Programmation d'un noyau sécurisé en Ada

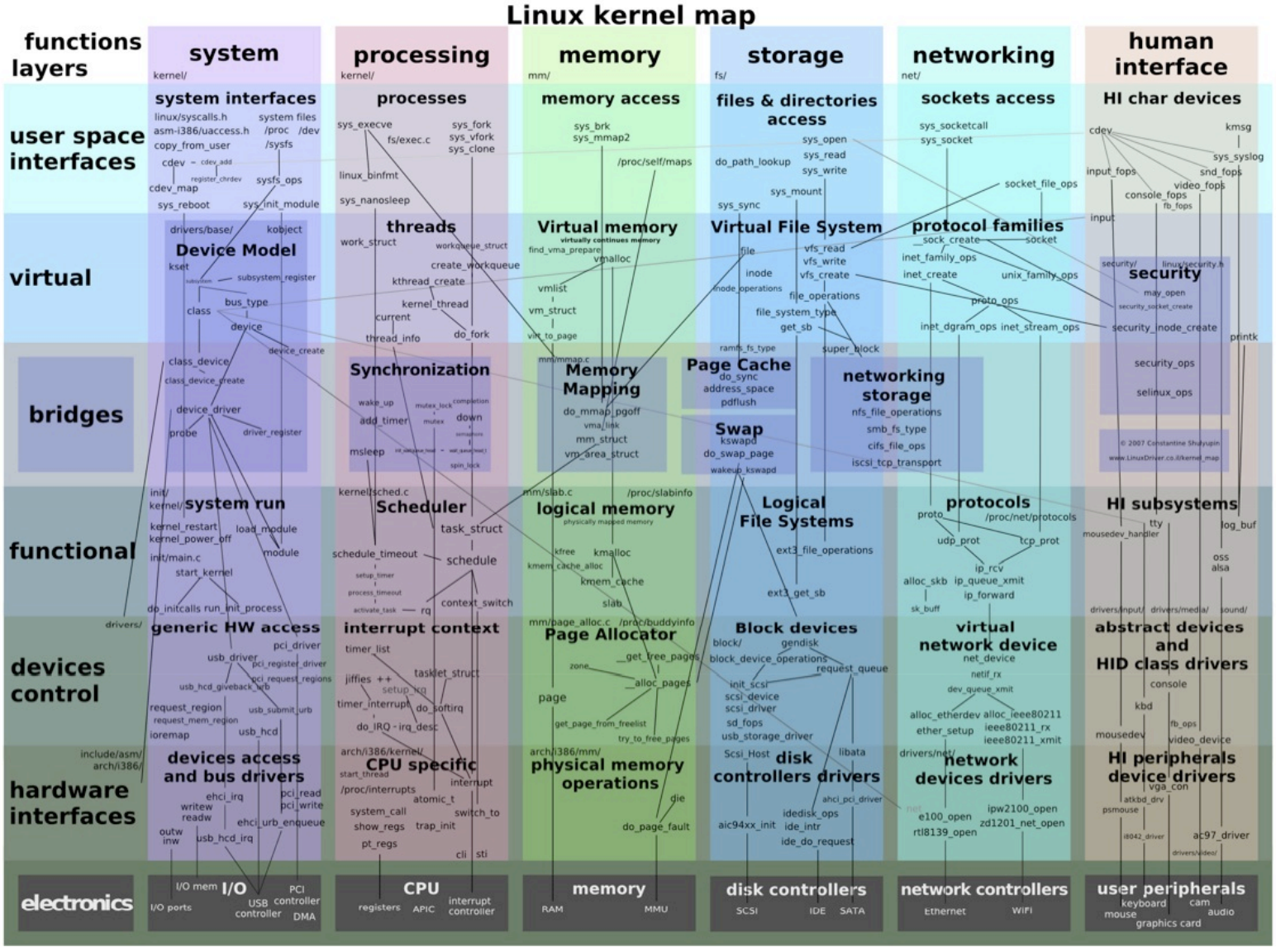
Arnauld Michelizza

ANSSI

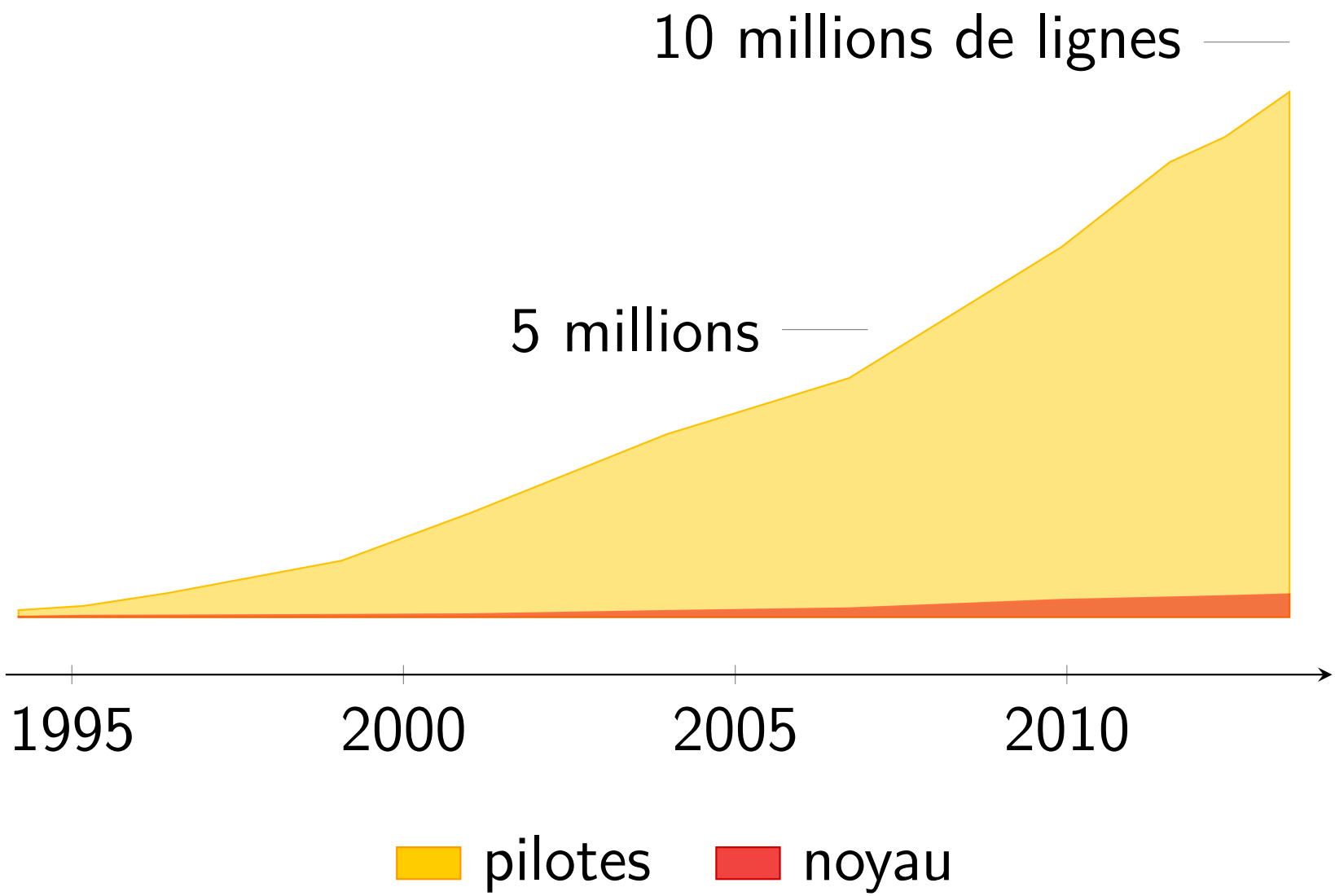
4 juin 2013

Des systèmes d'exploitation
de plus en plus vulnérables !

