Language-based software security, fifth cours

Typing and security

Xavier Leroy

2022-04-07

Collège de France, chair of software sciences

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

In this lecture...

Earlier, we saw that run-time safety is necessary for software security.

In this lecture, we study strong typing of programming languages

- as the primary mean to guarantee run-time safety;
- as a mean to obtain security guarantees that go beyond safety.

Typing in programming languages

Types

Folk wisdom:

> *Don't compare apples and oranges.*
> *On n'additionne pas des choux et des carottes.*

Physicist's wisdom: dimensional analysis.

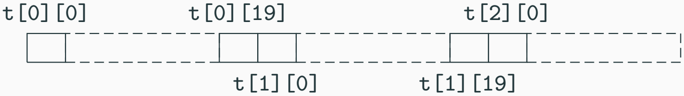
$d = v.t$	dist = dist	homogeneous, possibly correct
$d = v/t$	dist $\neq$ dist.temps^{-2}	not homogeneous, must be wrong

Types to aid compilation

As early as FORTRAN I (1957), type declarations determine the in-memory representation of data and guide machine code generation.

```
float t[10][20];    int i,j;    float x;
x = x + i * t[i][j];
```

In-memory representation of `t`: 200×4 bytes



Evaluation of `x`:

$$\mathtt{x} +^{\text{float}} (\mathtt{floatofint}(\mathtt{i}) \times^{\text{float}} \mathtt{load}(\mathtt{float}, \mathtt{t} + (\mathtt{i} \times 10 + \mathtt{j}) \times 4))$$

Types as a modeling device

> *A method is proposed for the representation in a computer of complex structured objects, and for their manipulation by a program written in a general purpose language, which is here assumed to be an extension of ALGOL 60.*
>
> *(C.A.R. Hoare, Record handling, 1965)*

Introduces records with named, typed fields

```
coloredpoint = { x: float; y: float; c: color }
```

and references to point from one record to another

```
intlist = { head: int; tail: ↑intlist }
```

Modeling with records and references

Hoare's example: representing a set of persons and their family relationships.

```
record class person
begin
  integer date of birth;
  Boolean male;
  reference father, mother,
             youngest offspring, elder sibling (person)
end;
```

Types to prevent programming errors

(C.A.R. Hoare, *Notes on data structuring*, 1970).

[T]he use of a high-level language [...] significantly reduces the scope for programming error.

In machine code programming it is all too easy to make stupid mistakes, such as using fixed point addition on floating point numbers, performing arithmetic operations on Boolean markers, or allowing modified addresses to go out of range.

Types to prevent programming errors

(C.A.R. Hoare, *Notes on data structuring*, 1970).

When using a high-level language, such errors may be prevented by three means:

1. *Errors involving the use of the wrong arithmetic instructions are logically impossible; no program expressed, for example in ALGOL, could ever cause such erroneous code to be generated.*
2. *Errors like performing arithmetic operations on Boolean markers will be immediately detected by a compiler, and can never cause trouble in an executable program.*
3. *Errors like the use of a subscript out of range can be detected by runtime checks on the ranges of array subscripts.*

Types to prevent programming errors

The view of typing as a program verification that avoids errors appears in the PhD thesis of J. H. Morris (1968, MIT):

> *We shall now introduce a type system which, in effect, singles out a decidable subset of those [expressions] that are safe; i.e., cannot given rise to ERRORs.*
>
> *This will disqualify certain [expressions] which do not, in fact, cause ERRORs and thus reduce the expressive power of the language.*

Morris' type system $\approx$ simple types for the λ-calculus.

Typing and run-time safety

In 1978, R. Milner, in his article *A theory of type polymorphism in programming*, states an essential property of a type system:

> *an expression (or program) with a legal type assignment cannot "go wrong"*

Since then, this property has been used as the characterization of a sound type system.

What does *going wrong* means?

Milner (1978) gives his language a denotational semantics based on Scott domains:

$$\mathcal{E} : \mathit{Expr} \to \mathit{Env} \to V$$

where

$$V = (\mathit{Int} + \cdots + (V \to V) + \{\texttt{wrong}\})_\perp$$

`wrong` is the denotation of nonsensical expressions such as

1 2	(the integer 1 used as if it were a function)
$1 + (\lambda x.x)$	(a function used as if it were an integer)

An operational semantics based on reductions

Terms $a ::= n \mid x \mid \lambda x.a \mid a_1\ a_2 \mid \texttt{add}$

Values $v ::= n \mid \lambda x.v \mid \texttt{add} \mid \texttt{add}\ v$

$$(\lambda x.a)\ v \to a[x \leftarrow v] \qquad \frac{n = n_1 + n_2}{\texttt{add}\ n_1\ n_2 \to n}$$

$$\frac{a \to a'}{a\ b \to a'\ b} \qquad \frac{b \to b'}{v\ b \to v\ b'}$$

Generally, terms that go wrong are those that "get stuck" during reduction: they are not values yet they don't reduce.

Example: $1\ 2$ and $\texttt{add}\ 1\ (\lambda x.x)$ don't reduce and are not values.

(See my lecture of 6/2/2020, "Semantics of a functional language")

Type safety in a reduction semantics

An execution of the program a is viewed as a sequence of reductions:

$$\begin{aligned} \text{Termination:} \quad & a \rightarrow \cdots \rightarrow v \in \mathit{Val} \\ \text{Divergence:} \quad & a \rightarrow \cdots \rightarrow a' \rightarrow \cdots \\ \text{Going wrong:} \quad & a \rightarrow \cdots \rightarrow b \not\rightarrow \ \text{ with } b \notin \mathit{Val} \end{aligned}$$

Type safety = if $\emptyset \vdash a : \tau$, the "going wrong" case cannot occur.

The standard proof:

- Preservation: if $a \rightarrow a'$ and $\emptyset \vdash a : \tau$ then $\emptyset \vdash a' : \tau$.
- Progress: if $\emptyset \vdash a : \tau$, then $a \in \mathit{Val}$ or $\exists a'.\ a \rightarrow a'$.

Run-time checks and “normal” errors

A simple type system cannot rule out all sources of run-time errors, especially

- out-of-bounds array accesses;
- arithmetic errors: divide by zero, overflow.

These errors are detected at run-time (dynamic checking) and reported

- either by aborting the program execution
- or by raising an exception, which can be handled by the program.

