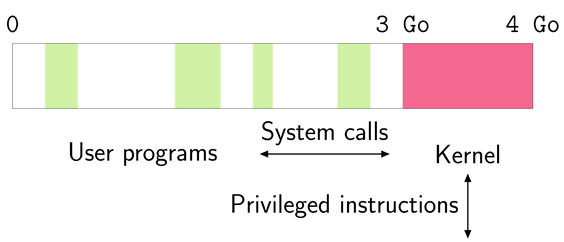
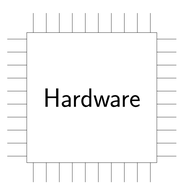
Verifying the Safety of User Pointers Using Static Typing

Etienne Millon[1,2] Emmanuel Chailloux[1] Sarah Zennou[2]
[1]Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France [2] Airbus Group Innovations

Separation in operating systems



Only the kernel can access the hardware.
User programs have to use *system calls* (`open`, `read`, ...).



Execution of a piece of code:
- with processor at level P
- with privilege level C
- accessing data with level D

is possible iff

$$P \leq \min\{C; D\}$$

On x86, levels are *rings*:
Kernel $= 0$ and User $= 3$.

P	C	D	Access
Kernel	Kernel	Kernel	OK
Kernel	Kernel	User	OK
Kernel	User	Kernel	OK
Kernel	User	User	OK
User	Kernel	Kernel	–
User	Kernel	User	–
User	User	Kernel	–
User	User	User	OK

The confused deputy problem

User programs can pass structures to system calls by pointer:

```
struct timeval tv;
int z = gettimeofday(&tv, NULL);
```

The kernel fills in `tv` with its own privileges (not the caller's).
A user can write the current time of the day at any address.
This is *the confused deputy problem*.
The kernel should check at run time that pointers controlled by userspace point to userspace (`copy_{from,to}_user`).
We detect the places where this dynamic check is omitted, using static typing of C code.

Pointer types

Depending on who controls their value, i.e. how they are created:
- kernel pointers: $\&\ x$, etc. They can be dereferenced.

$$\frac{\Gamma \vdash e : t\ *}{\Gamma \vdash *\ e : t}$$

- user pointers: system call arguments . They need a dynamic check.

$$\frac{\Gamma \vdash e_1 : t\ * \qquad \Gamma \vdash e_2 : t\ @}{\Gamma \vdash \texttt{copy\_from\_user}(e_1, e_2) : \textsc{Int}}$$

User pointer sources require annotations: one per system call.

Example: freedesktop.org bug #29340

```
int radeon_info_ioctl(struct drm_device *dev,
                      void *data,
                      struct drm_file *filp) {
  /*!npk userptr_fieldp data value*/
  struct drm_radeon_info *info = data;
  uint32_t *value_ptr = (uint32_t *)
      ((unsigned long)info->value);
  uint32_t value = *value_ptr;
  /* ... */
}
```

Inference output – **error**:

```
05-drm.c:17#10 - Type clash between :
  KPtr (_a15)
  UPtr (_a8)
```

But if we replace last line by the following:

```
if (copy_from_user(&value, value_ptr,
        sizeof(value)))
    return -14;
```

Inference output – **fully annotated program**:

```
(06-drm-ok.c:17#6)^{
  Int tmp_cir!0;
  (06-drm-ok.c:17#6)^tmp_cir!0 <-
    copy_from_user
      ( &(value) : KPtr (d),
        value_ptr_UPtr (d) : UPtr (d),
        4 : Int); }
```

Compilation

```
int f (int* x) {
    return (*x + 1);
}
```

Type erasure

```
let f =
    fun (x) ->
        return (*x + 1)
```

Type inference

```
let f : Int* -> Int =
    fun (x : Int) ->
        return ((((*x) : Int)
            + (1 : Int)) : Int)
```

Input is C code.
GNU extensions used in the kernel are handled.

C type annotations are removed.
The intermediate language has first-order functions and left-values (for partial updates).

Every subexpression gets a type.
This is more precise than C types since abstract types (e.g. user pointers) can be inferred.

Bibliography

N. Hardy.
The confused deputy (or why capabilities might have been invented).
ACM Operating Systems Review, 1988.

C. Hymans and O. Levillain.
Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C.
Technical Note 2008-IW-SE-00010-1, EADS IW/SE, 2008.

R. Johnson and D. Wagner.
Finding user/kernel pointer bugs with type inference.
In *USENIX Security Symposium*, 2004.