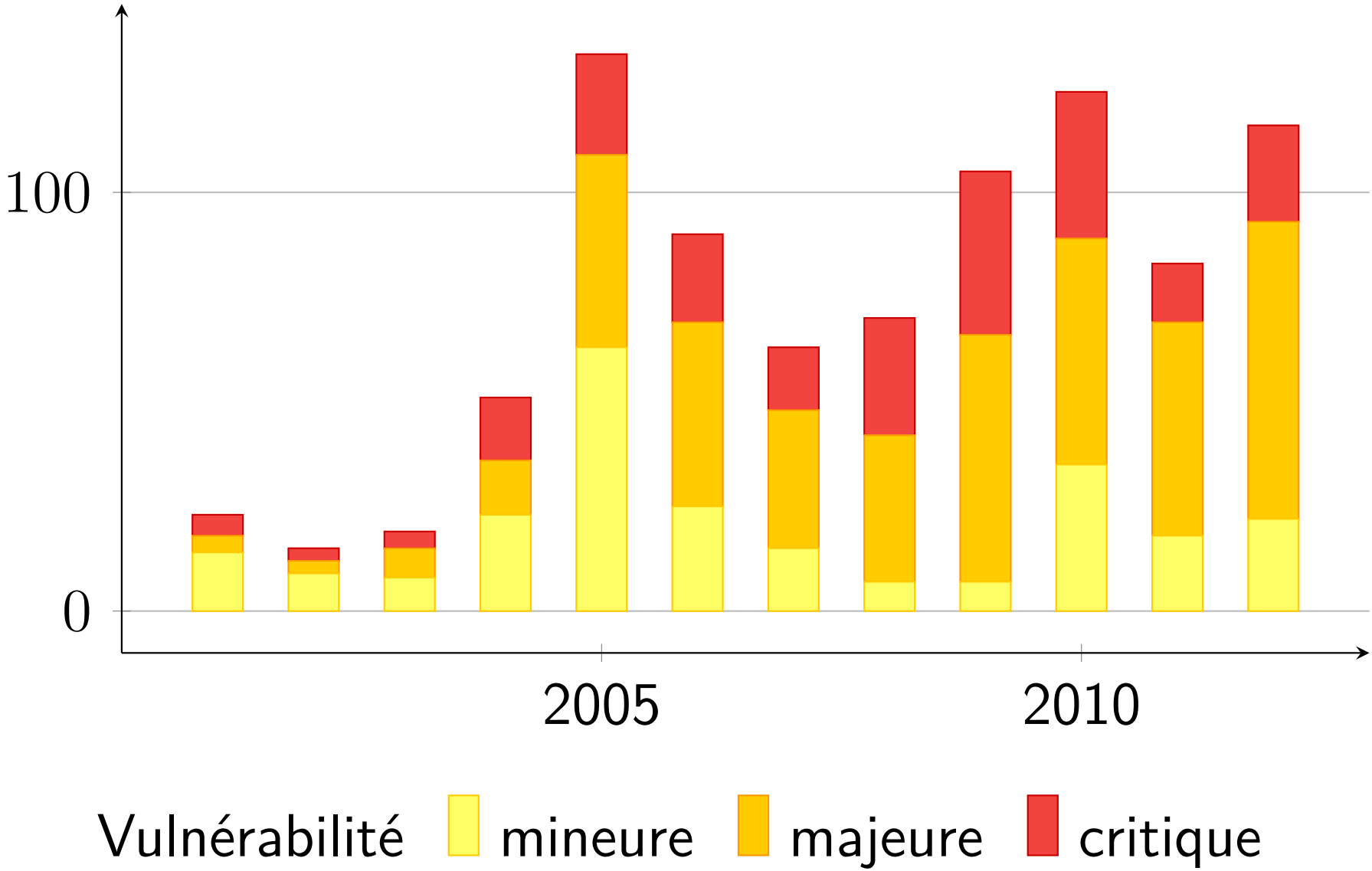
Ils sont complexes



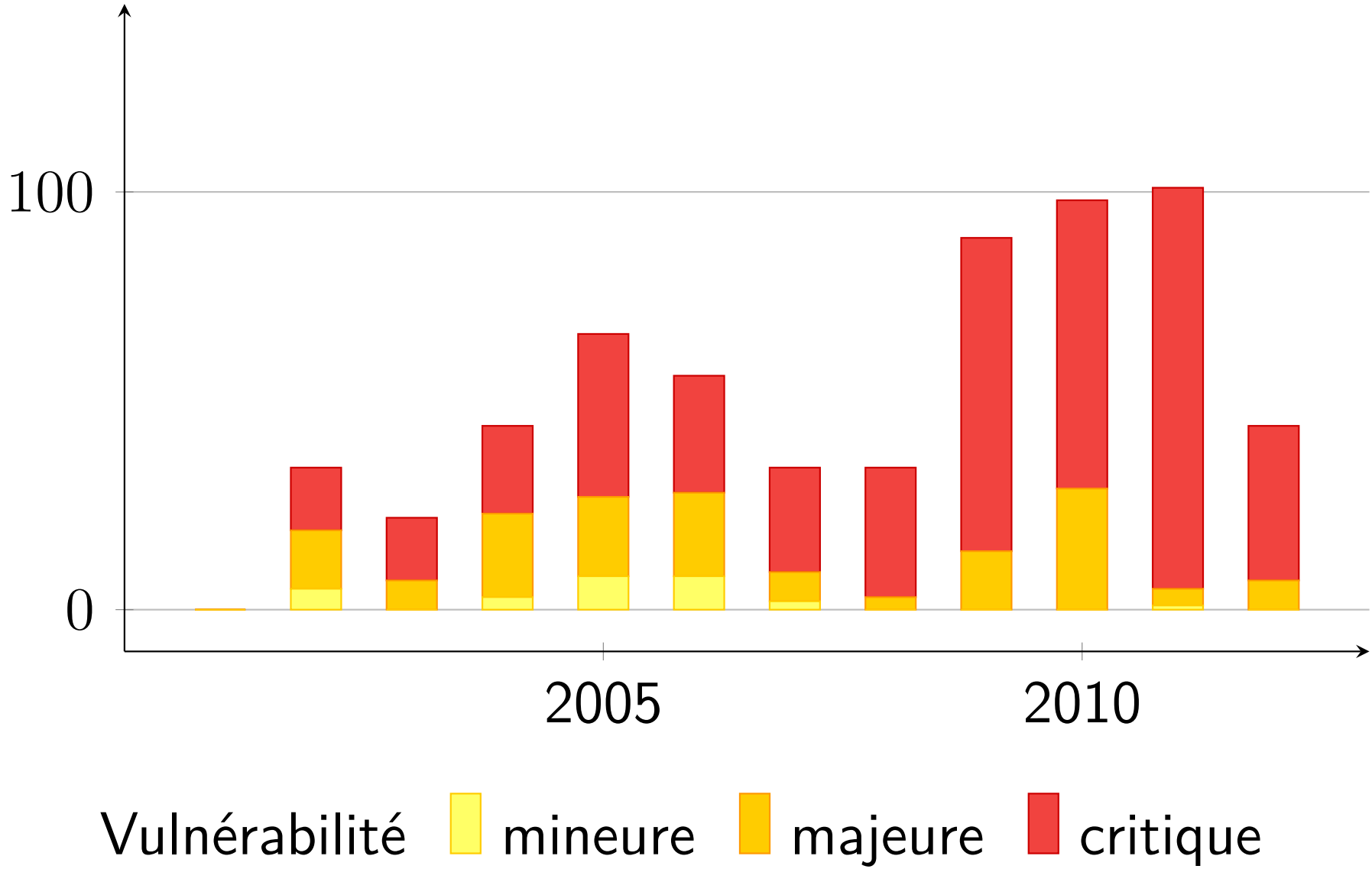
Ils sont complexes : Linux



Ils sont bogués : Linux



Ils sont bogués : Windows XP



Ils sont omniprésents

et connectés de façon permanente !

Comment faire un noyau
sans vulnérabilité ?

Adopter des contre-mesures *ad hoc*

Adopter des contre-mesures *ad hoc*

Empêcher l'exécution de *shellcode*

- pile non-exécutable (bit NX)
- ASLR (Pax/Randkstack)

Adopter des contre-mesures *ad hoc*

Empêcher l'exécution de *shellcode*

- pile non-exécutable (bit NX)
- ASLR (Pax/Randkstack)

Interdire au noyau l'accès à l'espace utilisateur

- Segmentation (Pax/Uderef)
- Pagination + bit SMAP + bit SMEP

Adopter des contre-mesures *ad hoc*

Empêcher l'exécution de *shellcode*

- pile non-exécutable (bit NX)
- ASLR (Pax/Randkstack)

Interdire au noyau l'accès à l'espace utilisateur

- Segmentation (Pax/Uderef)
- Pagination + bit SMAP + bit SMEP

Protéger la pile

- Canaris

Adopter des contre-mesures *ad hoc*

Problèmes

- Mesures intervenant après coup
