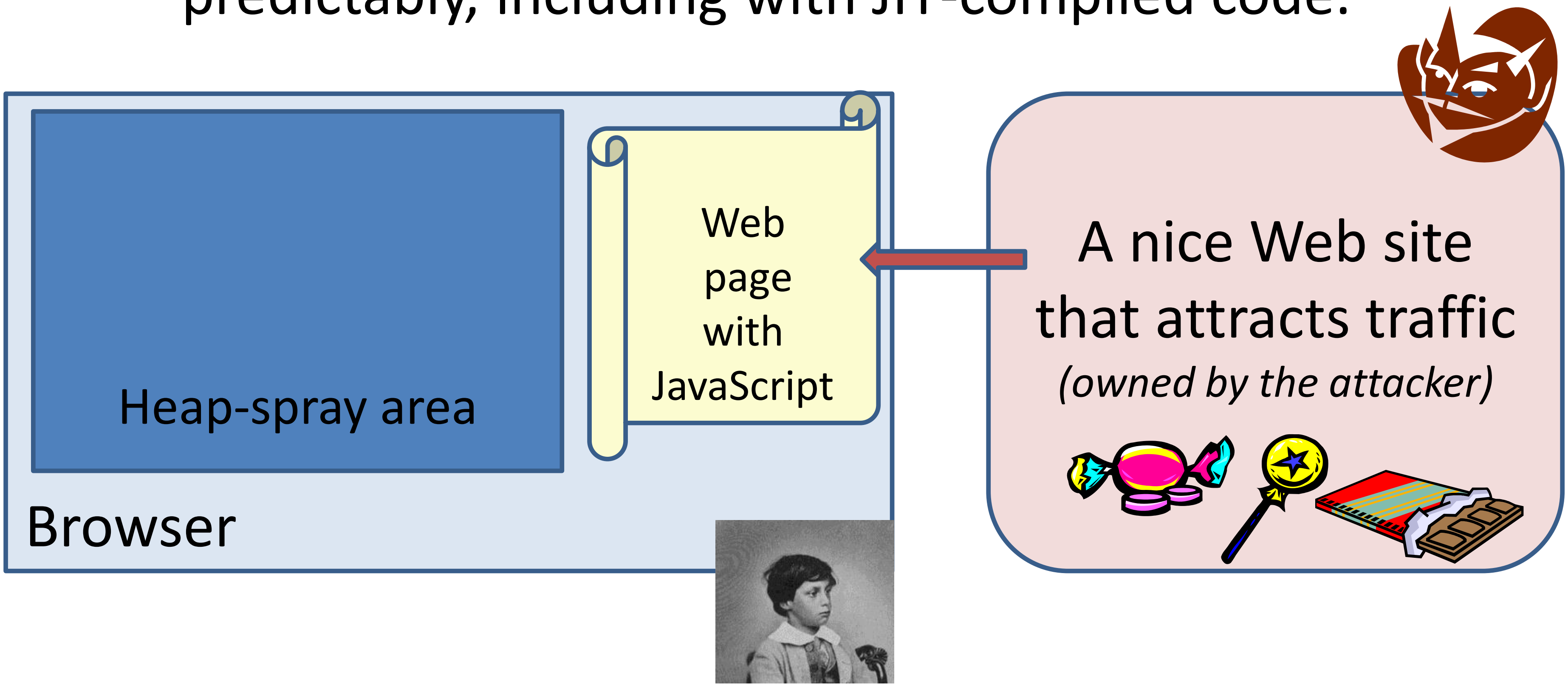
A nice Web site that attracts traffic

(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

Browser

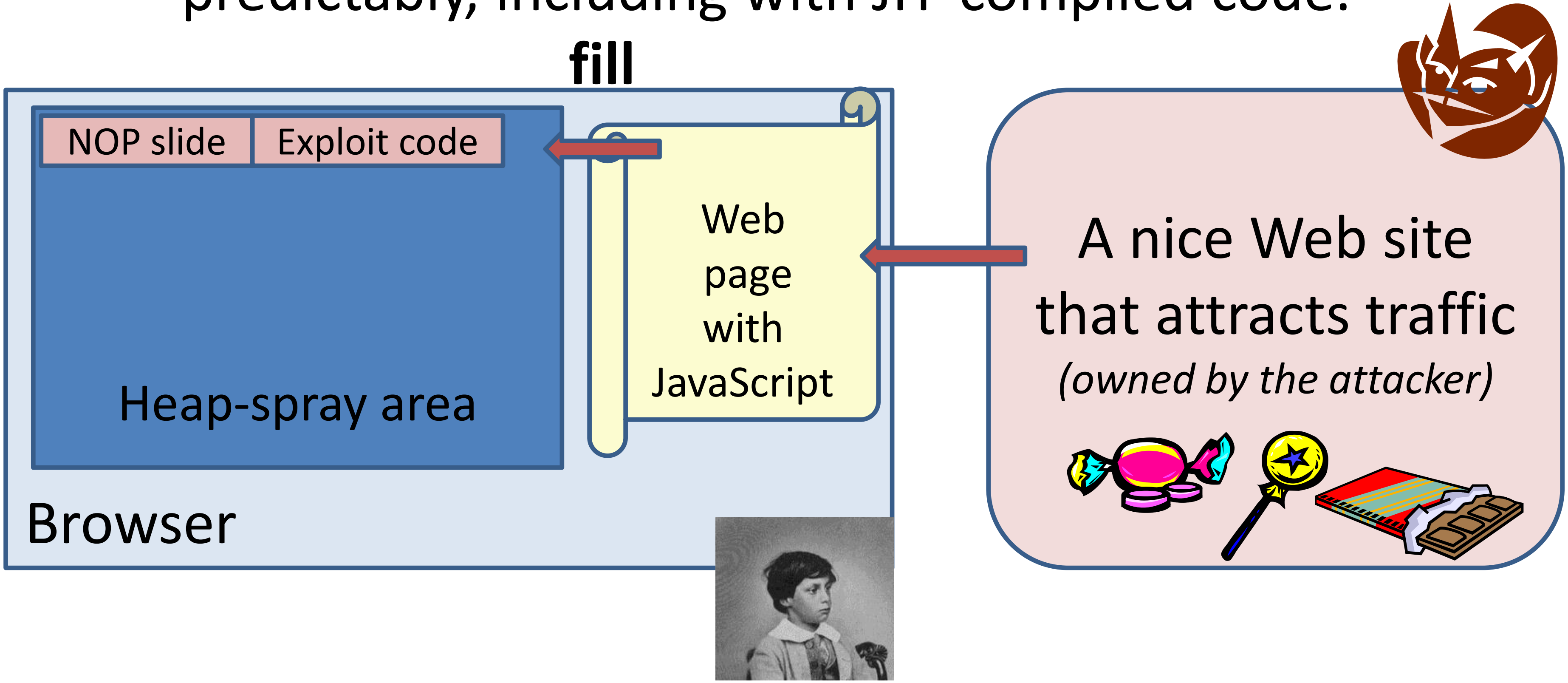
Web page with JavaScript

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

Browser

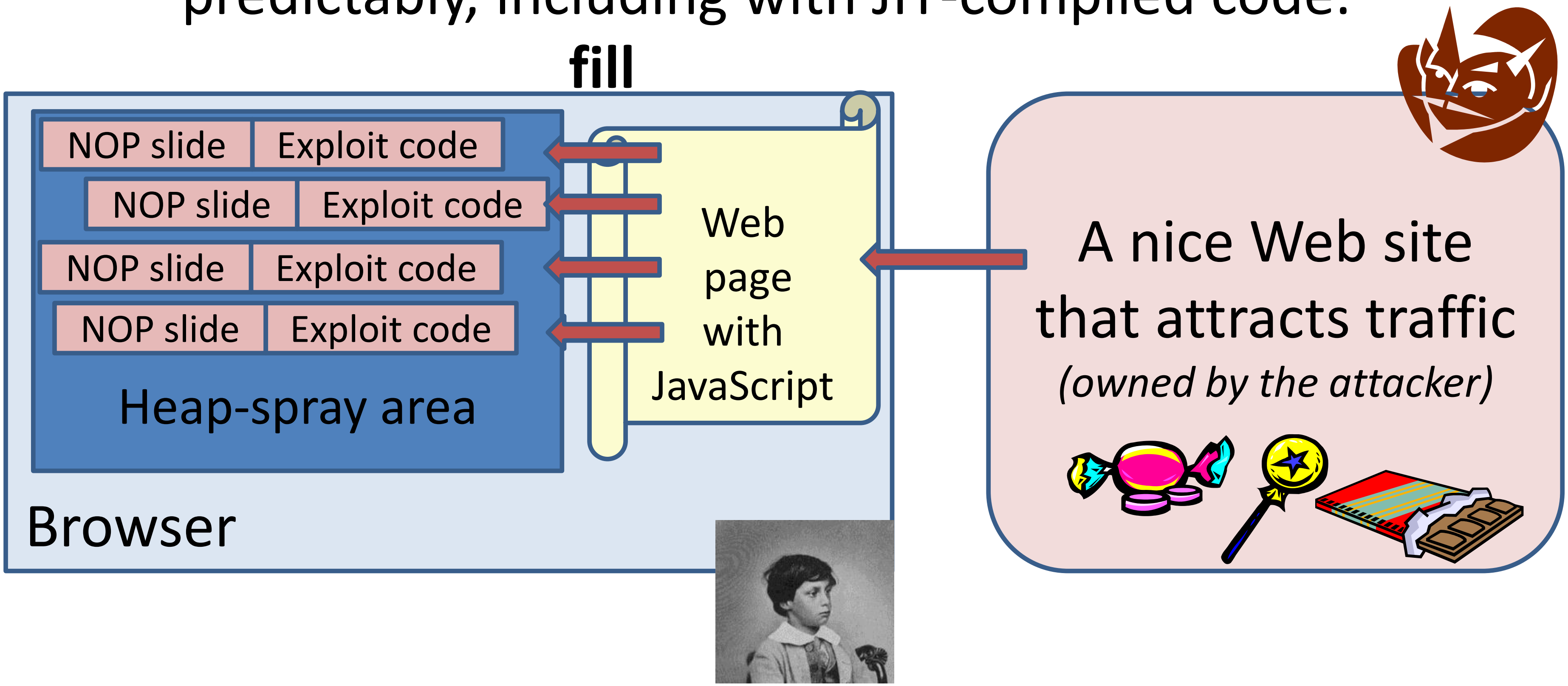
Web page with JavaScript

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

Browser

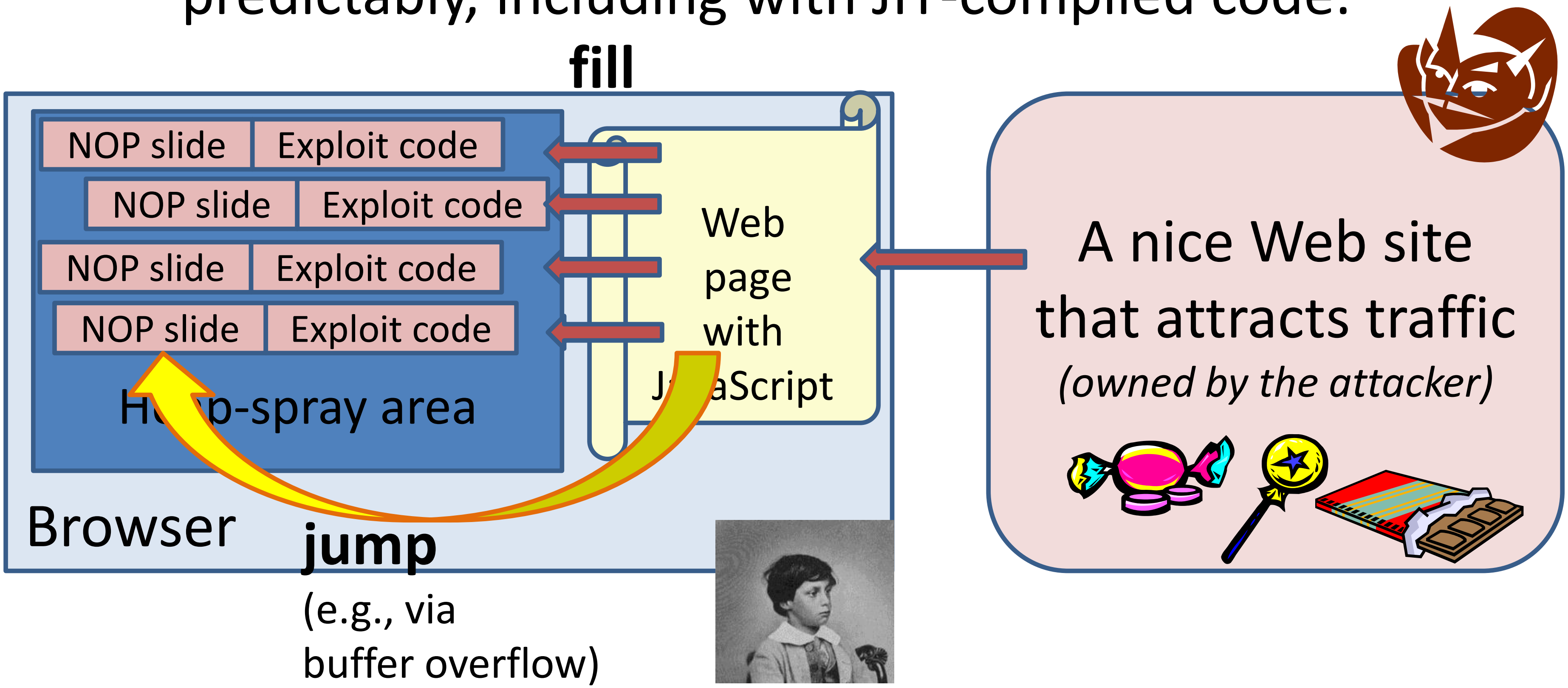
Web page with JavaScript

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

