Language-based software security, third lecture

Software isolation

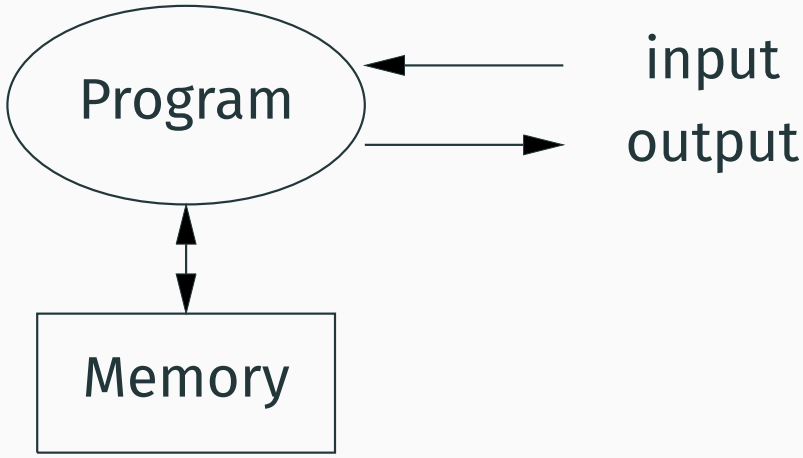
Xavier Leroy

2022-03-24

Collège de France, chair of software sciences

`xavier.leroy@college-de-france.fr`

A simple world...



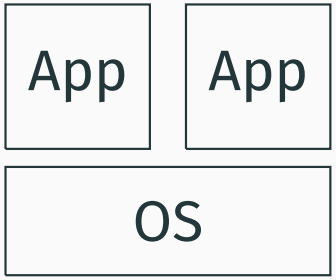
Execution proceeds as described by the program code.

The code is not modified from the outside (integrity).

Data in memory are as written by the program (integrity) and not accessible from the outside (confidentiality).

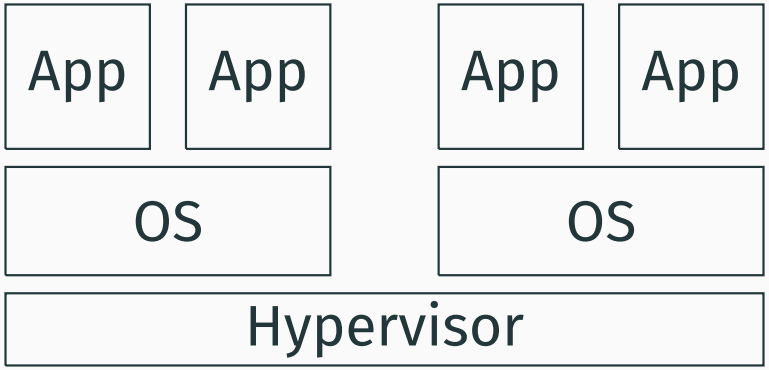
Interactions with the outside world take place only via explicit input/output operations.

The real world...



Time sharing, multiprocessors, multicore processors:
several processes share memory and devices.

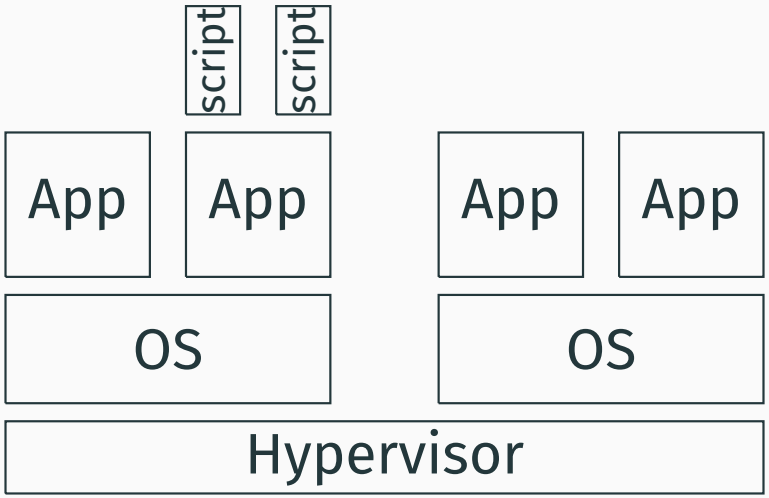
The real world...



Time sharing, multiprocessors, multicore processors: several processes share memory and devices.

Virtualization: several operating system share memory and devices.

The real world…

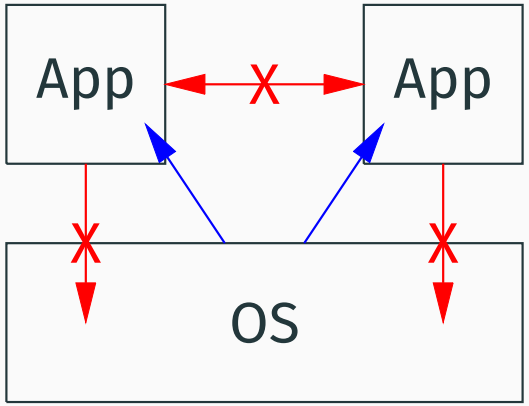


Time sharing, multiprocessors, multicore processors: several processes share memory and devices.

Virtualization: several operating system share memory and devices.

Scripts, macros, extensions, plug-ins, applets: several programs coming from different origins execute within the same process (e.g. the Web browser).

The isolation problem



How to ensure that a program executes without interference from other programs “at the same level” or “above”?

Without interference $\approx$ integrity of the code and its execution;
integrity and confidentiality of data.

Sandboxing vs. Shielded execution

Sandboxing:

An untrusted program is isolated to prevent it from interfering with other programs.

Shielded execution:

A trusted, critical program (OS, cryptographic library, …)
is isolated to prevent interference from other programs.

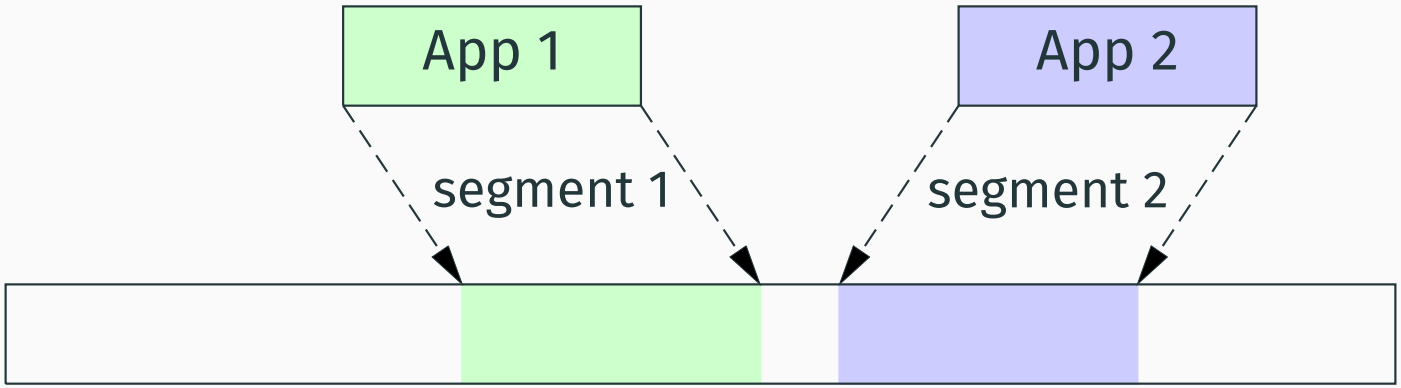
Hardware isolation

Sharing memory between several programs

Time sharing, multiprocessors: memory is shared between multiple programs and the OS.

How can we prevent a program from reading or writing in the data or the code of another program? of the OS?

Segmentation (Burroughs B5000, 1961)



A segment = a pair (base address, size).