In both cases, the program does not go wrong!
It just fails a run-time check.

Reduction semantics with errors

$$\mathtt{div}\ n\ 0 \to \mathtt{err} \qquad \frac{n_2 \neq 0 \qquad n = n_1/n_2}{\mathtt{div}\ n_1\ n_2 \to n}$$

Error propagation across execution:

$$\mathtt{err}\ a \to \mathtt{err} \qquad v\ \mathtt{err} \to \mathtt{err}$$

Handling the error:

$$\mathtt{handle}\ v\ a \to v \qquad \mathtt{handle}\ \mathtt{err}\ a \to a$$

$$\frac{a \to a'}{\mathtt{handle}\ a\ b \to \mathtt{handle}\ a'\ b}$$

Type safety in a reduction semantics with errors

A fourth possible outcome for the execution of a program:

Normal termination:	$a \to \cdots \to v \in \mathit{Val}$
Termination on error:	$a \to \cdots \to \mathtt{err}$
Divergence:	$a \to \cdots \to a' \to \cdots$
Going wrong:	$a \to \cdots \to b \not\to$ with $b \notin \mathit{Val}, b \neq \mathtt{err}$

Same definition for type safety:
if $\emptyset \vdash a : \tau$, the "going wrong" case cannot occur.

Similar proof, taking $\emptyset \vdash \mathtt{err} : \tau$
(err is a term that belongs to all types).

What about dynamically-typed languages?

In a dynamic typing approach, there are no "going wrong" situations at run-time, only normal errors.

We can model this by adding error generation rules:

$$x \rightarrow \mathtt{err} \qquad \mathtt{add}\ (\lambda x.a)\ v \rightarrow \mathtt{err}$$

$$n\ v \rightarrow \mathtt{err} \qquad \mathtt{add}\ n\ (\lambda x.a) \rightarrow \mathtt{err}$$

The "going wrong" case becomes impossible:

$$a \rightarrow \cdots \rightarrow b \not\rightarrow \ \text{with } b \notin \mathit{Val}, b \neq \mathtt{err}$$

since every term that is neither a value nor `err` can reduce.
Therefore, dynamic typing is type safe by construction…

From "going wrong" to "undefined behavior"

The standard formalization of "going wrong" suggests an execution that crashes and stops immediately:

$$\mathcal{E}(\eta, a) = \texttt{wrong} \qquad \text{or} \qquad a \not\rightarrow, a \notin \mathit{Val}, a \neq \texttt{err}.$$

This is not a major security risk!
(No more than stopping on a normal error $a \rightarrow \texttt{err}$.)

The major risks are executing arbitrary code, producing a wrong result, revealing a secret, etc.

The C and C++ standards use the notion of undefined behavior to state that anything can happen when the program goes wrong.

Modeling undefined behavior

First try: the problematic terms can reduce to any term a.

$$n\ v \rightarrow a \qquad \texttt{add}\ (\lambda x.b)\ v \rightarrow a \qquad \texttt{add}\ n\ (\lambda x.b) \rightarrow a$$

Limitation: some undefined behaviors cannot be expressed by a term of the language.
(For instance: performing I/O in a pure language.)

Inconvenience: it's hard to distinguish reduction sequences $a \rightarrow a_1 \rightarrow \cdots \rightarrow a_n \rightarrow \cdots$ where "everything is fine" from those that trigger undefined behavior.

Modeling undefined behavior

Alternative: problematic terms reduce to a special `wrong` term, which then can reduce to any term a.

$$n\ v \to \texttt{wrong} \qquad \texttt{add}\ (\lambda x.b)\ v \to \texttt{wrong} \qquad \texttt{add}\ n\ (\lambda x.b) \to \texttt{wrong}$$

$$\text{for all } a,\ \texttt{wrong} \to a$$

Reduction sequences $a \to a_1 \to \cdots \to a_n \to \cdots$ where "everything is fine" are those that do not contain `wrong`.

The usual proof of type safety can be easily adapted:

- Preservation: if $a \to a'$ and $\emptyset \vdash a : \tau$ then $a' \neq \texttt{wrong}$ and $\emptyset \vdash a' : \tau$.
- Progress: if $\emptyset \vdash a : \tau$, then $a \in \mathit{Val}$ or $\exists a'.\ a \to a'$, but $a \neq \texttt{wrong}$.

Type abstraction

Types for hiding representations

> *The meaning of a syntactically-valid program in a "type-correct" language should never depend upon the particular representation used to implement its primitive types.*
>
> *J. C. Reynolds, Towards a theory of type structure, 1974.*

The type system of a language distinguishes base types even when they have same machine-level representation:

integer $\neq$ float $\neq$ reference (repr: 64-bit words)

string $\neq$ code of a function (repr: array of bytes)

Functional encapsulation

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

By lexical scoping, the `counter` reference is only accessible to the `next` function.

In memory, `next` is represented by a function closure: a pair (code pointer, free variable `counter`).

Without strong typing, anyone could access `counter`:

```
let extracted_counter = snd (next : unit * int ref)
```

Type abstraction

> *The meaning of a syntactically-valid program in a "type-correct" language should never depend upon the particular representation used to implement its primitive types. [...]*
> *The main thesis of Morris (1971) is that this property of representation independence should hold for user-defined types as well as primitive types.*
>
> *J. C. Reynolds, Towards a theory of type structure, 1974.*

Type abstraction: a linguistic mechanism to hide the concrete representation of a program-defined data type, forcing users of this type to go through the operations provided over the type.

Capabilities as an abstract type

```
module Capa:
  : sig type t
        val init: unit -> t
        val allowed: permission -> t -> bool
        val drop: permission -> t -> t
    end
  = struct type t = permission list ... end
```

The signature constraint “hides” the fact that `Capa.t` is implemented as `permission list`.

For the clients of `Capa`, the type `Capa.t` is as “opaque” as `float` or `int` $\rightarrow$ `int`.

The only possible values of type `Capa.t` are those obtained by applying `Capa.init` and `Capa.drop`.