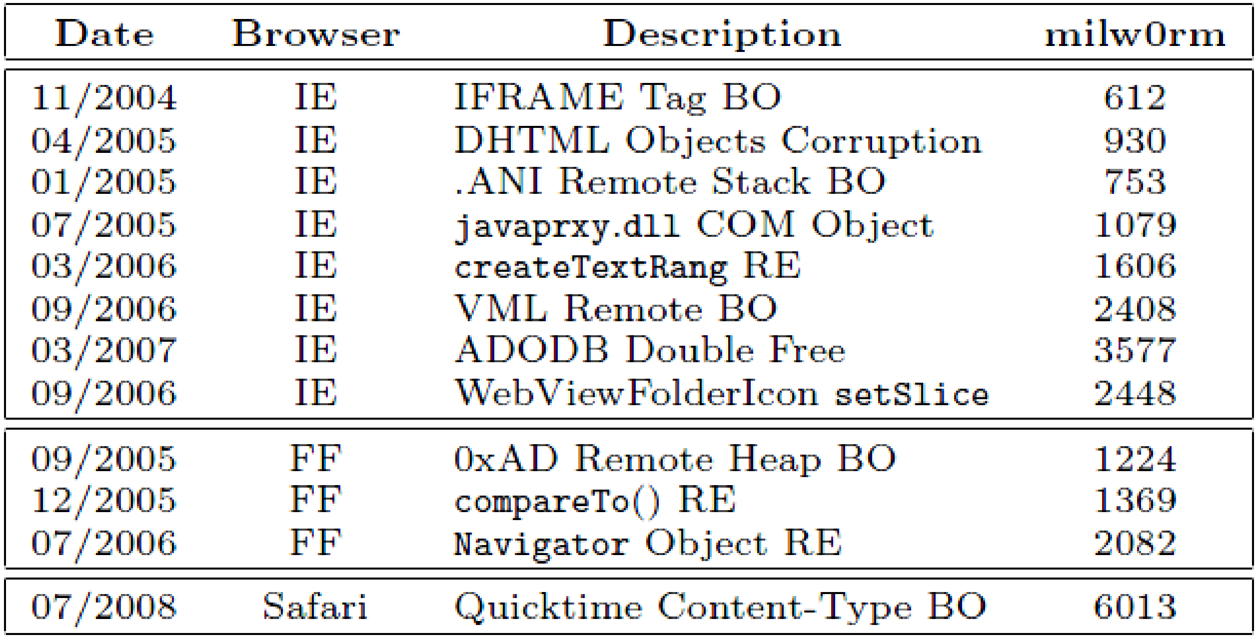
(owned by the attacker)

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…

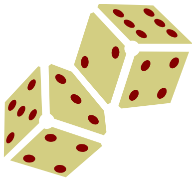
- This secrecy is not always effective.
  - “Heap spraying” can fill parts of the address space predictably, including with JIT-compiled code.
  - “Heap feng shui” influences heap layout [Sotirov].
  - …

Layout randomization: status

This is an active area, with

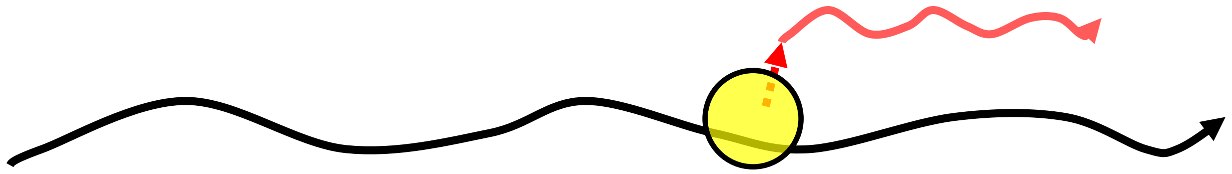