Translation logical address (app) $\rightarrow$ physical adress (RAM):

if $0 \leq$ logical address $<$ taille

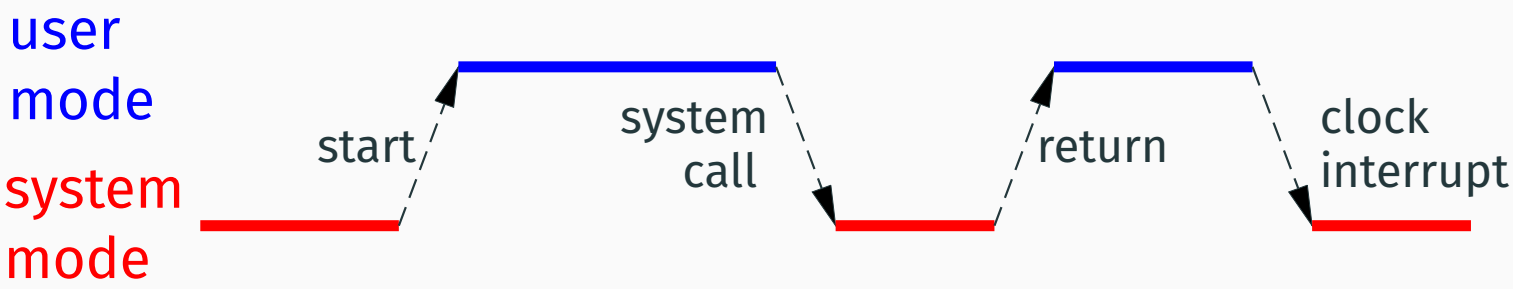
then base adress $+$ logical adress

else segmentation fault

One segment for each process. Segments are pairwise disjoint.

(+ For OS use, one segment that covers all of the RAM).

System mode, user mode

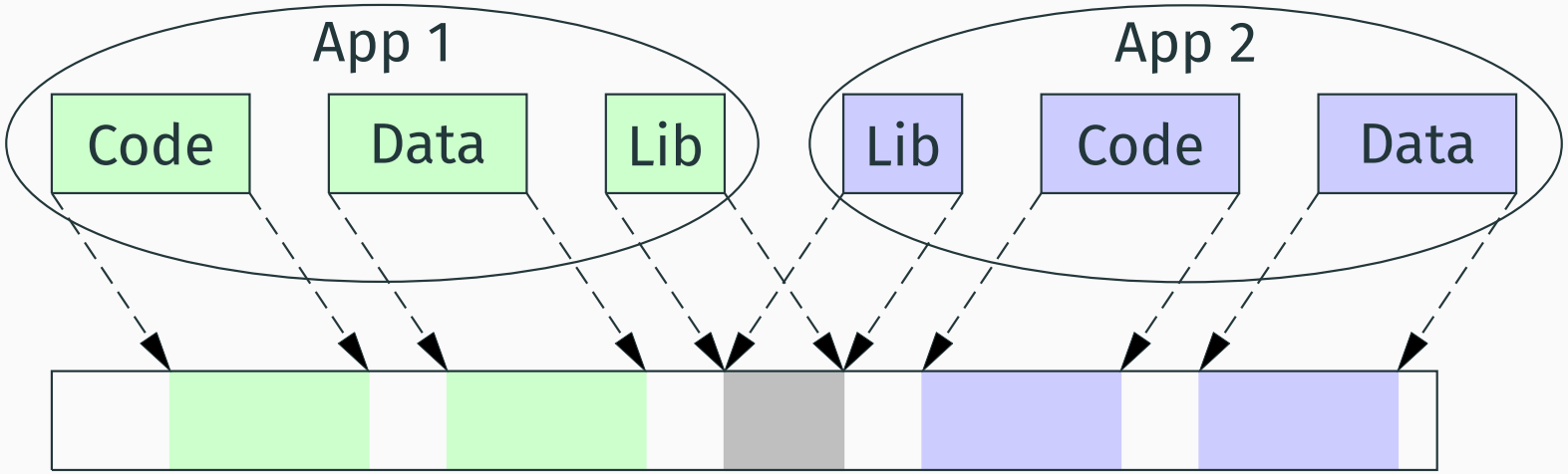


The processor runs in one of two modes: system and user.

Privileged instructions, such as segment register updates, run only in system mode.

Dedicated instructions enable user processes to enter system mode by jumping to a fixed entry point within the system.

Several segments per program (Multics, 1969)

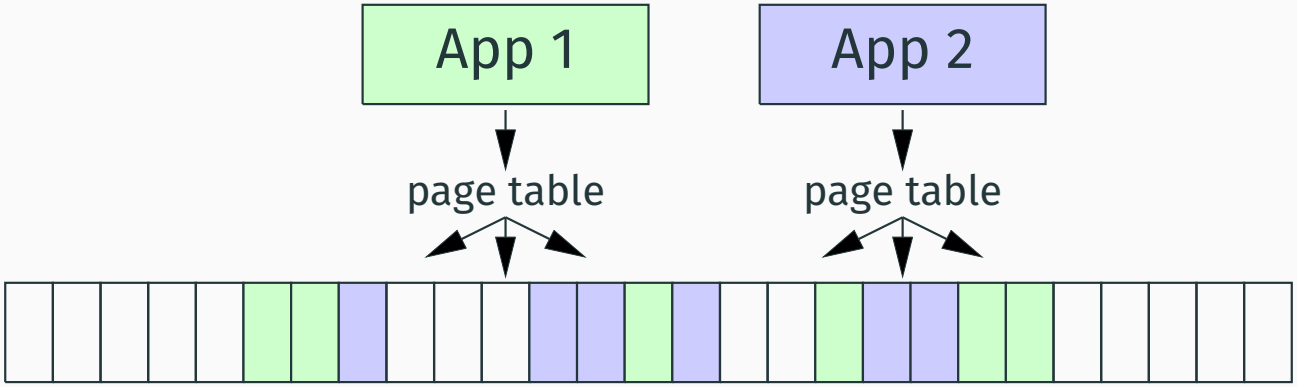


Each process can access several segments: code, data, stack, …

A readonly code segment can be shared between processes
$\rightarrow$ shared libraries.

A writable data segment can be shared between processes
$\rightarrow$ interprocess communication.

Memory paging (IBM S/360-67, 1966)



Memory is divided into pages (typically 4 kilobytes each).

For each process, a page table maps virtual memory pages to physical memory pages.

The pages of a process may not be contiguous in physical memory.

Memory paging

Each page has access rights: read, write, execution.

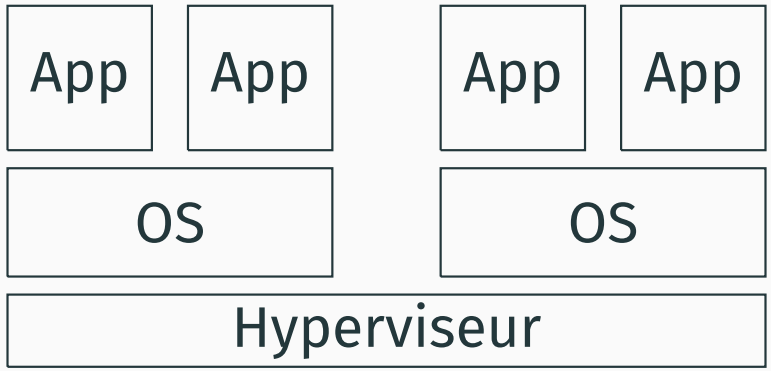
Code pages can be read on demand from an executable file.

Data pages can be *swapped* (written to disk and evicted) when we run out of physical memory.

The instructions that control the page table are privileged.

Complex handling of page tables, involving hardware circuits, microcode, and the OS.