- Tout ne peut être protégé
- Un service bogué accède à tout l'espace du noyau

Une autre idée ?

Architectures de type micro-noyau

Architectures de type micro-noyau

2 principes de conception

- Les services non fondamentaux sont exclus du noyau
- Chaque module ne peut accèder qu'à ses propres données

Architectures de type micro-noyau

2 principes de conception

- Les services non fondamentaux sont exclus du noyau
- Chaque module ne peut accèder qu'à ses propres données

Minix, Hurd, L4*...

Architectures de type micro-noyau

Avantages

- Noyau plus simple et donc moins vulnérable
- Confinement des compromissions

Architectures de type micro-noyau

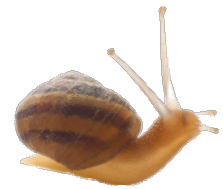
Problèmes

- Remplacer `call` par une IPC

Architectures de type micro-noyau

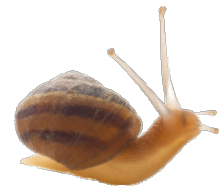
Problèmes

- Remplacer `call` par une IPC
  problèmes de performances



Architectures de type micro-noyau

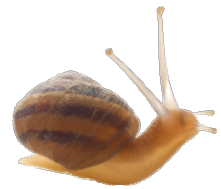
Problèmes

- Remplacer `call` par une IPC
  problèmes de performances 
- Un module compromis (TCP/IP, *pager*)