- variants and ongoing improvements to the randomization and its application,
- variants of the attacks,
- techniques detecting or mitigating the attacks.

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



Diverting control flow

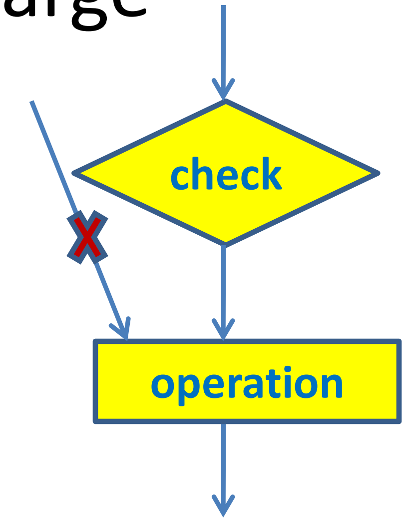
- Many attacks cause some sort of subversion of the expected control flow.



  - 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.

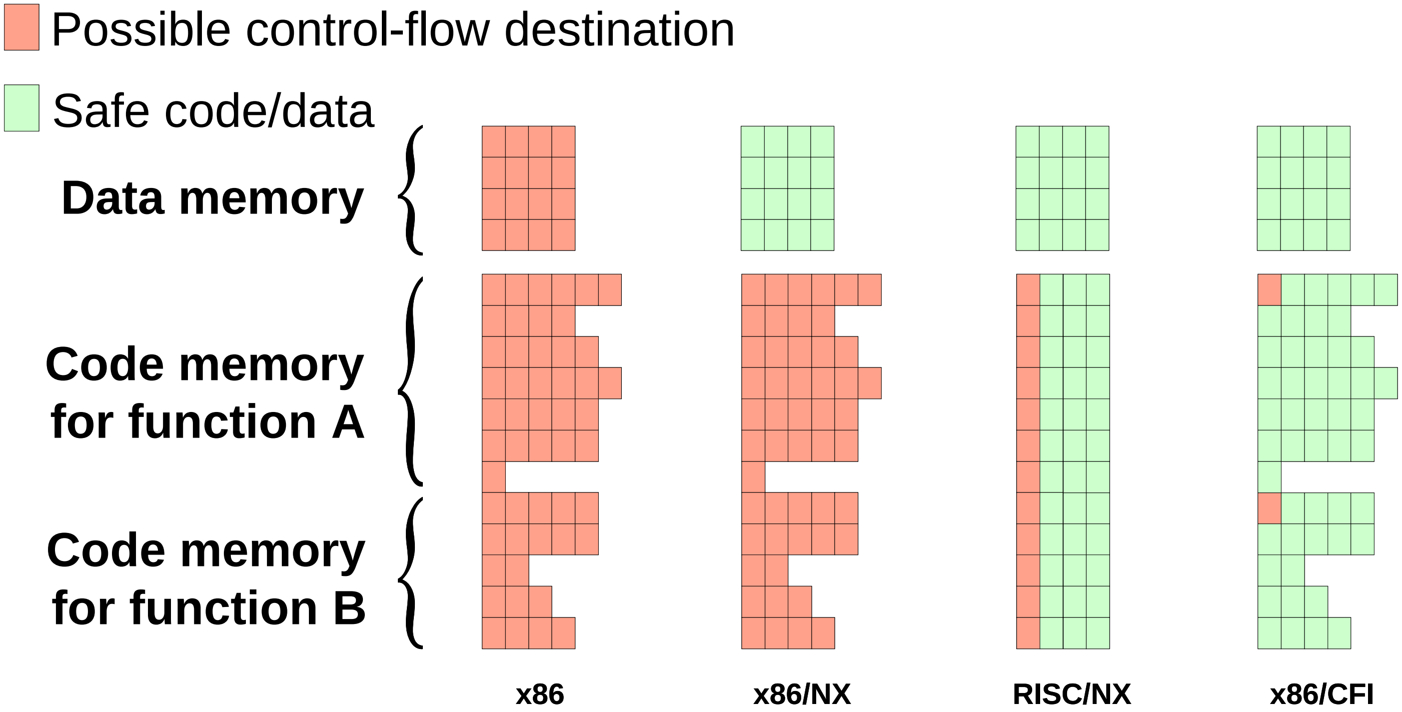
Control-flow integrity (CFI)

- 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.