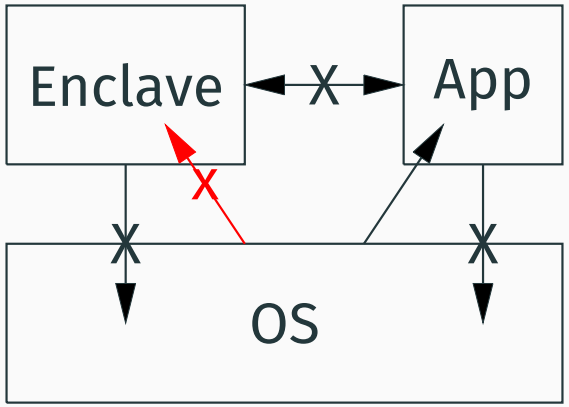
Virtualization (IBM S/370, 1970; *cloud computing*, 2002)



Run an OS not on real hardware, but on a virtual machine emulated by a software system called an hypervisor:

- The OS runs in user mode.
- Its privileged instructions are intercepted by the hypervisor.
- The hypervisor emulates the components of a real machine (page tables, buses, devices, …).

Enclaves (Arm TrustZone, 2004; Intel SGX, 2015)



A process started by / communicating through an OS, but protected against OS compromises using cryptographic hardware mechanisms:

- Encrypted memory pages
- Well-delimited entry points
- Can only run signed executables
- Cryptographic attestation mechanisms.

Summary on hardware and OS isolation

Strengths:

- Guarantees that memory spaces are separated.
- Applications cannot circumvent the access controls implemented by the OS.
- Enables many forms of virtualization and resource control.

Weaknesses:

- High costs for inter-process communications, for starting a process, for starting a virtual machine.
- Inflexible.

Can we think of lighter, more flexible mechanisms?
in particular to isolate scripts within one application?

Software fault isolation

Principle: Inline Reference Monitors

The reference monitor =

the software sub-system that implements access control.

Two typical implementations:

1. The monitor is separate and isolated,
   typically in the kernel of the OS.
2. The monitor is *inlined* in the application code,
   typically during compilation, or by machine code rewriting.

Application: data segmentation

We emulate a segmented architecture by transforming the code for each data read or write.

In pseudocode:

```
x = *p;    →    if (p >= seg.size) abort();
                x = *(seg.base + p);
```

In machine code:

```
                    cmp r1, rsize
load r0, [r1]   →   jae abort
                    add rtmp, rbase, r1
                    load r0, [rtmp]
```

(`rbase`, `rsize`, `rtmp` are reserved registers)

Remplacing bounds checks by masking

In the case of an out-of-bound access, instead of aborting, it is safe to access a random address within the segment.

Typically, we access the address modulo the segment size:

```
x = *p;    →    x = *(seg.base + p % seg.size);
```

If the segment size is a power of 2, this is equivalent to masking with `seg.mask` = `seg.size` − 1

```
x = *p;    →    x = *(seg.base + p & seg.mask);
```

```
                and rtmp, r1, rmask
load r0, [r1]  →  add rtmp, rbase, rtmp
                load r0, [rtmp]
```

When do we rewrite memory accesses?

During compilation from the source language or during machine code generation from an intermediate representation (LLVM, etc).

Example: WebAssembly and its 4 Gb addressing space.

Problem: compilation is part of the Trusted Computing Base.

By rewriting machine code after compilation.

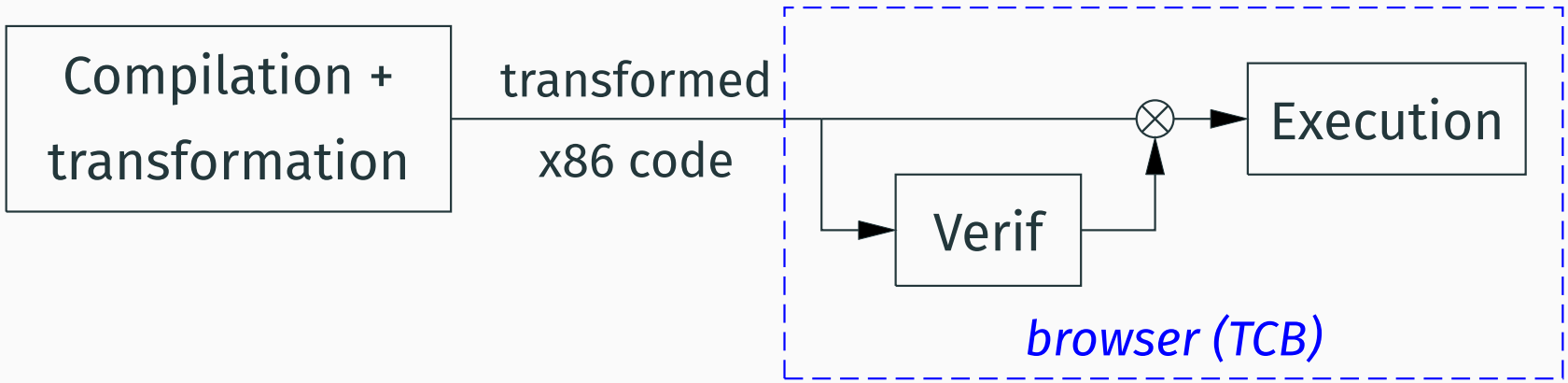
Problem: need to disassemble and reassemble the code;
need to find enough free registers.

Verifying the transformation *a posteriori*

Check the machine code to make sure the translation was applied and all memory accesses are protected:

- disassembly, code analysis
- but no need to rewrite nor to reassemble
- and no need to find free registers.

Example: Native Client (NaCl).



(Yee et al, *Native Client: A sandbox for portable, untrusted x86 native code*, 2009).

Verifying branches

Ensure that execution stays within the transformed code.

(Exception: we can call functions from the host application via “glue” code residing at the beginning of the code segment.)

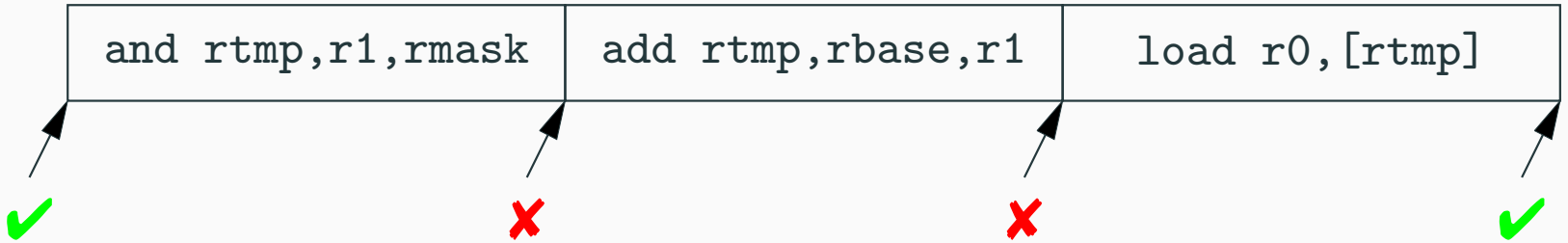
Branches with constant offsets: static verification.

Computed branches, function returns: run-time segmentation.

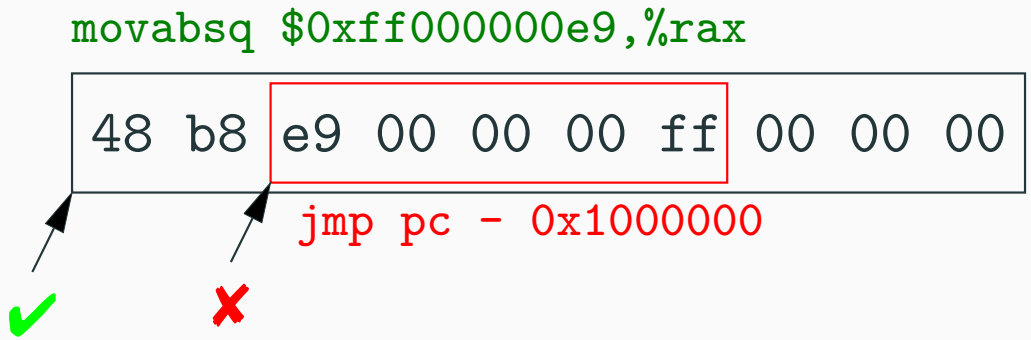
```
                          and rtmp, r0, rcodemask
jump r0          →        add rtmp, rcodebase, rtmp
                          jump r0
```

Verifying branch targets

The target of a branch must not fall in the middle of an instrumented sequence:

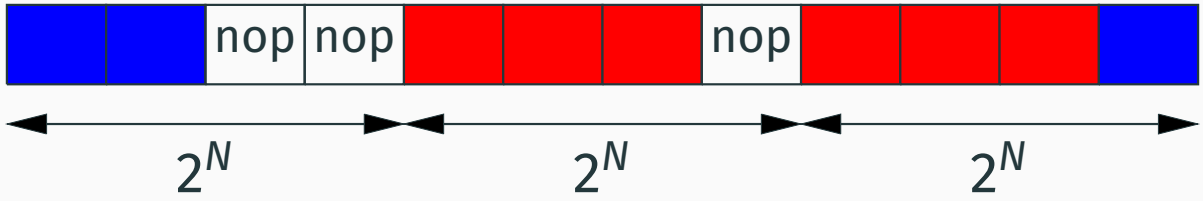


The target of a branch must not fall in the middle of a processor instruction:



Verifying branch targets

Check that instructions (in blue) and instruction sequences produced by the transformation (in red) are fully contained in blocks of 2^N bytes.