Architectures de type micro-noyau

Problèmes

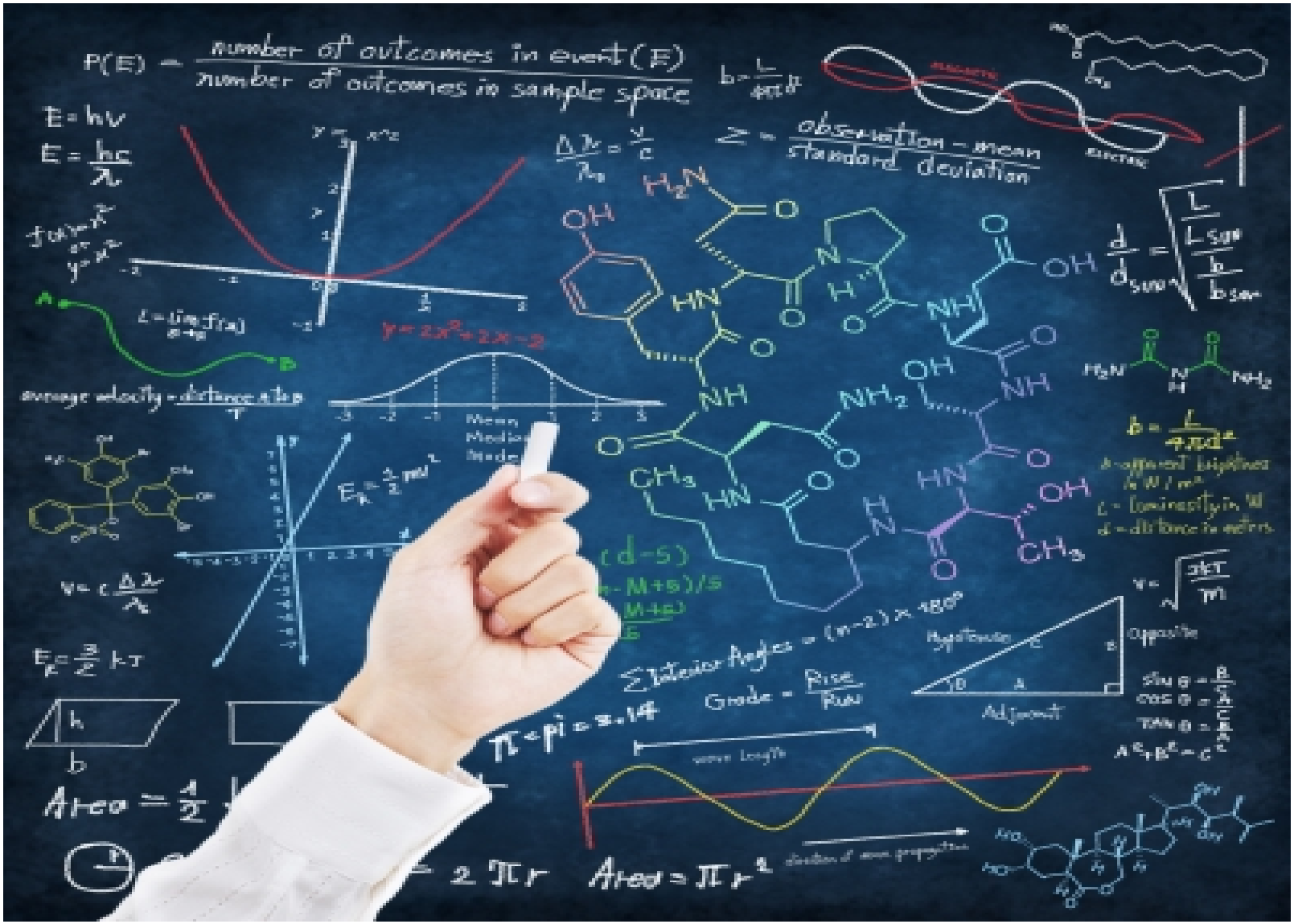
- Remplacer `call` par une IPC
  problèmes de performances



- Un module compromis (TCP/IP, *pager*)
  impacte la sécurité de tout le système

Encore une autre idée ?

Les méthodes formelles



Les méthodes formelles

Pour prouver mathématiquement que...

- la conception est juste
- l'implémentation est correcte

Les méthodes formelles

L'approche idéale ?

- Difficile à mettre en oeuvre

Difficulté de mise en oeuvre : SeL4

8500 lignes de C

Difficulté de mise en oeuvre : SeL4

8500 lignes de C

200000 lignes de preuve

Difficulté de mise en oeuvre : SeL4

8500 lignes de C

11 ans-homme

Les méthodes formelles

Une approche difficilement généralisable

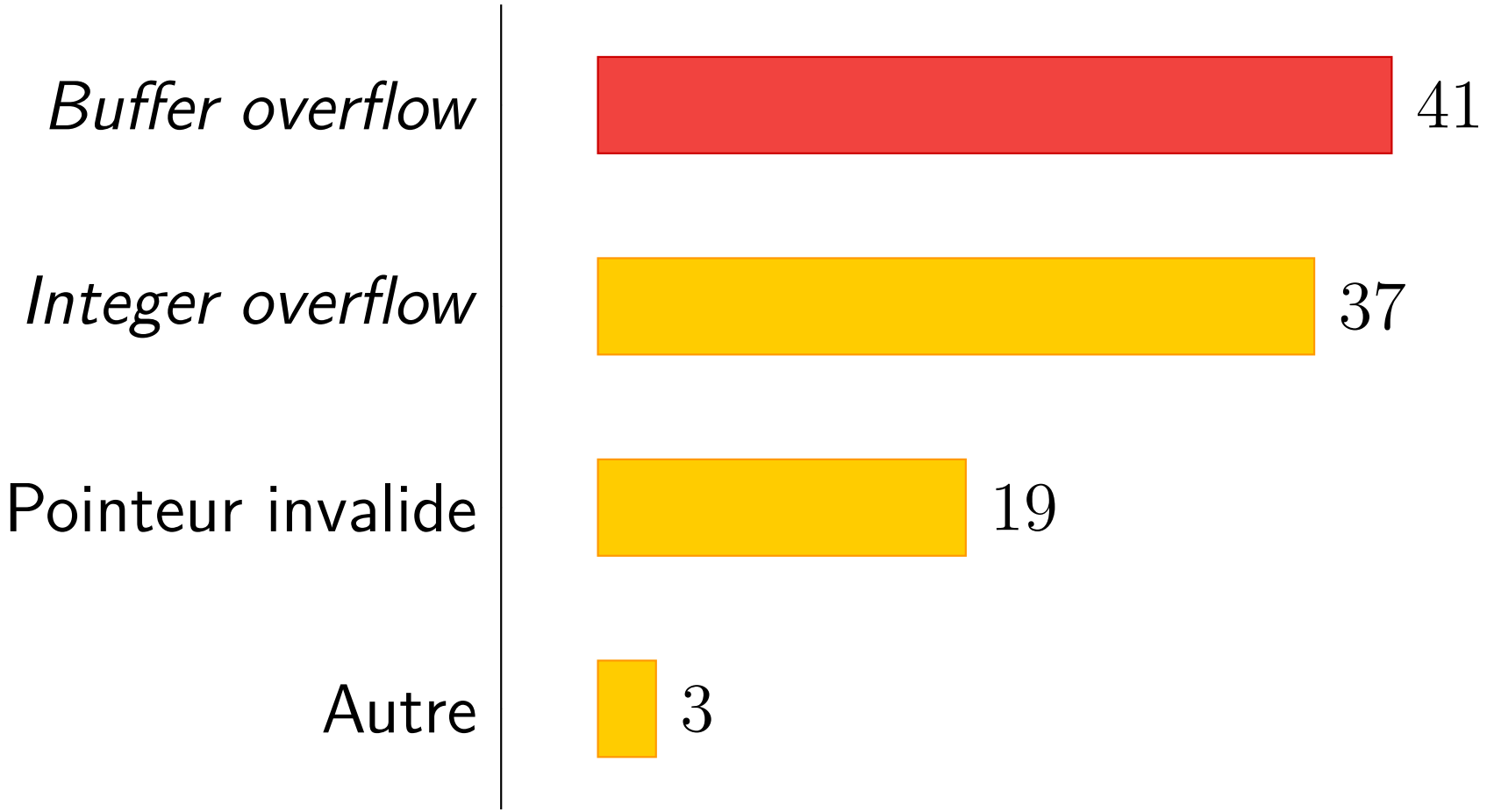
- Difficile à mettre en oeuvre
- Tout ne peut être prouvé
- Impose des contraintes