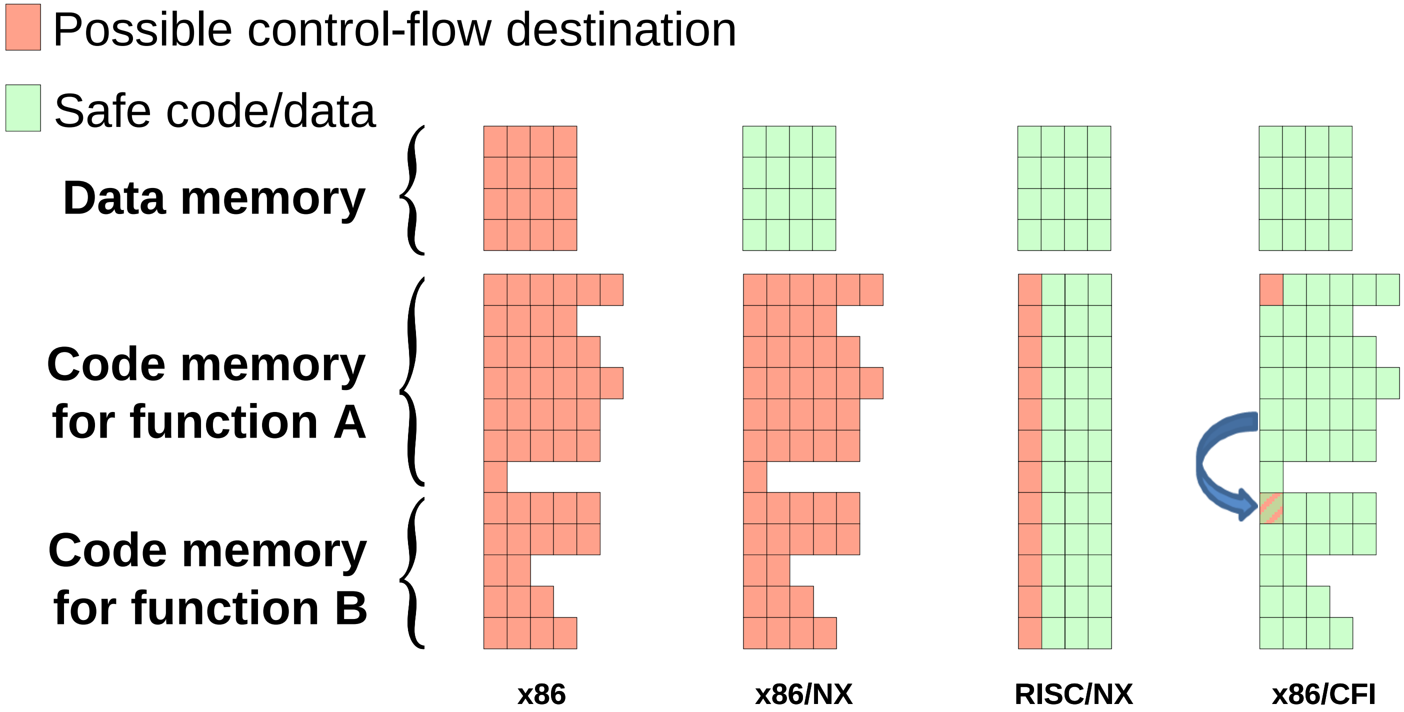
What bytes will the CPU interpret, with CFI?

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



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:

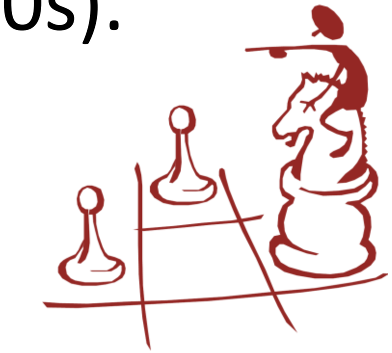


🟥 Possible control-flow destination

🟩 Safe code/data

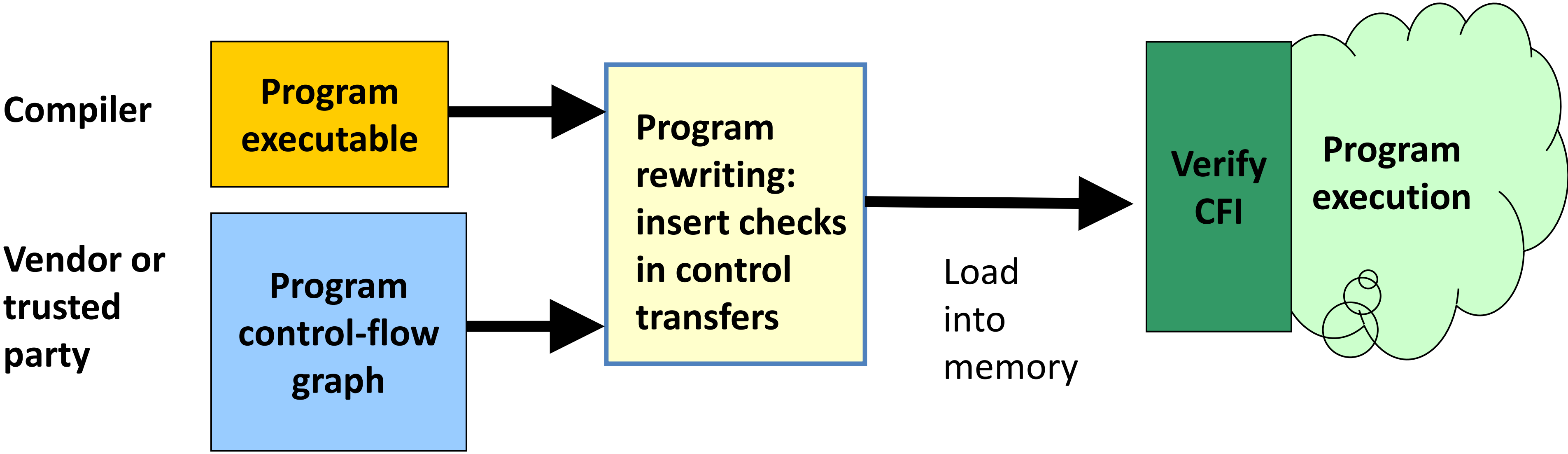
Some implementation strategies for CFI

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]