(Can always be achieved by adding `nop` during code generation.)

Set the N low bits of `rcodemask` to 0 so that all computed branches go to multiple of 2^N.

```
                          and rtmp, r0, rcodemask
jump r0          →        add rtmp, rcodebase, rtmp
                          jump r0
```

Summary on software fault isolation

An interesting example of isolation by code transformation, requiring no hardware support.

Main use: sandboxing of complex libraries in sensitive applications (the Firefox browser).

Decent performance: $\approx$ +20% in running time
($\approx$ +10% if only memory writes are protected but not reads).

Security is hard to guarantee $\rightarrow$ formal verification of the verifier.

(Morrisset et al, *Rocksalt: better, faster, stronger SFI for the x86*, 2012).

Language-based isolation

Language-based isolation

An approach that emerged in the 1990's in response to the needs of mobile code:

- untrusted pieces of code,
- downloaded over the Internet,
- automatically executed in a browser, a mail reader, a word processor, etc.

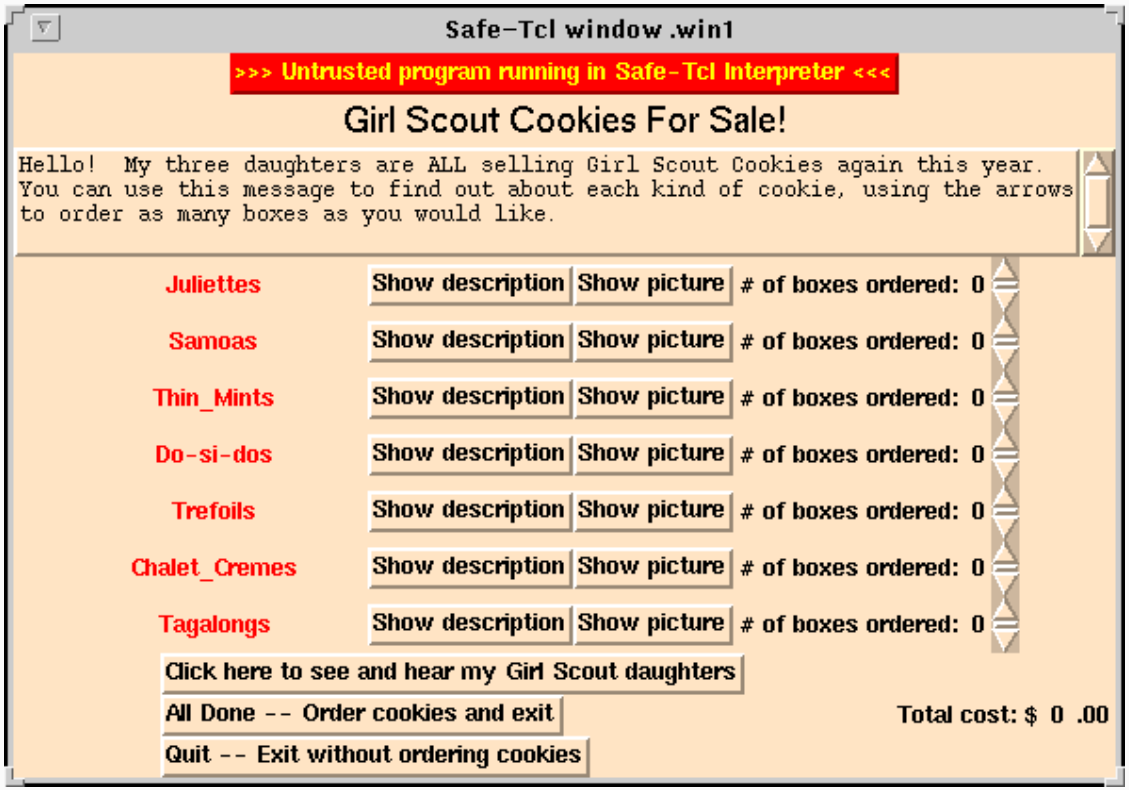
Two components to this approach:

- a safe programming language and execution environment (strong typing, either dynamically or statically enforced);
- restricted software interface (API), limiting the interactions between mobile code and execution environment.

A precursor: *Email with a mind of its own*

(N. Borenstein, *EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail*, 1991, 1997)

E-mail attachments that are programs (scripts): interactive form, animated greetings card, etc.



The Safe-TCL language and API

The TCL scripting language with a modified API:

- Interaction only via the TK graphical toolkit or the terminal.
- No access to files nor to system commands.
- Exception: can send e-mail or print a document, after interactive confirmation by the user.

Java applets (1995)

Java programs compiled to JVM code, downloaded on the Web, and executed in the browser.

Java applets (1995)

Efficient execution thanks to JVM bytecode verification
($\approx$ static type-checking) followed by interpretation with few run-time checks.

Same API for applets and for local applications.
(*Write once, run anywhere.*)

Access control integrated in the API:
permissions are determined by the code origin (local / applet) and the cryptographic signatures it may carry.

JavaScript (1995)

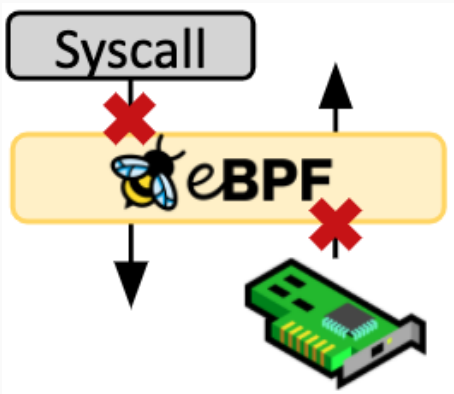
Initially: yet another scripting language, dynamically typed, interpreted, to make Web pages more interactive.

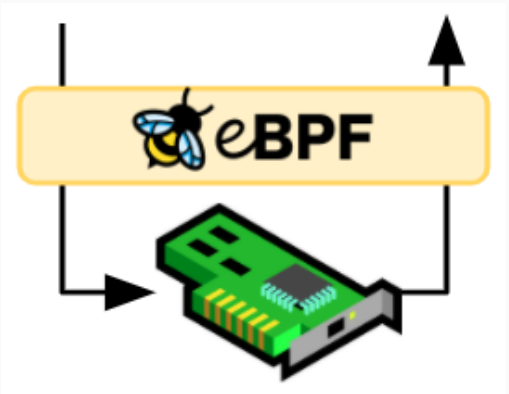
Since 2004: the implementation language for “AJAX” applications running in the browser.

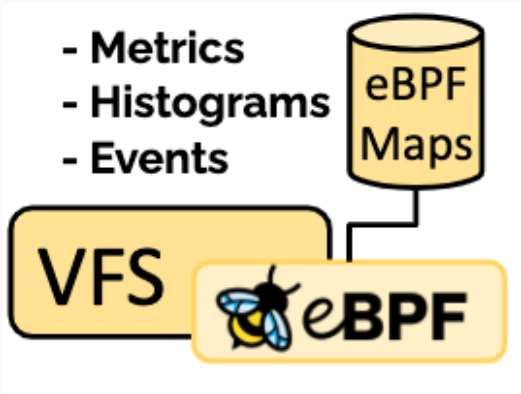
Today: one of the most widely used languages,
with excellent JIT compilers.