Bilan de ces 3 approches

- Aucune n'est à exclure
- Elles sont complémentaires
- Chacune a des limitations

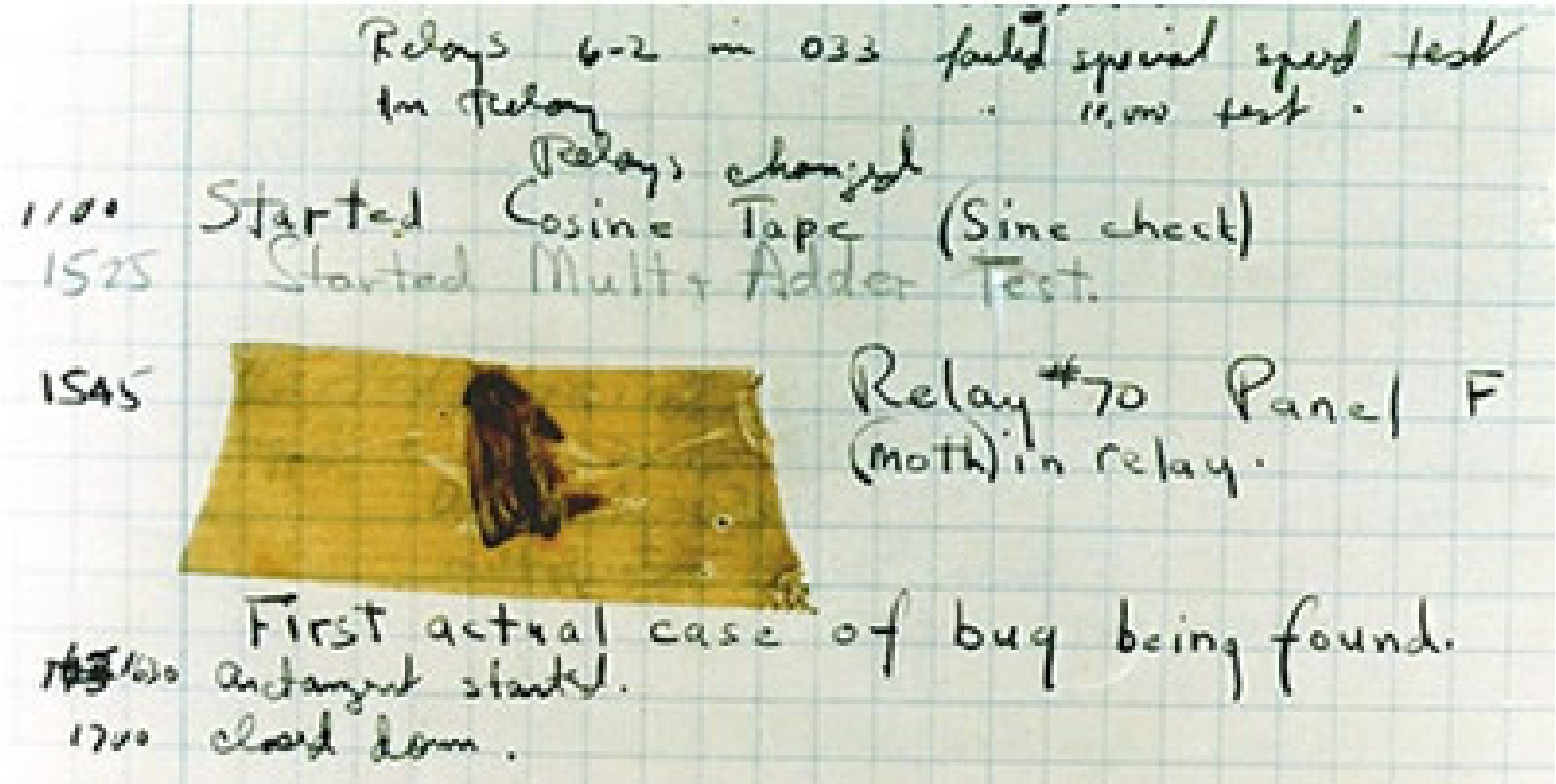
Est-ce qu'il existe un moyen
simple
pour faire un noyau
solide ?

Causes des vulnérabilités critiques



**Chen et al., 2011*

Comment faire un noyau sans bugs ?



Solution 1

Être un dieu de la programmation

Solution 2

Un langage garantissant

l'absence de débordement

=

80% des bugs évités

Quel langage ?

- Algol60, PL/1, Modula-2... plus maintenus

Quel langage ?

- Algol60, PL/1, Modula-2... plus maintenus
- Java, Haskell, Ocaml... pas adaptés

Quel langage ?

- Algol60, PL/1, Modula-2... plus maintenus
- Java, Haskell, Ocaml... pas adaptés
- SafeC, Cyclon, BitC... non maintenus

Quel langage ?