Type abstraction in object-oriented languages

In Java, similar guarantees can be achieved by hiding (with the help of visibility modifiers) the internal state and the default constructors of a class.

```
public final class Capa {
    private T capa;
    private Capa(T p) { this.capa = p; }
    public static Capa init() { return new Capa(...); }
    public bool allowed(int p) { ... }
    public Capa drop(int p) { ... }
}
```

Type abstraction and run-time safety

A type system can guarantee run-time safety
(in Milner's sense, *well-typed programs do not go wrong*)
yet fail to enforce type abstraction. Example:

```
module Capa:
  : sig type t ... end
  = struct type t = permission list ... end
```

SML signature constraints have a different meaning than in OCaml. In SML, the type-checker reveals to clients of `Capa` that `Capa.t = permission list`.

The client can, therefore, construct a list `[p1; p2]` of permissions and pass it to any function that expects a `Capa.t`.

This breaks security, but not run-time safety!

Representation independence

(J. C. Reynolds, *Types, abstraction and parametric polymorphism*, 1983)

Respecting type abstraction is not a property of one run of a client of the abstraction: it's a hyperproperty of two runs of the client, linked with two different implementations of the abstraction.

This hyperproperty is called representation independence:
it must be possible to replace one implementation of an abstract type (e.g. `Capa.t = permission list`) with another implementation, without changing the behaviors of the clients of the abstraction.

Two implementations of the abstraction `Capa`

```
type permission = P0 | P1 | P2
```

```
module Capa1 = struct

type t = permission list
let init () = [P0;P1;P2]
let allowed = List.mem
let drop = List.remove

end
```

```
module Capa2 = struct

type t = int
let mask = function
  P0 -> 1 | P1 -> 2 | P2 -> 4
let init () = 7
let allowed p c =
  c land mask p <> 0
let drop p c =
  c land lnot (mask p)

end
```

A relation between the two implementations

Idea: let's construct a relation between the two implementations, telling when a `permission list` and an `int` represent the same abstract set of permissions.

$$
\begin{aligned}
V(\texttt{Capa.t}) = \{\ (L, n) : \texttt{permission list} \times \texttt{int} \mid\ & \\
& (\texttt{P0} \in L \Leftrightarrow \texttt{bit}(n, 0) = 1) \\
& \wedge (\texttt{P1} \in L \Leftrightarrow \texttt{bit}(n, 1) = 1) \\
& \wedge (\texttt{P2} \in L \Leftrightarrow \texttt{bit}(n, 2) = 1)\ \}
\end{aligned}
$$

A logical relation

We then extend this relation $V(t)$ between values to all types t

$$\begin{aligned} V(\texttt{int}) &= \{\ (n, n) \mid n \text{ integer}\ \} \\ V(t \to s) &= \{\ (\lambda x_1.\, a_1, \lambda x_2.\, a_2) \mid \\ &\qquad \forall (v_1, v_2) \in V(t),\ (a_1[x_1 \leftarrow v_1], a_2[x_2 \leftarrow v_2]) \in E(s)\ \} \end{aligned}$$

We then extend it from values to terms (computations)

$$\begin{aligned} E(t) = \{\ (a_1, a_2) \mid \forall b_1,\ a_1 \xrightarrow{*} b_1 \wedge b_1 \text{ irreducible} \Rightarrow \\ \exists b_2,\ a_2 \xrightarrow{*} b_2 \wedge (b_1, b_2) \in V(t)\ \} \end{aligned}$$

Intuition: if $(a_1, a_2) \in E(t)$, the computation a_1 linked with the first implementation of `Capa` behaves exactly like the computation a_2 linked with the second implementation.

The fundamental theorem of logical relations

In a well-typed term a, free variables x_i can be interpreted by related values v_i, v'_i, and the two computations we obtain are related.

Theorem (logical relations)

If $x_1 : \tau_1, \ldots x_n : \tau_n \vdash a : \tau$ *and* $(v_i, v'_i) \in V(\tau_i)$ *for every i, then*

$$(a\{x_i \leftarrow v_i\}, a\{x_i \leftarrow v'_i\}) \in E(\tau)$$

This result is strictly stronger than type soundness: it shows not only run-time safety, but also representation independence.

Relating the two implementations of `Capa`

We show that the operations of the two implementations are related at their types:

$$
\begin{aligned}
&((\texttt{fun ()} \to \texttt{[P0;P1;P2]}), \texttt{fun ()} \to 7) \\
&\qquad \in \quad V(\texttt{unit} \to \texttt{Capa.t}) \\
&(\texttt{List.mem}, \texttt{fun p c} \to \texttt{c land mask p <> 0}) \\
&\qquad \in \quad V(\texttt{permission} \to \texttt{Capa.t} \to \texttt{bool}) \\
&(\texttt{List.remove}, \texttt{fun p c} \to \texttt{c land lnot (mask p)}) \\
&\qquad \in \quad V(\texttt{permission} \to \texttt{Capa.t} \to \texttt{Capa.t})
\end{aligned}
$$

(Just observe that related arguments are mapped to related results.)

Representation independence for `Capa`

Assume $a : \mathtt{int}$ under the typing hypotheses

$$\begin{aligned} \mathtt{Capa.init} &: \mathtt{unit} \to \mathtt{Capa.t} \\ \mathtt{Capa.allowed} &: \mathtt{permission} \to \mathtt{Capa.t} \to \mathtt{bool} \\ \mathtt{Capa.remove} &: \mathtt{permission} \to \mathtt{Capa.t} \to \mathtt{Capa.t} \end{aligned}$$

Let a_1, a_2 be the programs obtained by linking a with one of the implementations of `Capa`:

$$a_1 = a\{\mathtt{Capa} \leftarrow \mathtt{Capa1}\} \qquad a_2 = a\{\mathtt{Capa} \leftarrow \mathtt{Capa2}\}$$

Then, $(a_1, a_2) \in E(\mathtt{int})$.

This means that both programs evaluate to the same integer.

Static typing of resources

Explicit memory management

Dynamic memory allocation + explicit deallocation under the programmer's control.

Example: `malloc` and `free` in C, `new` and `delete` in C++.

```
p = malloc(10);
/* use p */;
free(p);
```

A source of many programming errors!