Security model: restricted API + same-origin policy.

Berkeley Packet Filter (BPF, 1992; eBPF, 2014)







Code for a register-based virtual machine that can be injected in OS kernels (BSD, Linux) for network filtering, OS monitoring, etc.

Strongly controlled access to the data structures of the kernel.

Safe execution guaranteed by static analysis of the code: value analysis, memory access safety, termination.

Safe execution

Two main approaches:

1. defensive execution (run-time tests) [JavaScript, TCL]
2. static verification (before execution) [Java, BPF]
   + “offensive” execution (fewer run-time tests)

Orthogonal to interpreted / compiled.

Orthogonal to the format of mobile code:
source code or compiled code for a virtual machine.

Securing an API

Restrict the functionalities available to mobile code:

- Expose only functions that are not dangerous.
- Hide things using visibility modifiers
  (in Java: `private`, `protected`, package-local).
- Hide things using type abstraction. ($\rightarrow$ 5th lecture)

Add access control to API functions.

- Example: Java's `SecurityManager`.

Java API security

The Java capability model

Each method is associated with a set of capabilities, also called permissions:

Files: read, write, execute, delete
(depending on the file path)

Network: opening an outgoing connection, accepting an incoming connection
(depending on the name and port of the other host)

Runtime: stop the program, load native code, define a new class loader, define a new security manager, ...

GUI: accessing the clipboard, the event queue, ...

(And much more).

Associating capabilities with methods

The capabilities are set by the class loader that loaded the code of the method prior to execution. Typically:

- Code loaded from local files: all permissions.
- Code loaded from the Web: no access to files or to the runtime; network connections only with the host where the code comes from.
- Additional capabilities can be granted to codes that carry a trusted cryptographic signature.

Access control

Before performing a risky operation, API methods call the `checkPermission` method (or its variants `checkFile`, `checkAccept`, etc) from the `SecurityManager` class.

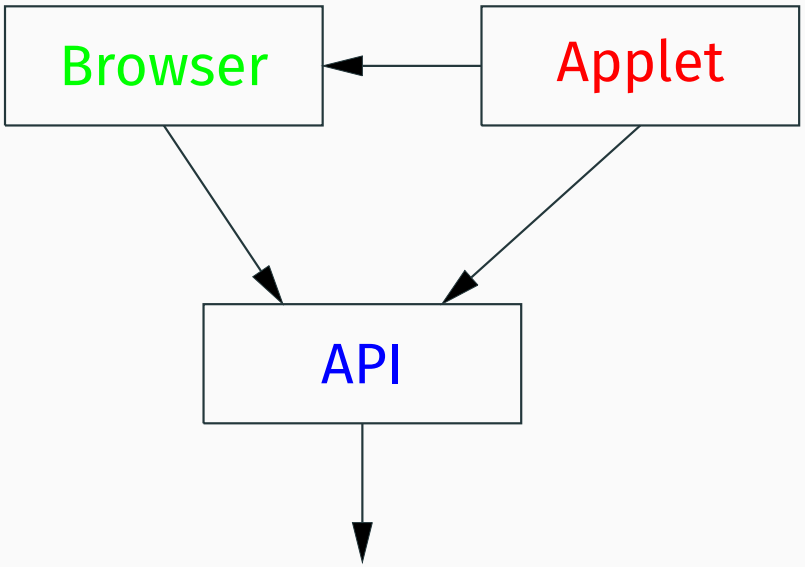
If all calling methods have the requested permission, `checkPermission` returns normally and access is granted.

If one of the calling methods lacks the requested permission, `checkPermission` raises an exception and access is denied.

Stack inspection

This permission check is called (call) stack inspection.

It enables an API to behave differently depending on whether it acts on behalf of trusted code (e.g. the browser) or of untrusted code (an applet).



Browser $\rightarrow$ API:
access granted.

Applet $\rightarrow$ API:
access denied.

Applet $\rightarrow$ Browser $\rightarrow$ API:
access denied.

Privilege amplification

An API may need to perform a dangerous operation regardless of the context of the call.

Example: to display text in a GUI, we may need to load a font from a local file, even if we're called from an applet.

Privilege amplification

The font loading code can use `doPrivileged` to execute with its own permissions, even if it is called from less privileged code.

```
void loadFontFile(String name) {
    // validation of the parameter
    AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() {
            // open and read the font file
            return null;
        }
    });
}
```

The access control algorithm

For each pending method call in the call stack,
from most recent to least recent:

> Let *M* be the method being executed in this call.
>
> Check if *M* has the requested permission.
> If not, throw `SecurityException`
>
> Check if *M* amplified its privileges.
> If so, access is granted.

Check if current thread inherited the requested permission.
If not, throw `SecurityException`
If so, access is granted.

The “confused deputy” problem

An untrusted principal coaxes a trusted principal into performing a sensitive operation on its behalf.

Stack inspection protects the callee from the caller.
But sometimes the caller is fooled by the callee…

Example: (Abadi & Fournet, 2003)

```
class Trusted {
  // all permissions
  static void main() {
    String s =
      Applet.tempfile();
    File.delete(s);
  }
}
```

```
class Applet {
  // no permissions
  static String tempfile() {
    return "/etc/passwd";
  }
}
```

The “confused deputy” problem

Another example based on subclassing:

(Abadi & Fournet, 2003)

```
class Trusted {
  // all permissions
  String tempfile = "/tmp/file";
  abstract void proceed();
  static void main() {
    File.create(tempfile);
    try {
      proceed();
    } finally {
      File.delete(tempfile);
    }
  }
}
```

```
class Applet extends Trusted {
  // no permissions
  void proceed() {
    tempfile = "/etc/passwd";
  }
}
```

From the call stack to execution history

(M. Abadi & C. Fournet, *Access Control based on Execution History*, 2003.)

Idea: base access control not on the methods currently being called, but on all the methods that have been called so far.

The machine state is extended with C = the current permissions.

Method call $\quad obj.m(\ldots)$

$$C := !C \cap \texttt{permissions}(obj.m)$$

Privilege amplification $\quad \texttt{grant}(P)\ \{\ B\ \}$

$$\texttt{let}\ \ C_0 = !C\ \ \texttt{in}\ \ C := !C \cup P;\ B;\ C := !C \cap C_0$$

Access control policies based on execution history

Knowing the execution history enables finer security policies such as

> *An applet can read a local file or open a network connection, but not both.*

Opening a file removes the “network” permission.
Opening a network connection removes the “files” permission.

Summary on the Java API

An interesting idea: access control based on the calling context.

An imperfect mechanism: stack inspection.

Alternatives to stack inspection, not used in practice.

No API for interacting between the applet and the Web document...

JavaScript API security

The JavaScript security model

Dynamically-typed language + restricted APIs:

- DOM: interaction with the document (displayed Web page)
- XmlHttpRequest: direct connection to HTTP servers
- Cookies, Web Storage: limited local data storage
- etc.

Protects rather well the browser and the OS from malicious pages and scripts.

Little isolation between scripts and between scripts and pages.

Same-origin policy (SOP)

The origin of a page or an `iframe` =

the (protocol, host, port) part of its URL.

A script or a DOM element has the same origin as the containing page.

Same-origin policy: a script can only

- inspect or modify a DOM element with the same origin;
- open an XmlHttpRequest connection to the originating host (other servers can be authorized via the CORS protocol);
- access Web cookies with same domain and same path.

SOP does not prevent CSRF (*Cross-Site Request Forgery*)

Hyperlinks `<a>`, HTML forms, included elements `<img>`, `<script>` are not restricted by the same-origin policy.