Compiler

Program executable

Vendor or trusted party

Program control-flow graph

Program rewriting: insert checks in control transfers

Load into memory

Verify CFI

Program execution

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

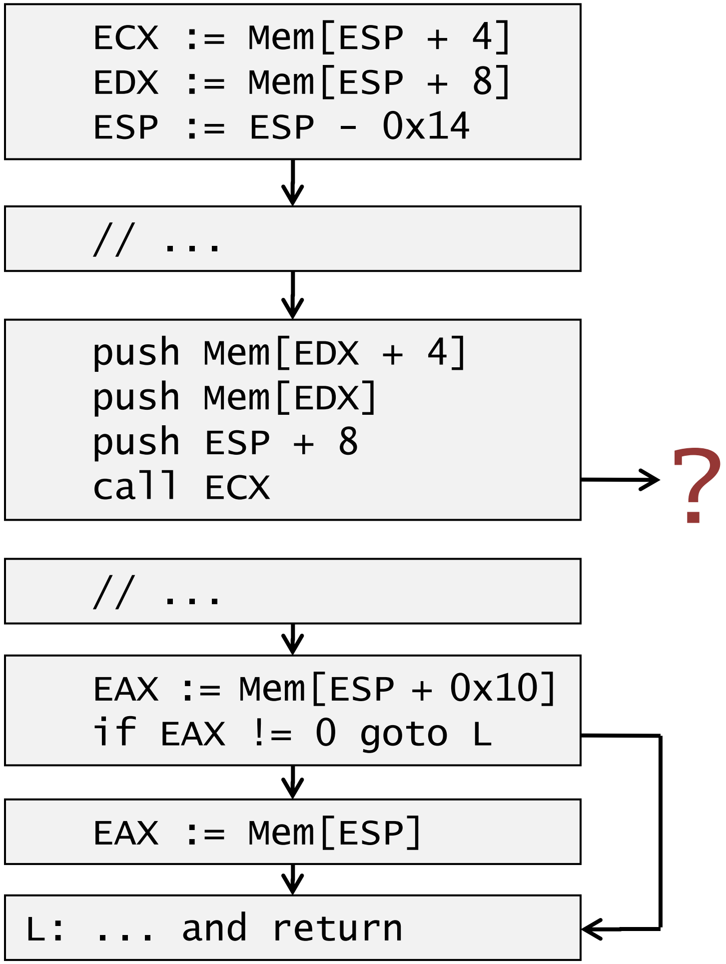
Example

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

C source code

```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
call ECX
```

?

```
// ...
```

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

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

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

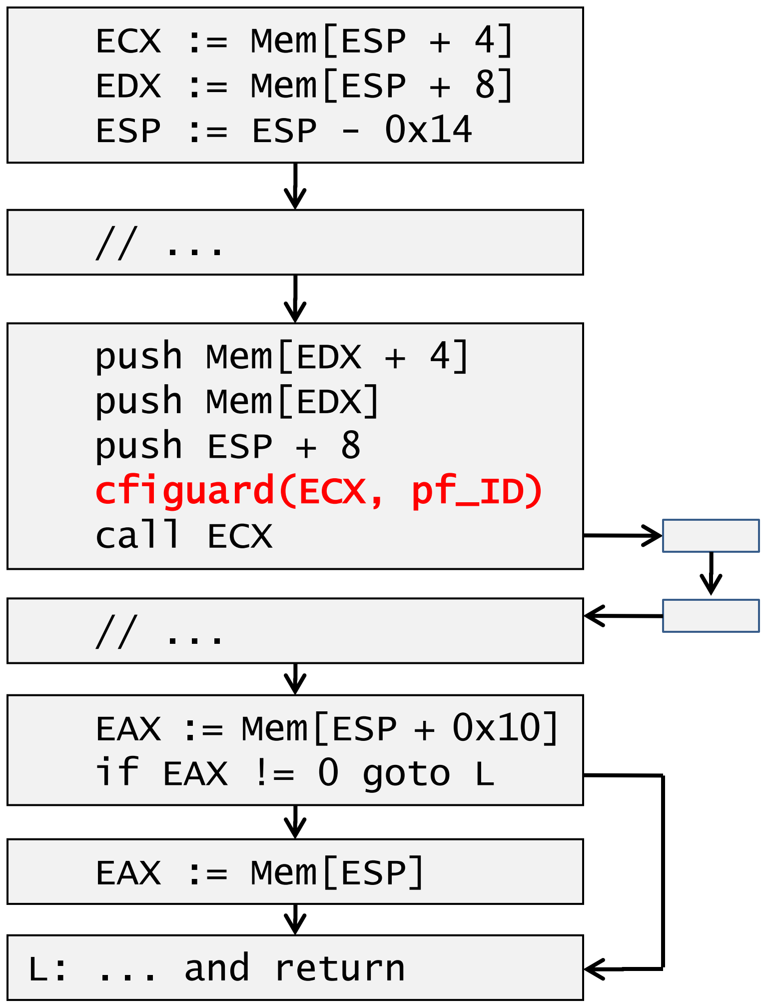
Example (cont.)

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

C source code

```
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
```

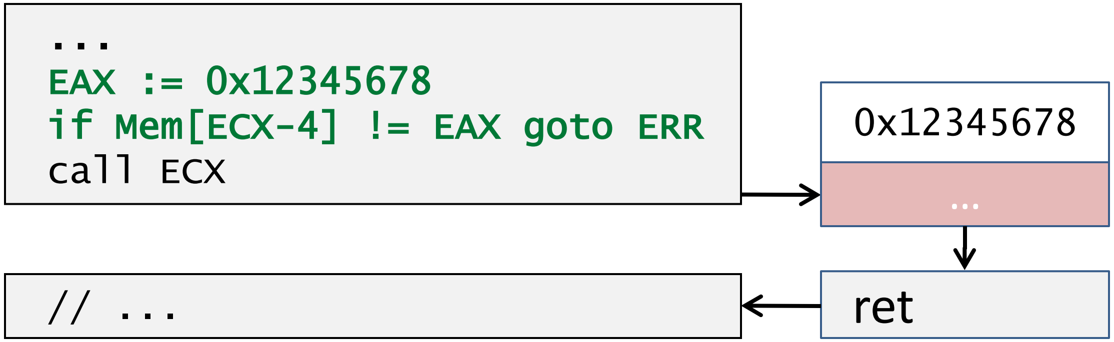
A CFI guard

(a 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



```
...
EAX := 0x12345678
if Mem[ECX-4] != EAX goto ERR
call ECX
```

```
// ...
```