Memory leak:

```
p = malloc(10);
if ... else return;
free(p);
```

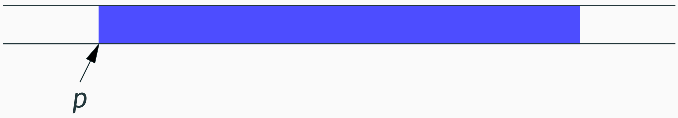
Use after free:

```
p = malloc(10);
...
free(p);
/* use p */
```

Double free:

```
p = malloc(10);
...
free(p);
...
free(p);
```

An attack on use-after-free



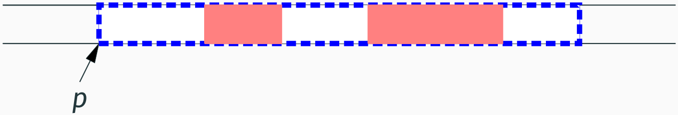
Allocate a large array *p*.

An attack on use-after-free

Allocate a large array *p*.

Free it immediately.

An attack on use-after-free

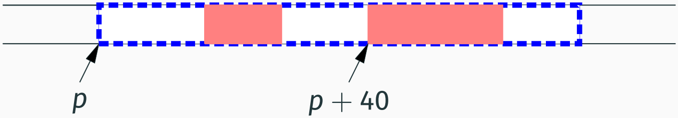


Allocate a large array p.

Free it immediately.

Wait for the memory area to be reused for other allocations (of sensitive data).

An attack on use-after-free



Allocate a large array p.

Free it immediately.

Wait for the memory area to be reused for other allocations (of sensitive data).

Read or modify sensitive data from pointer p.

Note: this invalidates run-time safety, even if the language is strongly typed!

Automatic memory management

No explicit deallocation of memory by the program, but automatic reclamation by the run-time system of the memory blocks that are no longer reachable.

(Reference counting, garbage collection, etc.)

Ex: Lisp, functional languages, scripting languages, Java, Go, …

For a long time, automatic memory management was believed to be necessary for type safety.

Limitations:

- Not always applicable
  (e.g. in an OS or within the implementation of a memory manager).
- Other kinds of resources still need manual management.

Example of resources: file descriptors

A familiar API to read files:

$$\begin{aligned} \texttt{open} &: \texttt{string} \to \texttt{file} \\ \texttt{input\_line} &: \texttt{file} \to \texttt{string} \\ \texttt{close} &: \texttt{file} \to \texttt{unit} \end{aligned}$$

A typical use:

```
let f = open "foo" in
let l = input_line f in
close f; l
```

Incorrect handling of file descriptors

A possible leak of a file descriptor:

```
let f = open "foo" in let l = input_line f in close f; l
```

If file `foo` is empty, `input_line f` raises an exception and `f` is not closed.

A read after close:

```
let f = open "foo" in ... close f; ...; input_line f
```

A double close:

```
let f = open "foo" in ... close f; ...; close f
```

Typically, these errors are detected at run-time.

Aliasing and resource sharing

```
let interleave f1 f2 =
    ... input_line f1 ... input_line f2 ...;
    close f1; close f2
```

If `f1` and `f2` are aliases on the same descriptor, we have a double close.

```
let interleave flist =
    ... List.map input_line flist ...
    List.iter close flist
in
    let f = open "foo" in
    let l1 = [f; open "gee"] and l2 = [f; open "buz"] in
    interleave l1; interleave l2
```

`f` is shared between the two lists `l1` and `l2`, causing a read-after-close in `interleave l2`.

Controlling resources by static typing

An idea that appeared in the pure functional language community, where (morally) we never modify a resource; instead, we return a modified resource.

$$
\begin{aligned}
\texttt{open} &: \texttt{string} \to \texttt{file} \\
\texttt{input\_line} &: \texttt{file} \to \texttt{string * file} \\
\texttt{close} &: \texttt{file} \to \texttt{unit}
\end{aligned}
$$

A new problem appears: we must not use a `file` value twice!

```
let f1 = open "foo" in
let (l1, f2) =
  input_line f1 in   ✔
let (l2, f3) =
  input_line f2 in   ✔
...
```

```
let f1 = open "foo" in
let (l1, f2) =
    input_line f1 in   ✔
let (l2, f3) =
    input_line f1 in   ✘
...
```

Uniqueness types in the Clean language

The type `unique` τ of values of type τ that are reachable via one reference only, and that can therefore be implemented using in-place modification.

```
      open : string → unique file
input_line : unique file → string * unique file
     close : unique file → unit
```

Prevents incorrect reuse of values:

```
let f1 = open "foo" in
let (l1, f2) = input_line f1 in
let (l2, f3) = input_line f1 in
```

Two uses of `f1` : `unique file`, rejected by type-checking.

Leaks are not prevented: we can still forget to close `f3`.

Linear types

(Inspired by Girard's linear logic.)

A type $\sigma \multimap \tau$ of functions that use their σ argument exactly once.

$$\begin{aligned} \texttt{open} &: \forall\alpha.\ \texttt{string} \to (\texttt{file} \multimap \alpha) \to \alpha \\ \texttt{input\_line} &: \texttt{file} \multimap \texttt{string} * \texttt{file} \\ \texttt{close} &: \texttt{file} \multimap \texttt{unit} \end{aligned}$$

Prohibits multiple uses of a `file` value and forces us to call `close` at the end.

(This would not be the case for $\texttt{open} : \texttt{string} \to \texttt{file}$.)

Tracing resource ownership

Another approach to resource management, originating in object-oriented languages, popularized by the Rust language.

```
 open: string → file
close: file → unit
```

In the simplest case, a resource is owned by a variable and is automatically freed at the end of the variable scope.

```
begin let f = open "foo" in
...
(* implicit call to close f *)
end
```

Tracing resource ownership

Another approach to resource management, originating in object-oriented languages, popularized by the Rust language.

```
 open: string → file
close: file → unit
```

The resource can also be explicitly transferred to a function, in which case it is no longer owned by the variable.

```
begin let f = open "foo" in
...
close f
(* no implicit call to close f *)
end
```

Tracing resource ownership

Another approach to resource management, originating in object-oriented languages, popularized by the Rust language.

$$\begin{aligned} \texttt{open:}\ & \texttt{string} \to \texttt{file} \\ \texttt{close:}\ & \texttt{file} \to \texttt{unit} \end{aligned}$$

After transfer, we cannot use the resource any longer, nor transfer it again.

```
let f = open "foo" in
...
close f;
close f ✘
```

Tracing resource ownership

Another approach to resource management, originating in object-oriented languages, popularized by the Rust language.

```
 open: string → file
close: file → unit
```

Taking an alias on a resource is treated like a transfer.

```
let f = open "foo" in
let g = f in
...
close f; ✗
close g
end
```

Borrowing a resource

```
     open: string → file
    close: file → unit
input_line: &mut file → string
  position: & file → int
```

A borrow gives temporary right to use the resource.

```
let f = open "foo" in
let l = input_line (&mut f) in
close f; l
```

The resource `f` is “lent” to `input_line`, then recovered when this function returns.