A malicious script can insert in the document an image loaded from an arbitrary address:

```
<img width="0" height="0"
     src="https://vulnerable.service.com/email/change
          ?newmail=attacker@evil.net">
```

If the user has an active session on `vulnerable.service.com`, the authentication cookies are transmitted and the request to change the e-mail address is honored.

Some other SOP vulnerabilities

Hosts are identified by name, not by IP addresses
$\rightarrow$ can be attacked by DNS manipulations.

A script can change the "host" part of the page origin
(but only to a super-domain).

```
                                    // Before: test.example.com
document.domain = "example.com";
                                    // After: example.com
```

If the attacker can access to the test site `test.example.com`,
they can now access the production site `example.com` ...

JavaScript: a very dynamic language

A function can be redefined after the fact:

```
f = function(arg) { /* new body */ };
```

The methods of an object can be changed after creation:

```
obj.meth = function(arg) { /* new body */ };
```

Likewise for the prototype of an object and its methods:

```
obj.prototype = new MyClass();
Object.prototype.__defineSetter__('x', myfunc);
```

This enables attacks by one script on another.

This can also be used to add protections to the API or a script!

Adding security protections to API functions

(Phung, Sands, Chudnov, *Lightweight Self-Protecting JavaScript*, 2009).

Idea: early in the page, load a script that redefines API functions to add access controls.

```
<html>
  <head> <script src="./policy.js"></script> </head>
  <body>
    <!-- page contents, unchanged -->
  </body>
</html>
```

Redefining API functions

Example: prevent `windows.open` from opening more than 5 pop-up windows.

```
function () {
    var orig = windows.open;
    var num_windows = 0;
    var wrapped = function () {
        if (++num_windows >= 5) abort();
        orig();
    }
    windows.open = wrapped;
} ();
```

Note: `orig` and `num_windows` have scope the block of the anonymous function; they are not accessible anywhere else.

Redefinition in aspect-oriented programming style

```
var wrap = function(pointcut, Policy) {
  // Override the prototype of the object if available
  var source = (typeof(pointcut.target.prototype) != undefined)
               ? pointcut.target.prototype : pointcut.target;
  var method = pointcut.method;
  // Save reference to the original method
  var original = source[method];
  // Weave the policy with the original method
  var aspect = function() {
    var invocation = {object:this, args:arguments};
    return Policy.apply(invocation.object,
        [{arguments:invocation.args, method:method,
          proceed:function(){
            return original.apply(invocation.object, invocation.args);}}]);
  }
  // Redefine the method
  source[method] = aspect;
  return aspect;
}
```

Vulnerabilities in the redefinition

(Magazinius, Phung, Sands, *Safe wrappers and sane policies for self protecting JavaScript*, 2012.)

```
return Policy.apply(invocation.object,
    [{arguments:invocation.args, method:method,
      proceed:function(){
        return original.apply(invocation.object, invocation.args);}}]);
```

The attacker can recover the `original` non-secured method, e.g. by redefining `Function.apply` or the *setter* for `proceed`.

```
var recover_builtin;
Object.prototype.__defineSetter__('proceed',
            function(o) { recover_builtin = o });
```

(See Magazinius et al for possible counter-measures.)

Protecting a script from the rest of the page

(K. Bhargavan, A. Delignat-Lavaud, S. Maffeis, *Defensive JavaScript: building and verifying secure Web components*, 2014.)

```
<html><body>
   <script src="attack1.js"></script>
   <script src="sensitive.js"></script>
   <script src="attack2.js"></script>
</body></html>
```

Context: a script implements a sensitive operation (end-to-end encryption, signature, interaction with a password manager, etc). It can coexist with malicious scripts in the same Web page.

Example: authenticated messaging

```
var f = function(msg) {
    var key = "...";
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "https://logging.example.com", false);
    xhr.send(Crypto.HMAC(key, msg) + "," + msg);
}
```

The `key` signature key is included in the script.

It cannot be read by other scripts in the same page because the scope of `key` is local to the `api` function.

However `f.toSource()` returns the text of the function (type `string`), from which attackers can extract the key!

Hiding the source of the function

```
var f = (function () {
    var g = function(msg) {
        var key = "...";
        // send the message
    }
    return function(msg){ return g(msg); }
}) ();
```

Now, `f.toSource()` reveals only the wrapper `function(msg){return g(msg);}` but not function `g`.

Calls to external functions

```
...
var key = "...";
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://logging.example.com", false);
xhr.send(Crypto.HMAC(key, msg) + "," + msg);
...
```

The attacking script can redefine `Crypto.HMAC` to leak the key:

```
var hmac = Crypto.HMAC; var leaked_key;
Crypto.HMAC =
  function (k,m) { leaked_key = k; return hmac(k,m); };
```

$\rightarrow$ we must reimplement all the crypto locally in function g.

Calls to external functions

```
...
var key = "...";
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://logging.example.com", false);
xhr.send(Crypto.HMAC(key, msg) + "," + msg);
...
```

The attacking script can redefine the `open` method of `XMLHttpRequest` to inspect the call stack and recover the code of function `g`:

```
stackwalk = function() {
  var apisource = stackwalk.caller.toSource(); ...
}
```

A protected code

Separate the function that contains the keys and does the crypto from the function that communicates.

```
var f_internal = (function (){
    var hmac = function(key,msg){ /* reimplementation of HMAC */ }
    var g = function(msg){
        var key = "...";
        return (hmac(key,msg) + "," + msg);
    }
    return function(msg){return g(msg);}
})();
var f = function (msg) {
    var mac = f_internal(msg);
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "https://logging.example.com", false);
    xhr.send(mac);
}
```

Defensive JavaScript

(K. Bhargavan, A. Delignat-Lavaud, S. Maffeis, *op. cit.*)

A subset of JavaScript, verified by static type-checking:

- Strict lexical scoping.
- Strict static typing.

  $\rightarrow$ Objects and arrays are initialized at creation-time.
- No implicit coercions (`toString`), no getters/setters.
- Memory heap separation: no objects are shared between defensive code and outside world.

  $\rightarrow$ Entry points in defensive code use only base types

  e.g. `string` $\rightarrow$ `string`.

Summary of JavaScript security

Not designed for security.

Ineffective same-origin policy.

One language feature (static scoping) gives isolation guarantees; many other features destroy isolation.

Capability machines

Two converging visions

Fine-grained segmentation:

- Each array, record, etc, has its own memory segment.
- A reference = a segment; a pointer = a pair (segment, offset).
- Automatic bounds checking at every access
  $\rightarrow$ no more *buffer overflow* problems!

Capabilities for accessing memory:

- Reference / pointer = right to access a memory area.
- No access (read, write, execute) possible
  unless the program has the corresponding capability.

Abstract view of a memory capability

$$\text{capability} = \langle \overbrace{\text{permissions}}^{\text{R, W, X, etc}}, \text{base}, \text{size}, \text{offset} \rangle$$

Gives access rights to the adresses $[\text{base}, \text{base} + \text{size}[$

Reading a 32-bit word:

$$\texttt{load32}(\langle p, b, t, d \rangle) =$$
$$\quad \text{if } \mathtt{R} \in p \wedge d + 4 \leq t$$
$$\quad \text{then read 4 bytes at address } b + d$$
$$\quad \text{else error}$$

Abstract view of a memory capability

Pointer arithmetic:

$$\texttt{offset}(\langle p, b, t, d\rangle,\ \delta) =$$
$$\quad \text{if } \{\texttt{R}, \texttt{W}, \texttt{X}\} \cap p \neq \emptyset \wedge 0 \leq d + \delta \leq t$$
$$\quad \text{then } \langle p, b, t, d + \delta\rangle \text{ else error}$$