Machine code with 0x12345678 as CFI guard ID

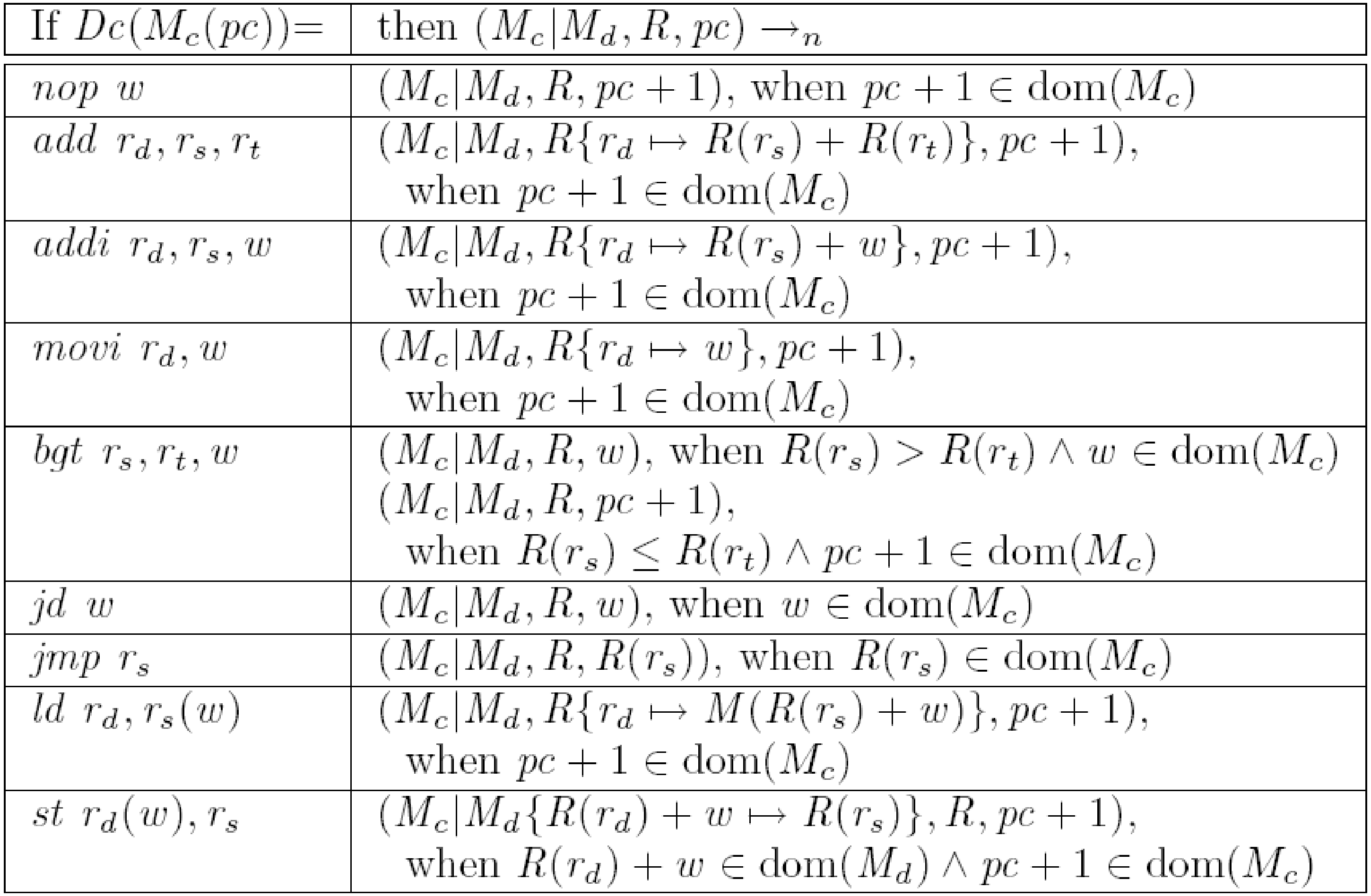
Proving that CFI works

- 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

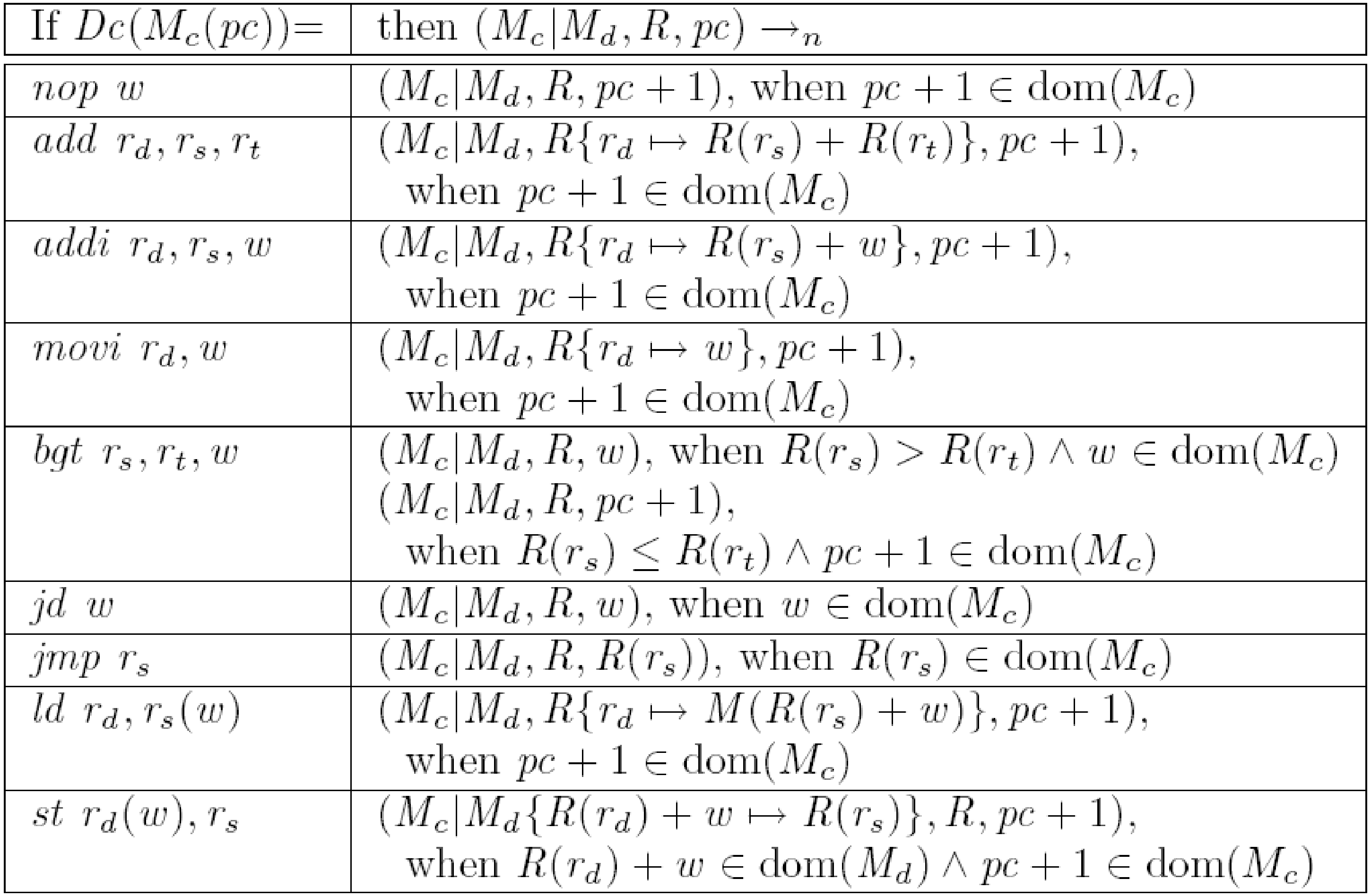
- Instructions: nop, $addi$, $movi$, bgt, jd, jmp, ld, st.
- States: each state is a tuple that includes
  - code memory M_c
  - data memory M_d
  - registers R
  - program counter pc
- Steps: transition relations define the possible state changes of the machine.