Borrowing a resource

```
     open: string → file
    close: file → unit
input_line: &mut file → string
 position: & file → int
```

During a mutable borrow, the original owner cannot perform any action on the resource.

```
let f = open "foo" in
let b = &mut f in
close f; ✘
input_line b
```

```
let f = open "foo" in
let b = &mut f in
let l = input_line (&mut f) in ✘
input_line b
```

Borrowing a resource

$$\begin{aligned}\texttt{open}: &\ \texttt{string} \to \texttt{file}\\ \texttt{close}: &\ \texttt{file} \to \texttt{unit}\\ \texttt{input\_line}: &\ \texttt{\&mut file} \to \texttt{string}\\ \texttt{position}: &\ \texttt{\& file} \to \texttt{int}\end{aligned}$$

Several immutable borrows can be ongoing at the same time.
(The "mutable XOR shared" policy.)

```
let f = open "foo" in
let b1 = & f in let b2 = & f in
position b1 + position b2
```

Application: zero-copy message passing

$$\texttt{send: buffer} \rightarrow \texttt{unit}$$
$$\texttt{receive: unit} \rightarrow \texttt{buffer}$$

Passing messages between two threads running concurrently:

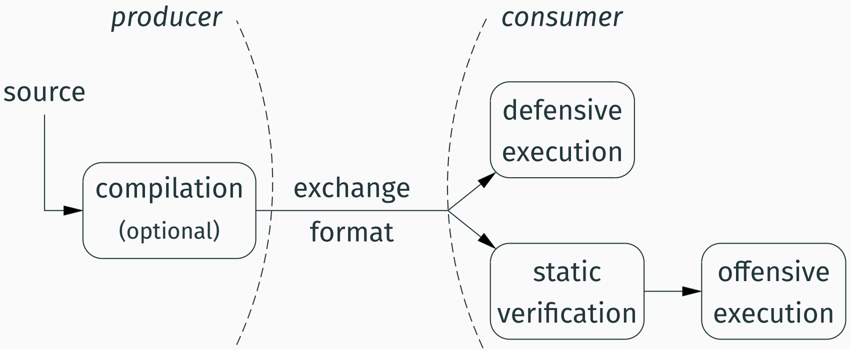
```
let b = new_buffer() in    ||    let b = receive() in
fill(&mut b);              ||    if check(& b)
send(b)                    ||    then use(&mut b)
                           ||    else error()
```

After `send(b)`, the left thread cannot operate on `b`

$\rightarrow$ no Time Of Check To Time Of Use attack
where `b` would change between `check(& b)` and `use(&mut b)`.

Typing and verification of mobile code

Mobile code



Mobile code formats: source (JavaScript), intermediate (JVM bytecode), native (x86 machine code).

What static verifications can we perform on machine codes, either virtual machine(JVM) or hardware processors (x86)?

Java bytecode verification

A static analysis on the JVM bytecode intermediate representation that establishes many useful properties before execution:

- Code is well formed.

  (E.g. no branch in the middle of another method.)
- Instructions receive arguments of the expected types.

  (E.g. `getfield C.f` receives an object of class `C` or a sub-class.)
- The expression stack does not overflow.

  (Within one method; dynamic check at method calls.)
- Local variables (registers) are initialized before use.

  (No access to values that remain from an earlier call.)
- Objects (class instances) are initialized before use.

  (I.e. `new C`, then call to a constructor, then use.)
- Visibility modifiers are respected.

  (E.g. no access to a `private` member outside of the defining class.)

Java bytecode verification

A number of verifications that are crucial for run-time safety and for security are still performed at run-time:

- bounds checks for array accesses;
- checks for null references;
- conversion to a sub-class (down-casting);
- typing stores into arrays of objects;
- stack inspection by the `SecurityManager`.

Verifying branchless code

Executing the JVM code by a defensive abstract machine that uses types in place of values.

- The machine maintains a stack of types and a set of registers containing types.
- For each instruction, it checks the types of the arguments, computes the type of the result, and updates the type of the destination.

Example:

```
class C {
    int x;
    void move(int delta) {
        int oldx = x; x += delta; D.draw(oldx, x);
    }
}
```

```
ALOAD 0

DUP

GETFIELD C.x : int

DUP

ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

r0: C, r1: int, r2: $\top$ [ ]

```
ALOAD 0

DUP

GETFIELD C.x : int

DUP

ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                          r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                          r0: C, r1: int, r2: ⊤    [ C ]
DUP

GETFIELD C.x : int

DUP

ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                        r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                        r0: C, r1: int, r2: ⊤    [ C ]
DUP
                        r0: C, r1: int, r2: ⊤    [ C; C ]
GETFIELD C.x : int

DUP

ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                         r0: C, r1: int, r2: ⊤   [ ]
ALOAD 0
                         r0: C, r1: int, r2: ⊤   [ C ]
DUP
                         r0: C, r1: int, r2: ⊤   [ C; C ]
GETFIELD C.x : int
                         r0: C, r1: int, r2: ⊤   [ C; int ]
DUP

ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                       r0: C, r1: int, r2: ⊤   [ ]
ALOAD 0
                       r0: C, r1: int, r2: ⊤   [ C ]
DUP
                       r0: C, r1: int, r2: ⊤   [ C; C ]
GETFIELD C.x : int
                       r0: C, r1: int, r2: ⊤   [ C; int ]
DUP
                       r0: C, r1: int, r2: ⊤   [ C; int; int ]
ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