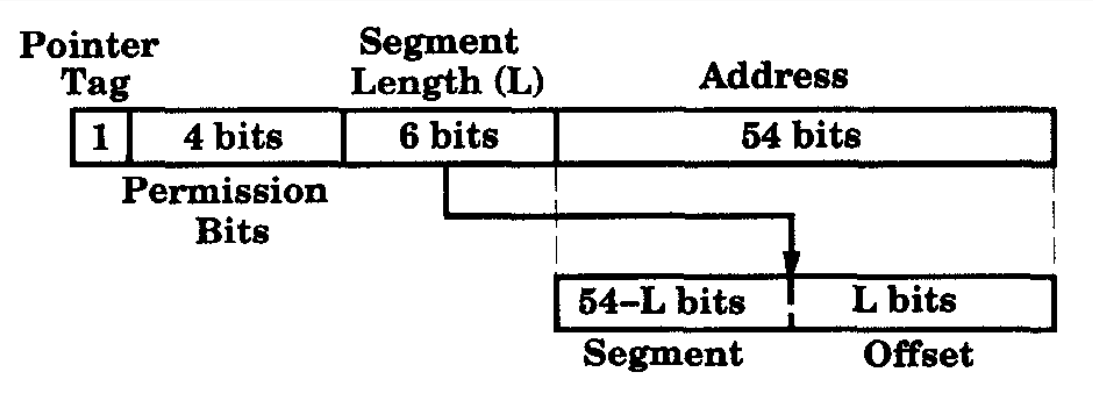
Restricting the size / the permissions of a capability:

$$\texttt{subseg}(\langle p, b, t, d\rangle,\ p',\ t'\ ) =$$
$$\quad \text{if } p' \subseteq p \wedge d + t' \leq t$$
$$\quad \text{then } \langle p', b + d, t', 0\rangle \text{ else error}$$

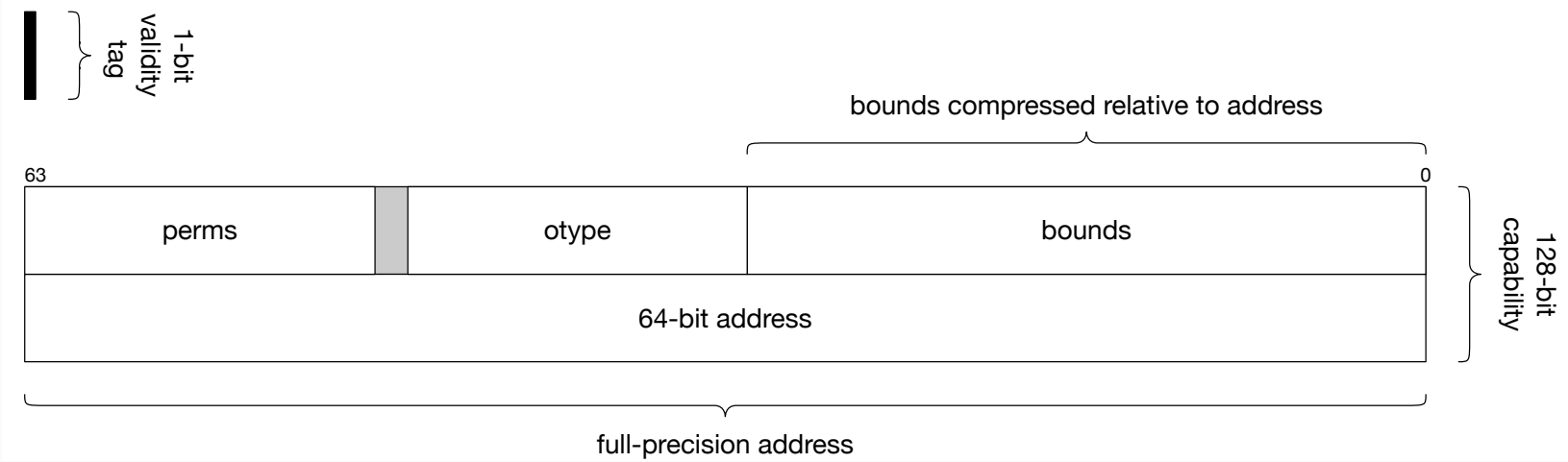
Decrease property: in user mode, a program cannot construct capabilities that are stronger than the capabilities given in the initial registers and memory state.

Concrete representation of a memory capability

Guarded pointers (Carter, Keckler, Dally, 1994):



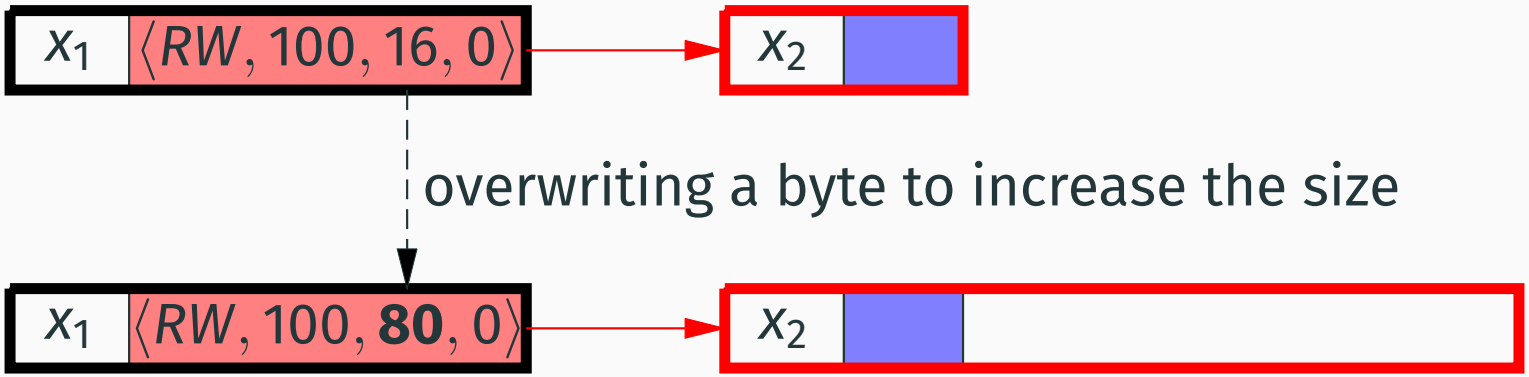
The CHERI architecture (U. Cambridge, 2016):



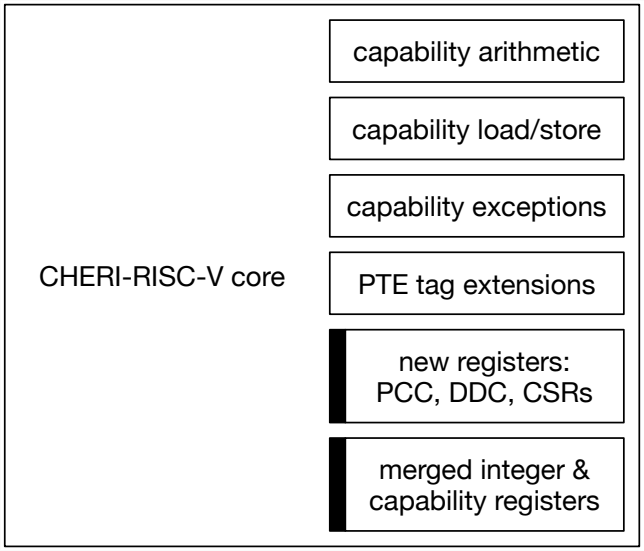
Ensuring the integrity of capabilities

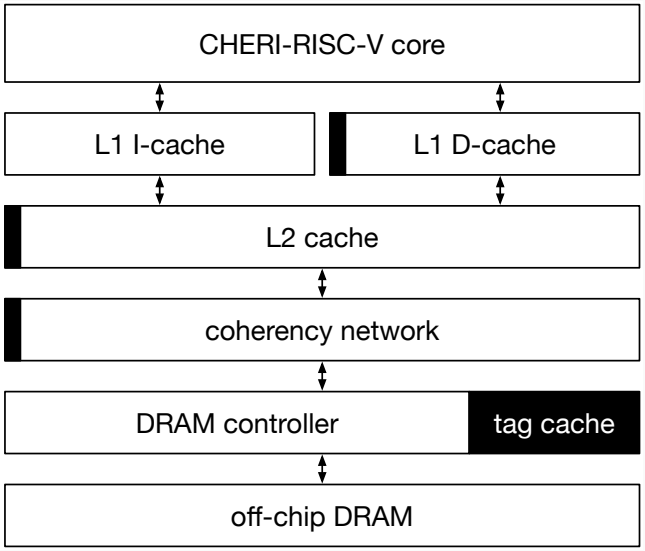
Just like pointers, capabilities must be stored in memory, e.g. to build linked lists:

What prevents the program to forge a capability by overwriting bytes in memory?