1. A small model of a machine



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

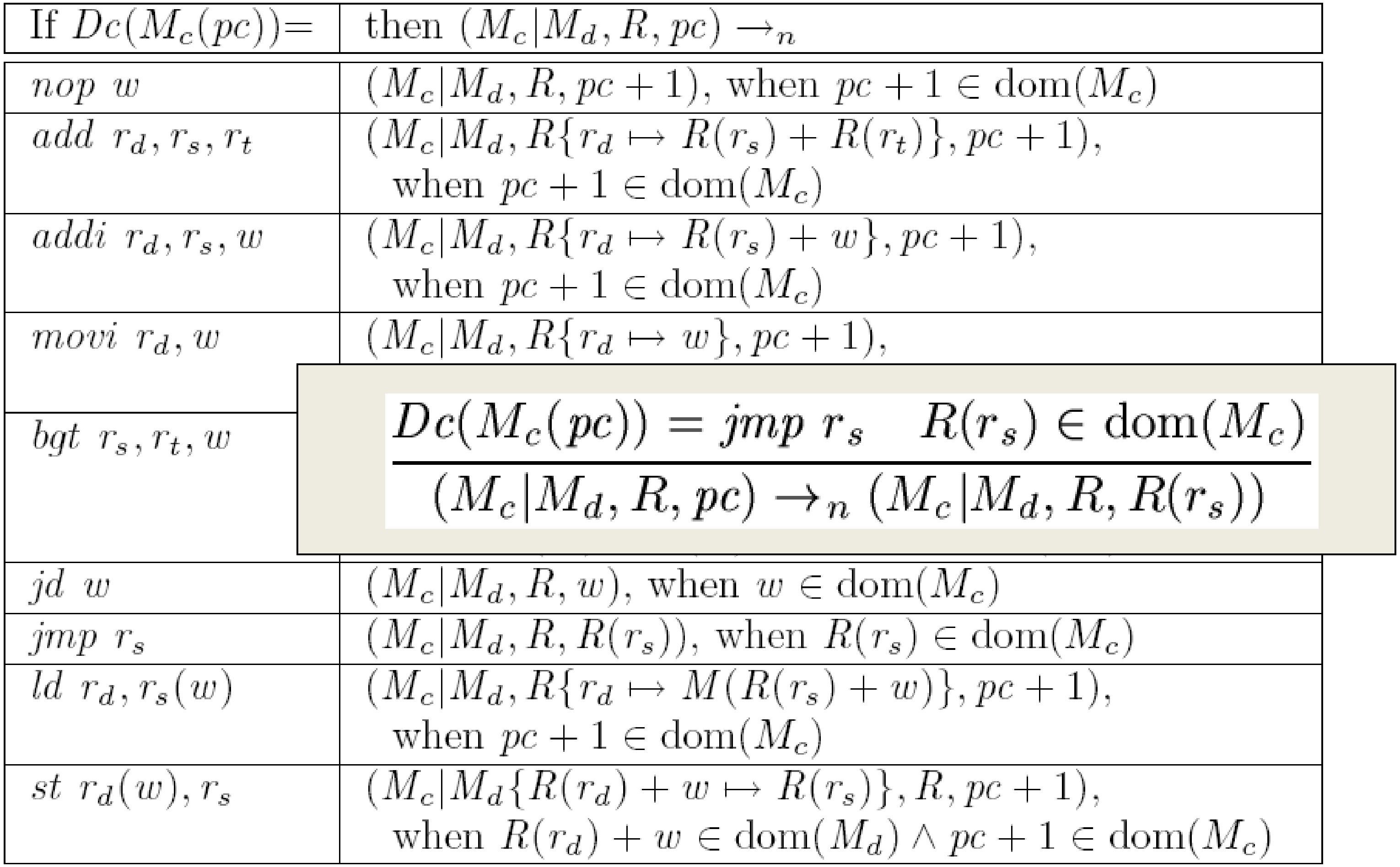
1. A small model of a machine



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

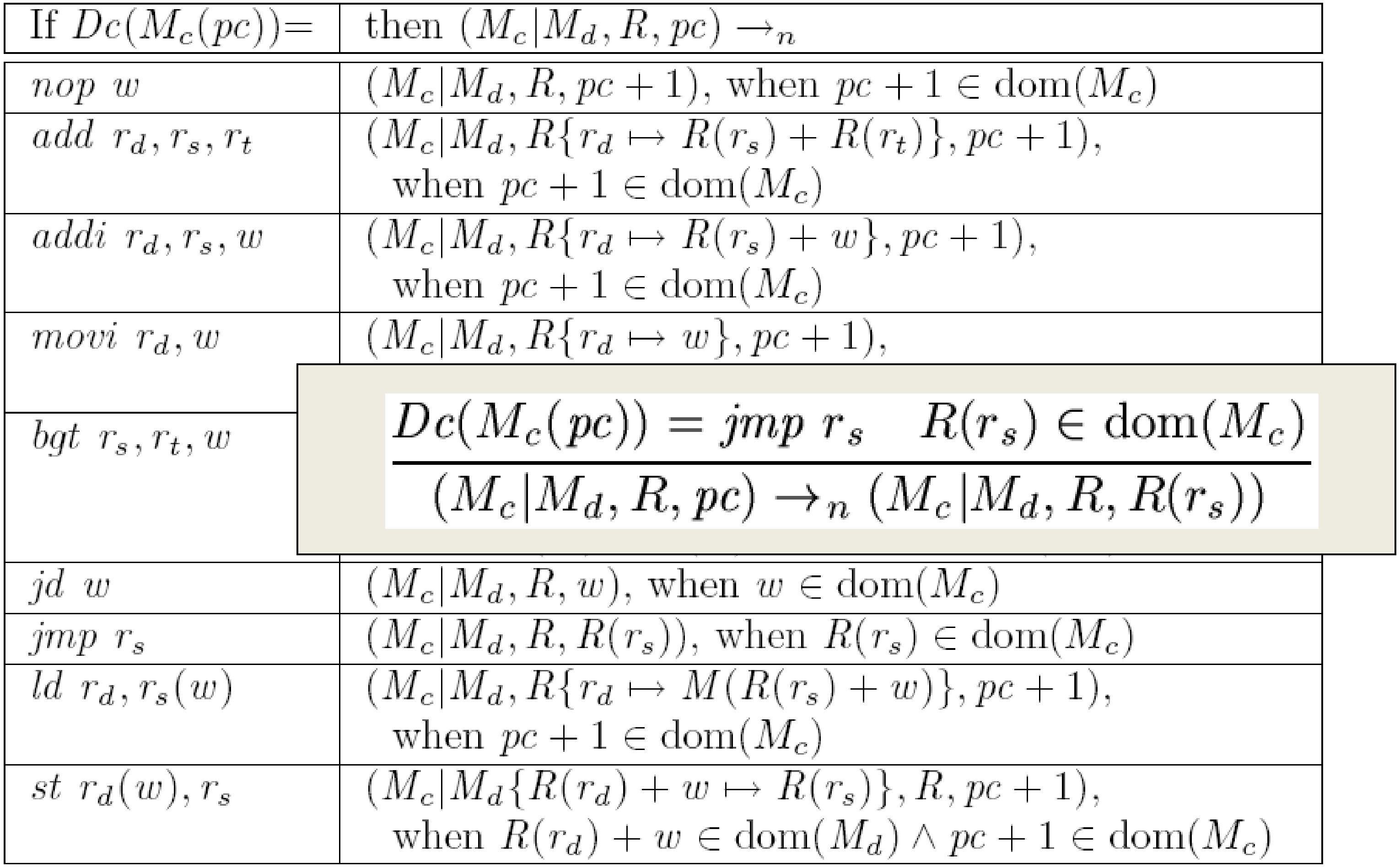
1. A small model of a machine



If $Dc(M_c(pc))=$	then $(M_c \vert M_d, R, pc) \rightarrow_n$
$nop\ w$	$(M_c \vert M_d, R, pc+1)$, 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