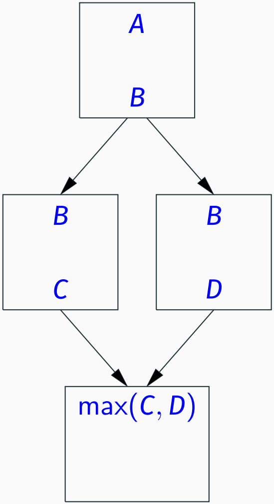
```
                         r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                         r0: C, r1: int, r2: ⊤    [ C ]
DUP
                         r0: C, r1: int, r2: ⊤    [ C; C ]
GETFIELD C.x : int
                         r0: C, r1: int, r2: ⊤    [ C; int ]
DUP
                         r0: C, r1: int, r2: ⊤    [ C; int; int ]
ISTORE 2
                         r0: C, r1: int, r2: int  [ C; int ]
ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                              r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                              r0: C, r1: int, r2: ⊤    [ C ]
DUP
                              r0: C, r1: int, r2: ⊤    [ C; C ]
GETFIELD C.x : int
                              r0: C, r1: int, r2: ⊤    [ C; int ]
DUP
                              r0: C, r1: int, r2: ⊤    [ C; int; int ]
ISTORE 2
                              r0: C, r1: int, r2: int  [ C; int ]
ILOAD 1
                              r0: C, r1: int, r2: int  [ C; int; int ]
IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                        r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                        r0: C, r1: int, r2: ⊤    [ C ]
DUP
                        r0: C, r1: int, r2: ⊤    [ C; C ]
GETFIELD C.x : int
                        r0: C, r1: int, r2: ⊤    [ C; int ]
DUP
                        r0: C, r1: int, r2: ⊤    [ C; int; int ]
ISTORE 2
                        r0: C, r1: int, r2: int  [ C; int ]
ILOAD 1
                        r0: C, r1: int, r2: int  [ C; int; int ]
IADD
                        r0: C, r1: int, r2: int  [ C; int ]
SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                       r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                       r0: C, r1: int, r2: ⊤    [ C ]
DUP
                       r0: C, r1: int, r2: ⊤    [ C; C ]
GETFIELD C.x : int
                       r0: C, r1: int, r2: ⊤    [ C; int ]
DUP
                       r0: C, r1: int, r2: ⊤    [ C; int; int ]
ISTORE 2
                       r0: C, r1: int, r2: int  [ C; int ]
ILOAD 1
                       r0: C, r1: int, r2: int  [ C; int; int ]
IADD
                       r0: C, r1: int, r2: int  [ C; int ]
SETFIELD C.x : int
                       r0: C, r1: int, r2: int  [ ]
ILOAD 2
                       r0: C, r1: int, r2: int  [ int ]
ALOAD 0
                       r0: C, r1: int, r2: int  [ int; C ]
GETFIELD C.x : int
                       r0: C, r1: int, r2: int  [ int; int ]
INVOKESTATIC D.draw : void(int,int)
                       r0: C, r1: int, r2: int  [ ]
RETURN
```

Verifying code containing branches



A classic dataflow analysis:

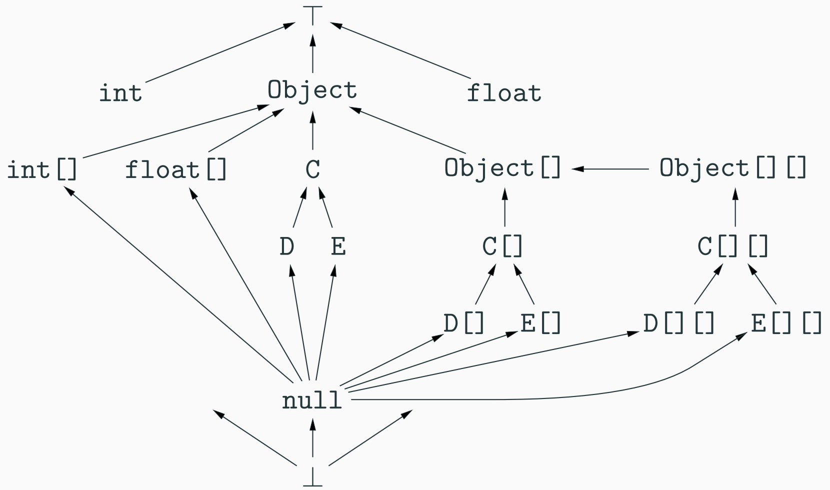
Fork points:
propagate types to
all successors.

Join points:
take the least upper bound of
the types of all predecessors.

Iterate the analysis,
until a fixed point is reached.

(See lecture of 19/12/2019.)

The lattice of JVM types (simplified)



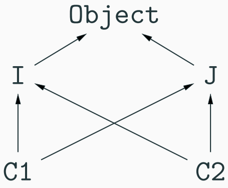
Delicate points

Several JVM features complicate bytecode verification beyond a classic dataflow analysis:

- Interfaces:
  the subtype relation is not a semi-lattice.
- The object initialization protocol:
  requires a bit of must-alias analysis.
- Subroutines:
  a code sharing mechanism, no longer in use, that required a polyvariant analysis.

Interfaces

```
interface I { ... }        class C1 implements I, J { ... }
interface J { ... }        class C2 implements I, J { ... }
```



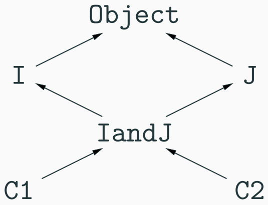
A class can be subtype of several interfaces.

Thus, the subtyping order is not a semi-lattice:

`C1` and `C2` have two incomparable super-types, `I` and `J`.

Interfaces

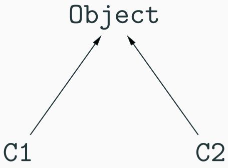
```
interface I { ... }        class C1 implements I, J { ... }
interface J { ... }        class C2 implements I, J { ... }
```



Dedekind-MacNeille completion: adding points to recover a semi-lattice.
Here, the pseudo-class `IandJ` was added as l.u.b. of `C1` and `C2`.

Interfaces

```
interface I { ... }        class C1 implements I, J { ... }
interface J { ... }        class C2 implements I, J { ... }
```



Java's original solution: the bytecode verifier ignores interfaces, treating them all like `Object`.
A run-time check is performed by the `invokeinterface I.m`, since the argument cannot be statically guaranteed to implement interface `I`.

A more modern approach: verification using certificates

(E. Rose, *Lightweight Bytecode Verification*, 2003. The KVM. The JVM since Java 7.)