- Algol60, PL/1, Modula-2... plus maintenus
- Java, Haskell, Ocaml... pas adaptés
- SafeC, Cyclon, BitC... non maintenus
- ADA

Le langage ADA aujourd'hui

- Logiciels critiques (ferroviaire, aviation, spatial...)
- Gnat : portage GCC
- Peu de projets opensources
  C 10253 projets vs. ADA 83 projets
  *freecode.com
- Peu enseigné

Un noyau en ADA, c'est possible ?

Noyau ADA

- 11000 lignes en Ada, 400 en assembleur
- Architecture *amd64*, support du *long mode* (64 bits)
- Multi-tâches
- Pagination
- Écran, clavier, disque ATA (mode PIO), carte Intel Pro1000
- Exécutables au format ELF64
- EXT2 (en lecture)
- IPv6

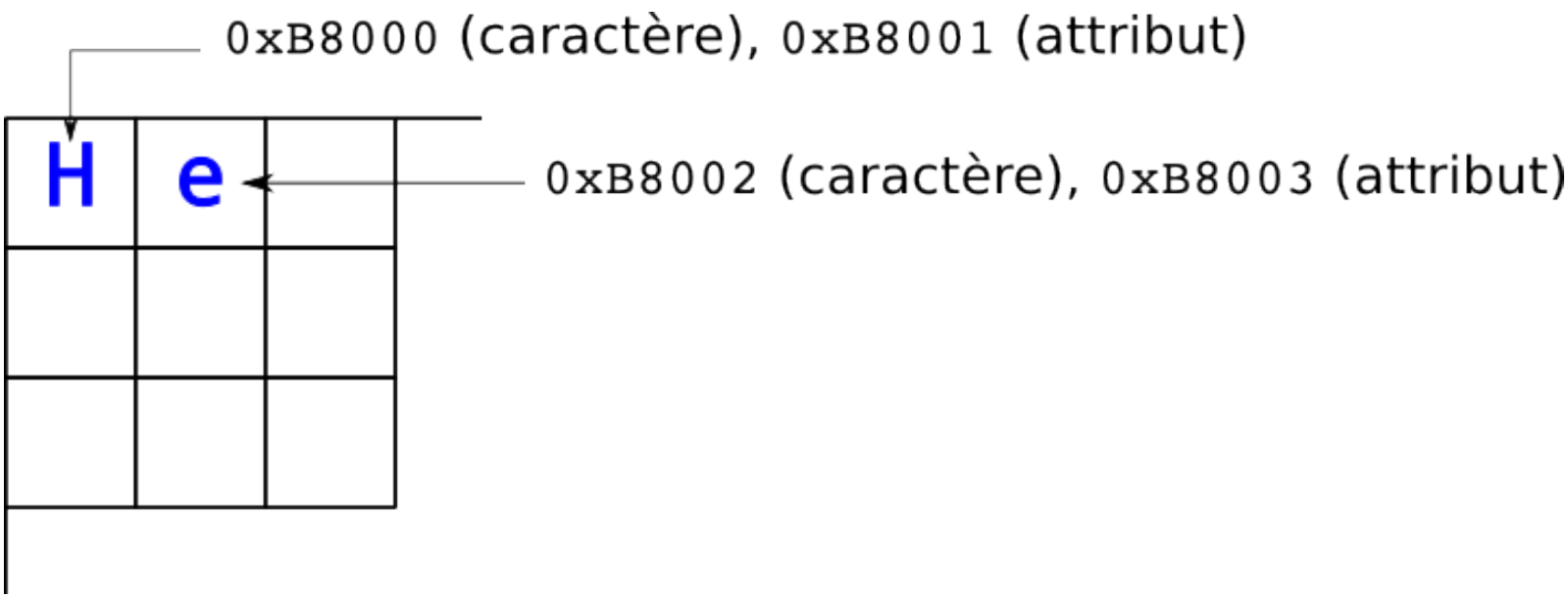
Difficultés rencontrées

- Langage difficile à apprendre
- Penser autrement la conception
- 6 mois pour être à l'aise, et encore des choses à apprendre !

Exemples de code

Écrire *hello world !* à l'écran

- Mémoire vidéo projetée en 0xB8000
- 1 caractère à l'écran codé sur 2 octets



- Il faut écrire directement en RAM

hello world ! en C

Il est facile d'écrire directement en mémoire !

```
volatile char *video = (volatile char *)
    (0xb8000 + 2 * column + 160 * line);
*video = c;
*(video + 1) = color;
```

hello world ! en ADA

- ADA : impossible d'écrire directement en mémoire
- Il faut créer des abstractions !

hello world ! en ADA : définir des types

Nouveau type pour les caractères

```
type t_char is
   record
      char  : unsigned_8;
      attr  : unsigned_8;
   end record;
pragma pack (t_char);
for t_char'size use 16;
```

hello world ! en ADA : définir des types

Nouveau type pour l'écran

```
subtype t_column is integer range 1 .. 80;
subtype t_line   is integer range 1 .. 25;

type t_video_ram is
   array (t_line, t_column) of t_char;
pragma pack (t_video_ram);
for t_video_ram'size use 80 * 25 * 16;

video  : t_video_ram;
for video'address use address (16#b8000#);
```

hello world ! en ADA

Afficher un caractère

```
video(line,column).char  := to_unsigned_8 (c);
video(line,column).attr  := color;
```

Détecter les erreurs de typage en ADA

Détecter les erreurs de typage : exemple

```
1 procedure main is
2    tab  : array (1 .. 10) of integer;
3 begin
4    for i in 1 .. 11
5    loop
6       tab(i) := i;
7    end loop;
8 end main;
```

Détecter les erreurs de typage : exemple

```
1 procedure main is
2    tab  : array (1 .. 10) of integer;
3 begin
4    for i in 1 .. 11
5    loop
6       tab(i) := i;
7    end loop;
8 end main;
```