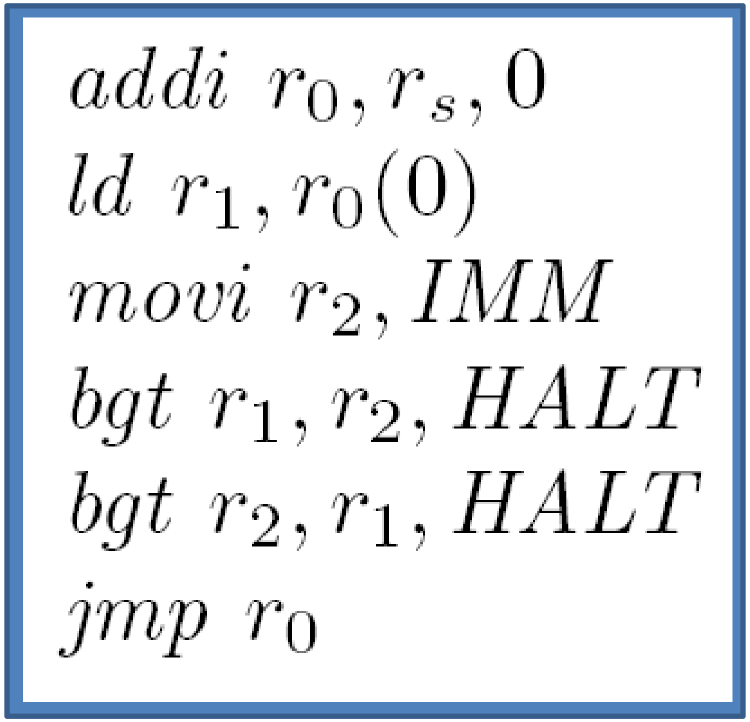
If $Dc(M_c(pc))=$	then $(M_c \mid M_d, R, pc) \rightarrow_n$
$nop\ w$	$(M_c \mid M_d, R, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$add\ r_d, r_s, r_t$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+R(r_t)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$addi\ r_d, r_s, w$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+w\},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

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

+ 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:



$addi\ r_0, r_s, 0$

$ld\ r_1, r_0(0)$

$movi\ r_2, IMM$

$bgt\ r_1, r_2, HALT$

$bgt\ r_2, r_1, HALT$

$jmp\ r_0$

2. Example condition on instrumentation

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

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

HALT is the address of a halt instruction.

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

3. A result

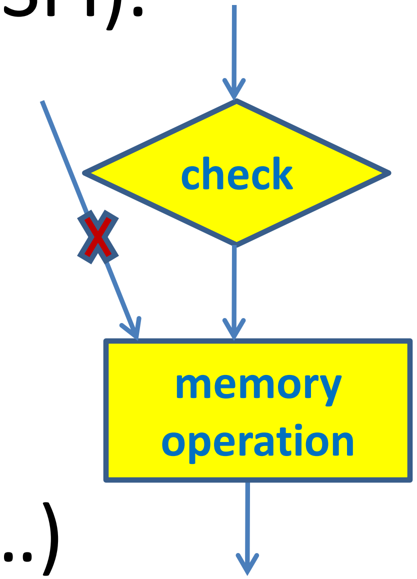
Let S_0 be a state with pc = 0 and code memory M_c that satisfies the instrumentation condition for a given CFG.

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.

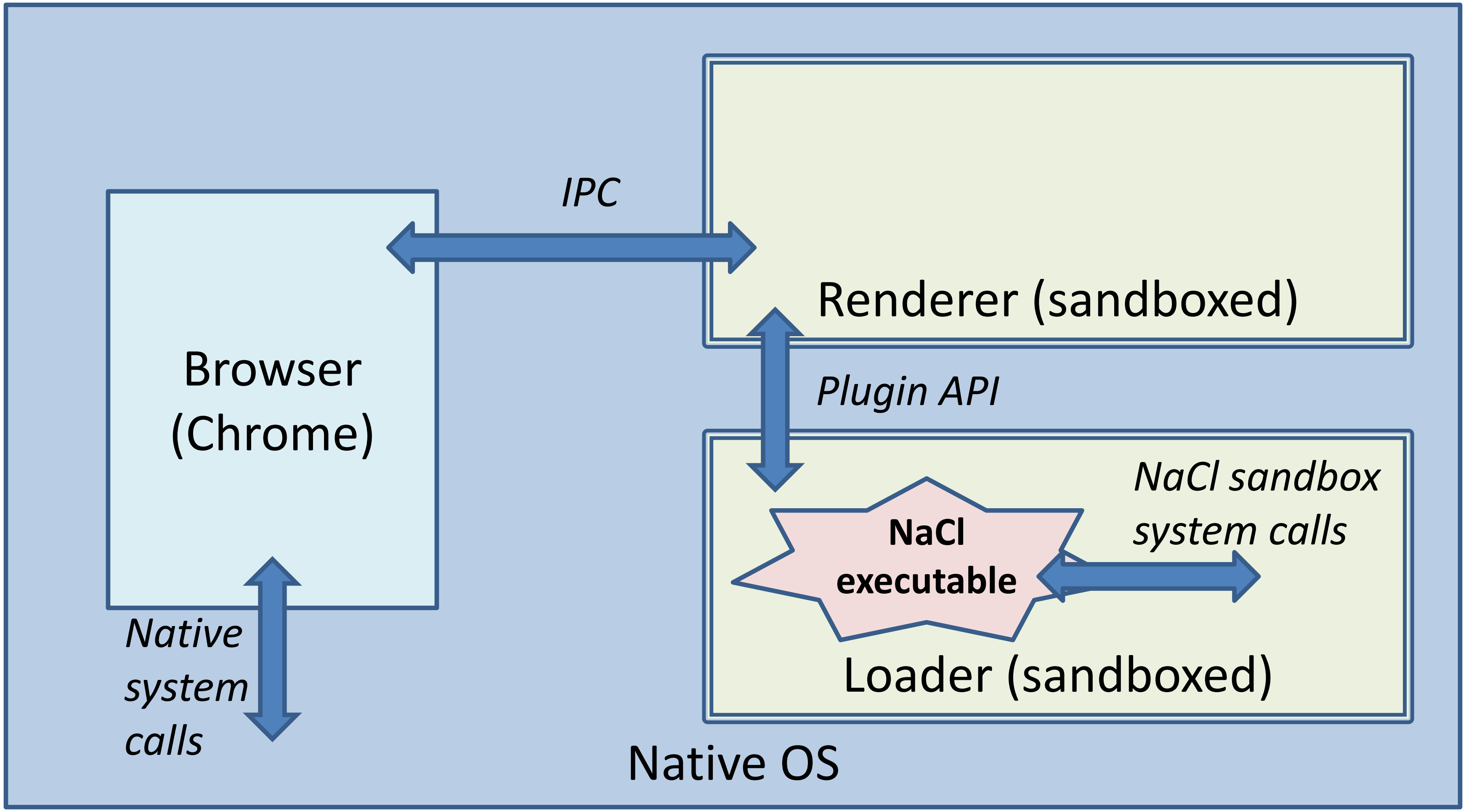
For each i, if $S_i \rightarrow_n S_{i+1}$ then pc at S_{i+1} is one of the allowed successors of pc at S_i according to the CFG.

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.

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.

Reading

- Aleph One’s “Smashing the stack for fun and profit”

  http://www.insecure.org/stf/smashstack.txt
- Pincus & Baker’s “Beyond stack smashing: Recent advances in exploiting buffer overruns”
  http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1324594&tag=1
- Erlingsson’s“Low-level Software Security: Attacks and Defenses”

  http://research.microsoft.com/apps/pubs/default.aspx?id=64363

Homework 4 (due November 8)

Exercise 1:

In MicroIL, are the following two programs well-typed, with respect to any F and S? (yes/no). If so, give one pair of suitable F and S (by defining F_1, F_2, F_3, S_1, S_2, and S_3.)

a) push0 $\cdot$ inc $\cdot$ halt

b) inc $\cdot$ inc $\cdot$ halt

Homework 4

Exercise 2:

Re. Kennedy’s Problem 4, sketch a small example of a function g that illustrates the difficulty being discussed in Section 3 (p9).

Homework 4, cont.

Exercise 3:

Erlingsson’s paper describes six defense techniques (and some variants). Summarize which of them rely on the secrecy of certain information.