The Java compiler annotates the produced JVM bytecode with stack maps, i.e. types for the stack and the registers, at some points in the bytecode:

- at the beginning of each basic block (Java 7);
- at each instruction where the types “before” differ from the types “after” the preceding instruction (E. Rose).

Type checking can then be performed in a single linear pass, without fixed-point iteration, and without computing least upper bounds.

Proof-Carrying Code

(G. Necula, P. Lee, *et al*, 1996-2000)

A general, ambitious approach to the security of mobile code:

- Much freedom to choose the language of the mobile code, all the way to machine code (x86 or other) produced by an optimizing compiler, or hand-written.
- Much freedom to choose a security policy, from type safety to triples $\{ P \} \; c \; \{ Q \}$ in a program logic.
- Verifying the code against the policy can be expensive, even undecidable, and involve automated theorem proving.

Proof-Carrying Code

Core idea: separate the verification of the code in two phases:

1. Certification (on the code producer side):
   production of a "proof term" or "certificate"
2. Validation (on the code consumer side):
   checking consistency between certificate, code, and expected property.

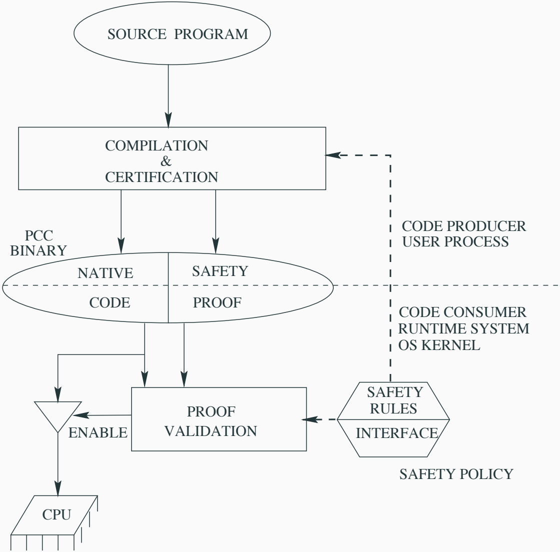
Example: Java bytecode verification.

- Certification: producing stack maps for all basic blocks.
- Validation: type-checking every basic block.

Example: proving a theorem $P : Prop$ in Coq.

- Certification: construction of a proof term $p : P$.
- Validation: type-checking to verify that $p : P$.

The PCC architecture



A fragment of a security policy

(G. Necula, *Compiling with Proofs*, 1998.)

$$
\begin{array}{llcl}
\text{Expressions:} & e & ::= & x \mid e_1 + e_2 \mid e_1 \,\&\, e_2 \mid e_1 \mid e_2 \mid \mathtt{sel}(m, e) \\
\text{Memory:} & m & ::= & x \mid \mathtt{upd}(m, e_1, e_2) \\
\text{Types:} & \tau & ::= & \mathtt{int} \mid \mathtt{bool} \mid \mathtt{array}(\tau, e) \\
\text{Predicates:} & P & ::= & P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall x. P_x \\
 & & & \mid e_1 \geq 0 \mid e : \tau \mid \mathtt{saferd}(e)
\end{array}
$$

Rules:

$$
\frac{e_1 : \mathtt{bool} \qquad e_2 : \mathtt{bool}}{e_1 \,\&\, e_2 : \mathtt{bool}}
\qquad
\frac{e_1 : \mathtt{bool} \qquad e_2 : \mathtt{bool}}{e_1 \mid e_2 : \mathtt{bool}}
\qquad
\frac{}{\mathtt{sizeof}(\mathtt{bool}) = 1}
$$

$$
\frac{a : \mathtt{array}(\tau, len) \qquad i \geq 0 \qquad i < len * \mathtt{sizeof}(\tau)}{\mathtt{saferd}(a + i)}
$$

$$
\frac{a : \mathtt{array}(\tau, len) \qquad \mathtt{sizeof}(\tau) = 1 \qquad i \geq 0 \qquad i < len}{\mathtt{sel}(m, a + i) : \tau}
$$

Array accesses are within bounds.

The representation of the `bool` type is kept abstract.

Production of optimized machine code

$$
\begin{array}{l}
\texttt{bool main}(\texttt{bool } A[], \texttt{bool } B)\ \{\\
\quad \texttt{int } I;\\
\quad \texttt{bool } R\ = B;\\
\quad \texttt{for}(I = 0; I < \texttt{length}(A); I\texttt{++})\\
\qquad R = R \mathbin{\texttt{\&\&}} A[i];\\
\quad \texttt{return } R;\\
\}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{r}_I = 0\\
 & \mathbf{r}_R = \mathbf{r}_B\\
\mathrm{L}_0: & \texttt{INV } \mathbf{r}_I \geq 0 \wedge \mathbf{r}_I : \texttt{int} \ \wedge \mathbf{r}_R : \texttt{bool}\\
 & \texttt{if } \mathbf{r}_I \geq \mathbf{r}_L \texttt{ goto } \mathrm{L}_{\texttt{end}}\\
 & \mathbf{r}_T = *(\mathbf{r}_A + \mathbf{r}_I)\\
 & \mathbf{r}_R = \mathbf{r}_R \mathbin{\&} \mathbf{r}_T\\
 & \texttt{goto } \mathrm{L}_0\\
\mathrm{L}_{\texttt{end}}: & \texttt{return } \mathbf{r}_R
\end{array}
$$

The compiler generated no run-time bounds check because it detected that the access $A[i]$ is always within A's bounds.

The compiler annotated the generated code with a loop invariant.