A l'exécution on obtient

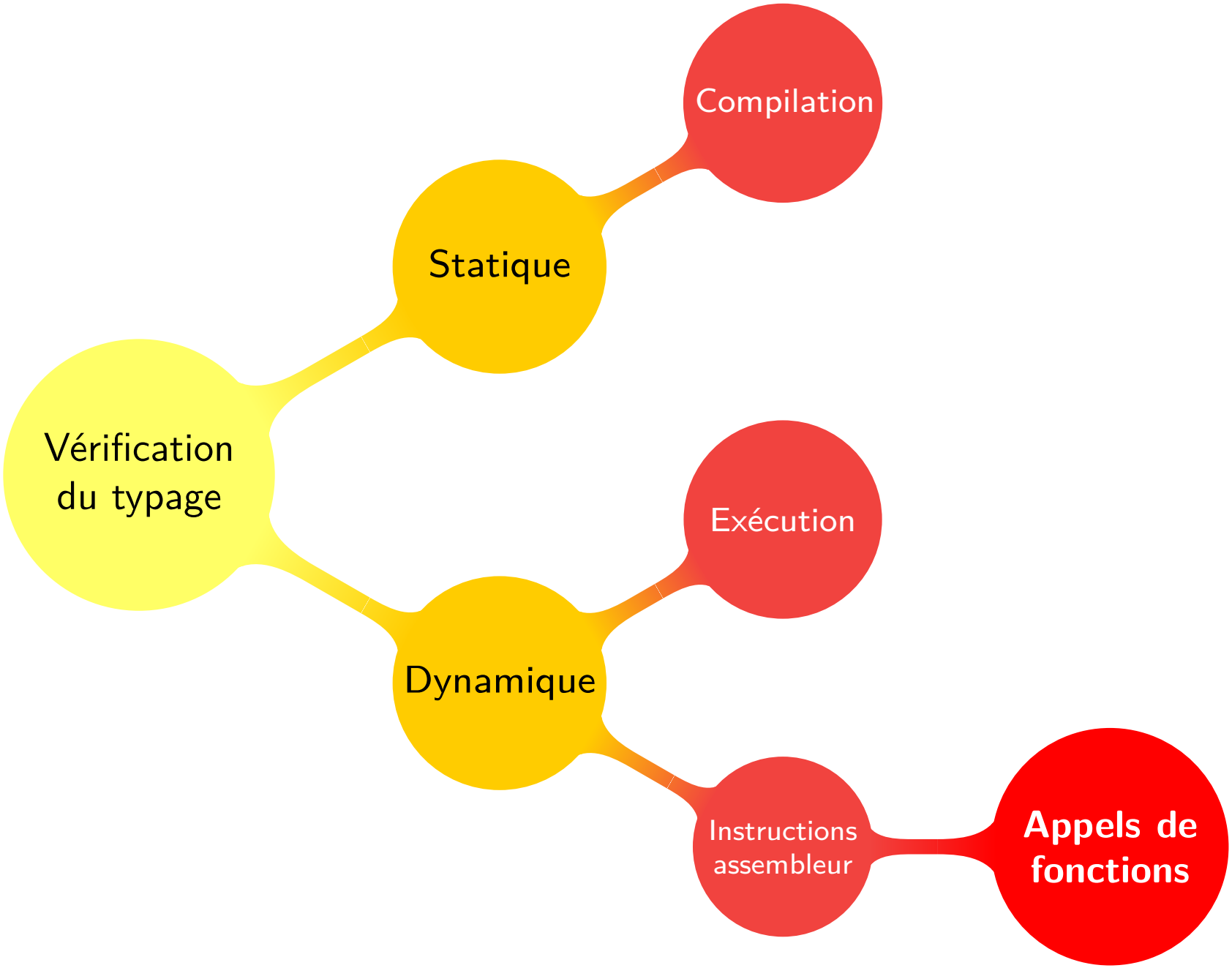
```
raised CONSTRAINT_ERROR : main.adb:6 index check
    failed
```

Détecter les erreurs de typage : exemple

```
1 procedure main is
2    tab  : array (1 .. 10) of integer;
3 begin
4    for i in 1 .. 11
5    loop
6       tab(i) := i;
7    end loop;
8 end main;
```

A l'exécution on obtient

```
raised CONSTRAINT_ERROR : main.adb:6 index check
    failed
```


Compilation
Statique
Vérification du typage
Exécution
Dynamique
Instructions assembleur
Appels de fonctions

Détecter les erreurs dans le noyau

Exemple : un appel système bogué

L'appel système `print` ne vérifie pas l'adresse transmise

```
113 case syscall is
114    -- print
115    when 1 =>
116       declare
117          string_addr  : constant user_address;
118          for string_addr'address use rdi'address;
119       begin
120          syscall_print (string_addr);
121       end;
```

Un code utilisateur malveillant

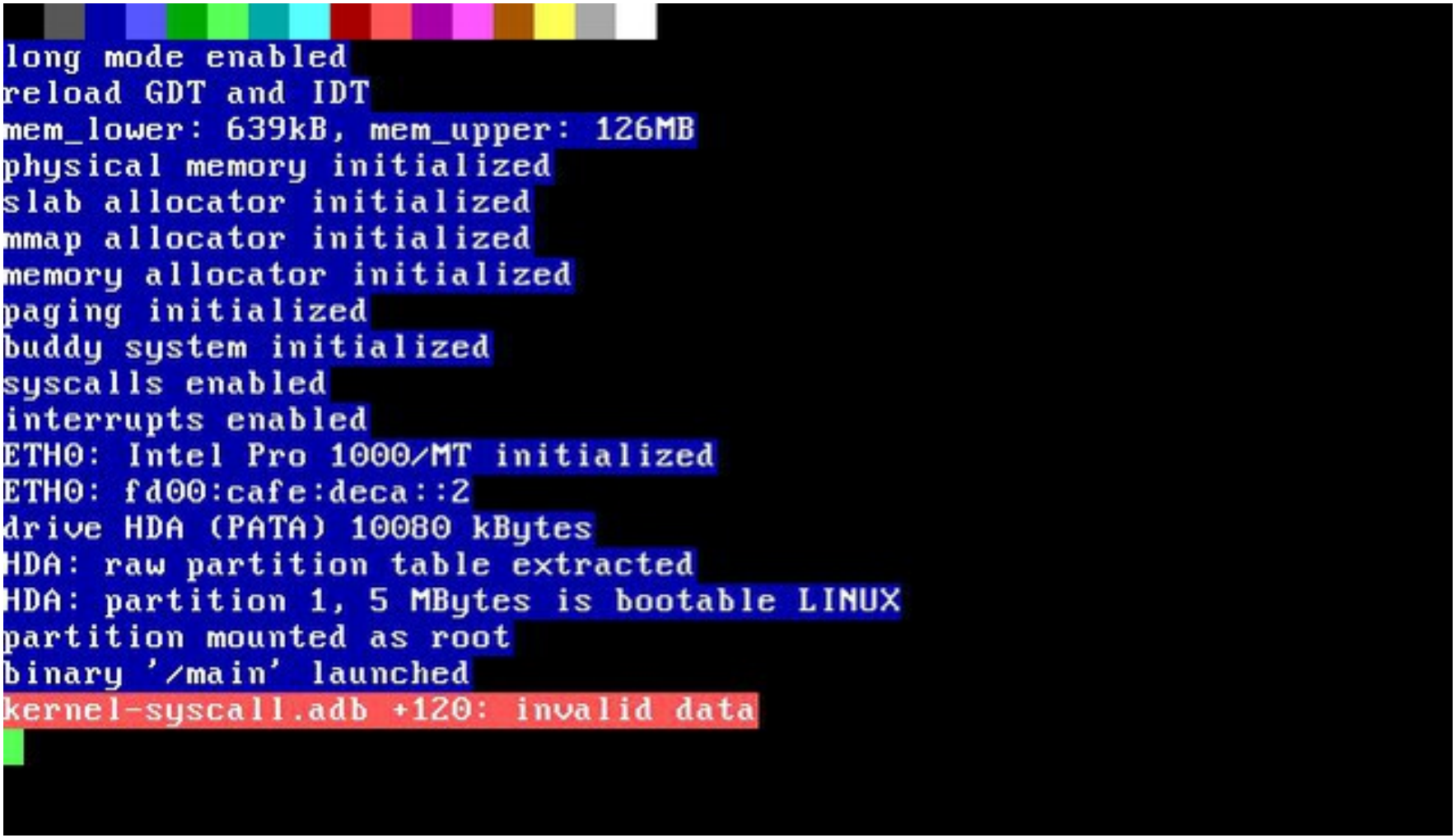
Code normal

```
print ("hello, world\n");
```