A tag for valid capabilities





Add one bit (65th or 129th bit) to every register and every memory word. This tag bit says whether a capability is valid.

- Tag set to 1 by the instructions that produce capabilities (`offset`, `subseg`, etc).
- Tag set to 0 by all other instructions (arithmetic, logical, memory writes to a sub-word, etc).

Protecting data with capabilities

Each variable, each dynamically-allocated block (`malloc`) can be placed in "its" memory area, disjoint from any other, with automatic bounds checking at every access.

```
int f() {
    char b[80];                    // b = ⟨RW, 1000, 80, 0⟩
    int ok = 0;                    // &ok = ⟨RW, 1080, 4, 0⟩
    char * p = malloc(1024);       // p = ⟨RW, 4000, 1024, 0⟩
    gets(b);
    ...
}
```

In particular, no assignment `b[i] = ...` can modify `ok`, even if the corresponding memory areas are adjacent

$\rightarrow$ the call to `gets(b)` is safe!

Programming one's memory allocator

```
#define HEAPSIZE 1000000
static char heap[HEAPSIZE];
static int next = 0;

void * malloc(size_t sz) {
    if (next + sz >= HEAPSIZE) return NULL;
    void * p = subseg(heap + next, RW, sz);
    next += sz;
    return p;
}
```

The pointer returned by `malloc` only gives permission to access elements `next`, ..., `next` + `sz` − 1 of array `heap`.

To make the allocated block inaccessible other than via this pointer, it suffices to hide the `heap` array from clients of `malloc`.

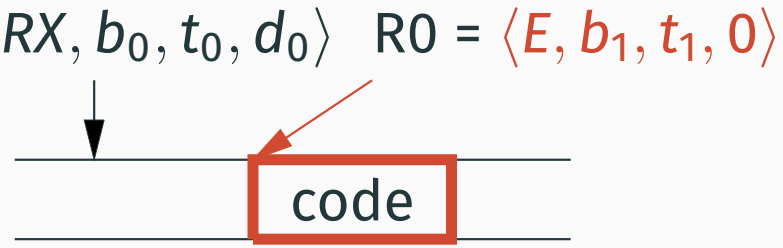
Protecting code with capabilities

Using `subseg`, we can seal a code pointer, turning its access rights from `RX` to `E` (as in `Enter`).

A sealed `E` pointer gives no access rights to the memory area. It only enables a `CALL` instruction to jump to the corresponding code address.

The code being called is granted `RX` permissions on the memory area.

Before `CALL R0`:

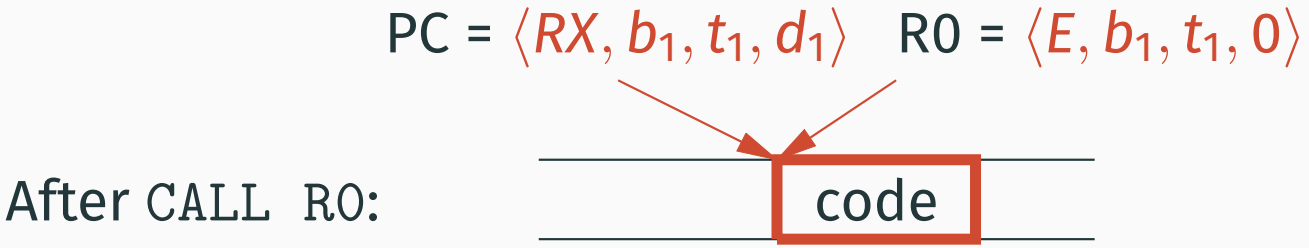


Protecting code with capabilities

Using `subseg`, we can seal a code pointer, turning its access rights from `RX` to `E` (as in `Enter`).

A sealed `E` pointer gives no access rights to the memory area. It only enables a `CALL` instruction to jump to the corresponding code address.

The code being called is granted `RX` permissions on the memory area.

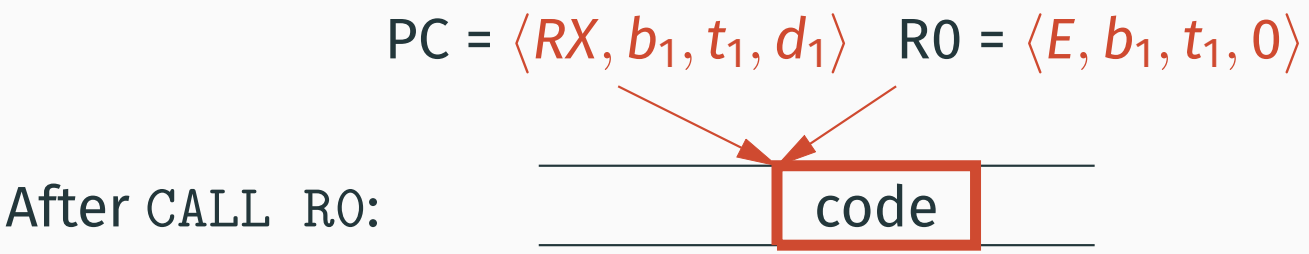


Protecting code with capabilities

Using `subseg`, we can seal a code pointer, turning its access rights from `RX` to `E` (as in `Enter`).

A sealed `E` pointer gives no access rights to the memory area. It only enables a `CALL` instruction to jump to the corresponding code address.

The code being called is granted `RX` permissions on the memory area.

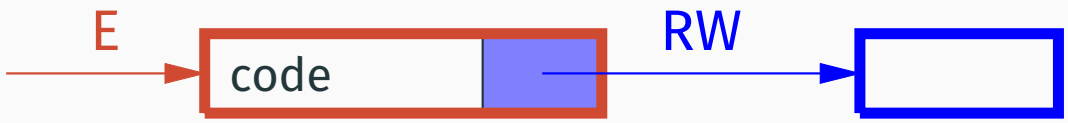


No pointer arithmetic on `E` capabilities

$\rightarrow$ no jumping “in the middle” of the code of a function!

Making data private to a piece of code

The sealed code area can contain a `RW` capability to a data area. If this data area is not visible to the caller, it is private to the sealed code.



Making data private to a piece of code

Example: a monotonically increasing counter (in pseudo-asm).

```
    // Code area RX                    // Data area RW
next: r1 = load(data)                    counter: .word 0
      r0 = load(r1)
      if r0 = MAX_INT: abort
      r0 = r0 + 1
      store(r1, r0)
      r1 = 0      // avoid leaking the data capability!
      return r0
data: .word counter
```

A client having only an `E` capability on `next` cannot modify `counter` directly, but only by calling the `next` function.

Procedural or object encapsulation

This kind of encapsulation of private data within code is expressed in many high-level languages using lexical scoping of variable bindings.

```
int next(void) { static int counter = 0; return ++counter; }

let next =
  let counter = ref 0 in fun () -> incr counter; !counter

class Counter {
  private int counter = 0;
  public int next() { return ++counter; }
}
```

Capability machines can enforce this encapsulation even if the attacker does not respect lexical scoping.

Summary on capability machines

The hope: lightweight, fine-grained isolation (like software) but 100% reliable (like hardware).

A fairly old idea (Dennis & Van Horn, 1966).

A commercial failure (Intel iAPX 432, 1981).

Renewed interest with the CHERI architecture (U. Cambridge, 2016) and the ARM Morello processor (2022).

An open question: which capabilities are needed to achieve full isolation for high-level programming languages?

(e.g. A. Lippeveldts (2019) proposes linear capabilities to isolate the stack.)

Interesting connections with separation logic.

(Georges et al, *Cerise: program verification on a capability machine in the presence of untrusted code*, preprint, 2021.)