Production of the verification condition

Using a strongest postcondition calculus, with the specification

$$\{\, r_B : \texttt{bool} \land r_A : \texttt{array}(\texttt{bool}, r_L) \,\}\ c\ \{\, r_R : \texttt{bool} \,\}$$

$$
\begin{array}{l}
\forall \mathbf{r}_A . \forall \mathbf{r}_B . \forall \mathbf{r}_L . \forall m . \\
\quad \mathbf{r}_B : \texttt{bool} \ \land \ \mathbf{r}_A : \texttt{array}(\texttt{bool}, \mathbf{r}_L) \supset \\
\qquad (0 \geq 0 \ \land \ 0 : \texttt{int} \ \land \ \mathbf{r}_B : \texttt{bool}) \ \land \\
\qquad \forall \mathbf{r}_I . \forall \mathbf{r}_R . \\
\qquad\quad \mathbf{r}_I \geq 0 \ \land \ \mathbf{r}_I : \texttt{int} \ \land \ \mathbf{r}_R : \texttt{bool} \supset \\
\qquad\qquad (\mathbf{r}_I < \mathbf{r}_L \supset \mathbf{r}_I + 1 : \texttt{int} \ \land \ \mathbf{r}_R \,\&\, \texttt{sel}(m, \mathbf{r}_A + \mathbf{r}_I) : \texttt{bool} \ \ \land \\
\qquad\qquad\qquad\qquad\qquad \texttt{saferd}(\mathbf{r}_A + \mathbf{r}_I)) \ \ \land \\
\qquad\qquad (\mathbf{r}_I \geq \mathbf{r}_L \supset \mathbf{r}_R : \texttt{bool})
\end{array}
$$

Representing and verifying the proof in LF

The *Logical Framework*: a dependently-typed lambda-calculus, able to express propositions and proof terms.

$$
\begin{array}{lcl}
\texttt{exp} & : & \texttt{Type} \\
\texttt{tp} & : & \texttt{Type} \\
\texttt{pred} & : & \texttt{Type} \\
\texttt{pf} & : & \texttt{pred} \to \texttt{Type} \\
\\
\texttt{true} & : & \texttt{pred} \\
\texttt{and} & : & \texttt{pred} \to \texttt{pred} \to \texttt{pred} \\
\texttt{imp} & : & \texttt{pred} \to \texttt{pred} \to \texttt{pred} \\
\texttt{int} & : & \texttt{tp} \\
\texttt{array} & : & \texttt{tp} \to \texttt{exp} \to \texttt{tp} \\
\texttt{of} & : & \texttt{exp} \to \texttt{tp} \to \texttt{pred} \\
\texttt{saferd} & : & \texttt{exp} \to \texttt{exp} \to \texttt{pred}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{truei} & : & \texttt{pf}\ \ \texttt{true} \\
\texttt{andi} & : & \Pi P\!:\!\texttt{pred}.\Pi R\!:\!\texttt{pred}.\texttt{pf}\ P \to \texttt{pf}\ R \to \texttt{pf}\ \ (\texttt{and}\ P\ R) \\
\texttt{andel} & : & \Pi P\!:\!\texttt{pred}.\Pi R\!:\!\texttt{pred}.\texttt{pf}\ \ (\texttt{and}\ P\ R) \to \texttt{pf}\ P \\
\texttt{szbool} & : & \texttt{pf}\ \ (= (\texttt{sizeof bool})\ 1) \\
\texttt{rdarray} & : & \Pi M\!:\!\texttt{exp}.\Pi A\!:\!\texttt{exp}.\Pi I\!:\!\texttt{exp}.\Pi L\!:\!\texttt{exp}.\Pi T\!:\!\texttt{tp}. \\
 & & \texttt{pf}\ \ (\texttt{of}\ A\ (\texttt{array}\ T\ L)) \to \\
 & & \texttt{pf}\ \ (= (\texttt{sizeof}\ T)\ 1) \to \\
 & & \texttt{pf}\ \ (>= I\ 0) \to \\
 & & \texttt{pf}\ \ (< I\ L) \to \\
 & & \texttt{pf}\ \ (\texttt{of}\ (\texttt{sel}\ M\ (\texttt{plus}\ A\ I))\ T)
\end{array}
$$

Verifying that *c* is a valid proof of proposition *P* is easy: it suffices to check that $c : \texttt{pf}\ P$.

Some uses of PCC

The Touchstone compiler (Colby et al, 2000)

Compilation Java bytecode $\rightarrow$ optimized x86 code, producing certificates of type safety.

Native code injection for network packet filtering (Necula & Lee, 1996)

Like BPF and eBPF, but the code injected in the kernel is native code, and verifying the certificate is simpler than the safety analysis done by eBPF.

PCC limitations

Certificates are huge

- Much redundancy in LF terms, can be improved by using implicit arguments.
- Alternate approach for validation: nondeterministic proof search guided by an “oracle”, which is the certificate.

The security policy and the v.c.gen. are part of the trusted computing base

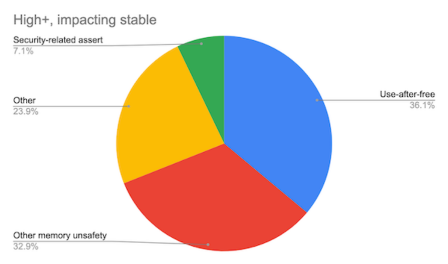
- They must be verified independently.
- *Foundational Proof-Carrying Code*: (Appel et al, 1999-2005) the typing rules and the program logic are derived from the operational semantics of the machine code.

Summary

What does typing contribute to security?

Strong typing (static or dynamic) provides basic guarantees that are necessary for software security: integrity of executions, data structures, and memory.

For instance, these guarantees would have prevented up to 70% of the serious bugs in the Chrome browser.



https://www.chromium.org/Home/chromium-security/memory-safety/

What does typing contribute to security?

Strong typing (static or dynamic) provides basic guarantees that are necessary for software security: integrity of executions, data structures, and memory.

Some type systems provide additional guarantees, such as:

- Type abstraction and representation independence.
- Control of the ownership and proper usage of resources.
- Control of information flow; non-interference (lecture #2).

What does typing contribute to security?

Strong typing (static or dynamic) provides basic guarantees that are necessary for software security: integrity of executions, data structures, and memory.

Some type systems provide additional guarantees, such as:

- Type abstraction and representation independence.
- Control of the ownership and proper usage of resources.
- Control of information flow; non-interference (lecture #2).

Difficulties in combining these type-based approaches and to put them into practice.

What does typing contribute to security?

Strong typing (static or dynamic) provides basic guarantees that are necessary for software security: integrity of executions, data structures, and memory.

Some type systems provide additional guarantees, such as:

- Type abstraction and representation independence.
- Control of the ownership and proper usage of resources.
- Control of information flow; non-interference (lecture #2).

Difficulties in combining these type-based approaches and to put them into practice.

An ongoing transition from typing to program proof,
already anticipated in Proof Carrying Code.