Code malveillant

```
print ((char*) 0xffff800000000000);
```

Le noyau est protégé !



kernel-syscall.adb +120: invalid data

Le bug est identifié

```
113  case syscall is
114     -- print
115     when 1 =>
116        declare
117           string_addr  : constant user_address;
118           for string_addr'address use rdi'address;
119        begin
120           syscall_print (string_addr);
121        end;
```

```
kernel-syscall.adb +120: invalid data
```

Avantages de ADA

- Absence de *buffer overflows*, Integer overflows et déréférencement de pointeurs nuls
- Visibilité des erreurs
- Facilité pour déboguer

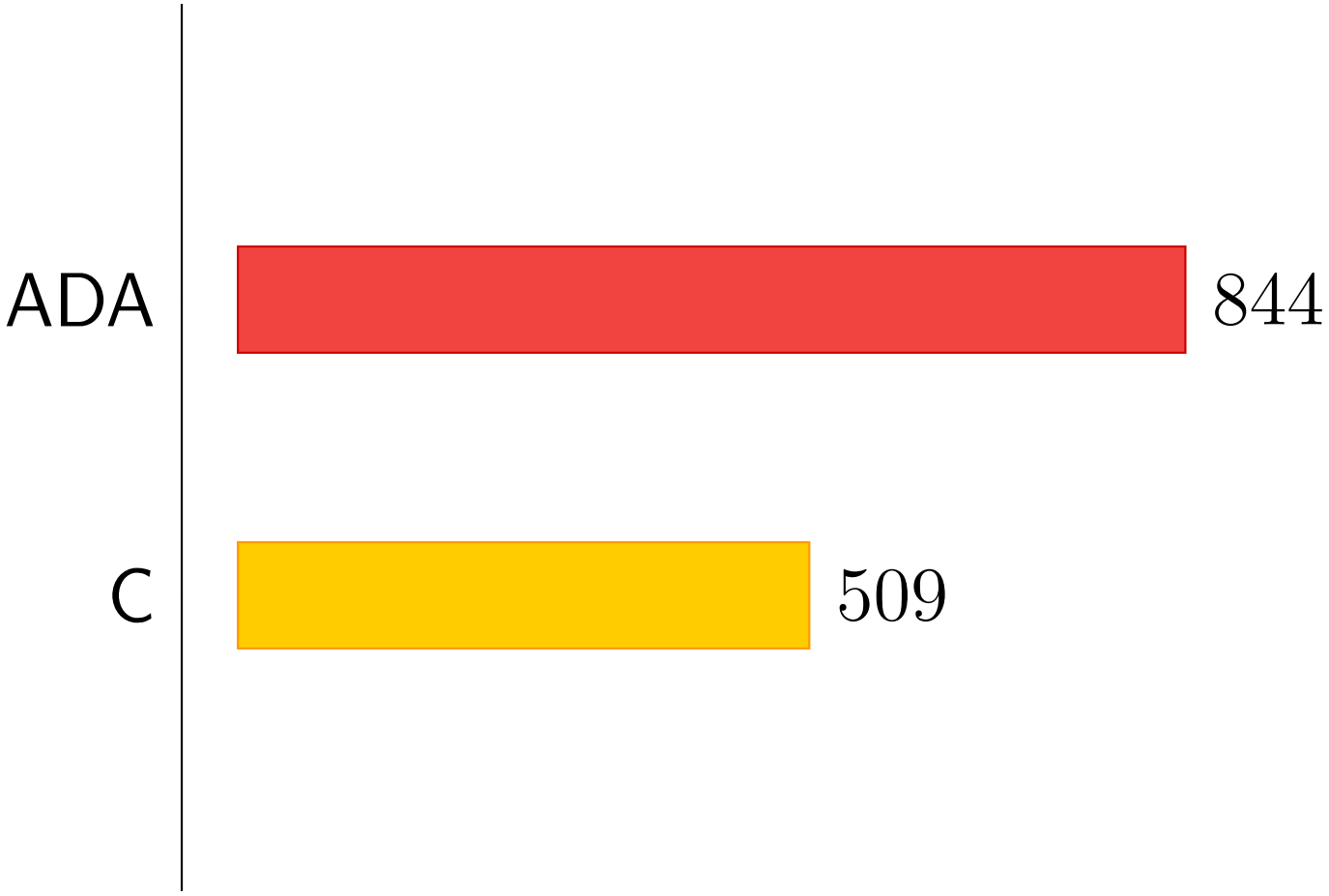
Linux : un exemple récent

CVE-2013-2094

1. Erreur de conversion : `int` au lieu de `u64`
2. Utilisation comme index dans un tableau
3. Débordement de tableau (*underflow*)
4. Au final : une élévation de privilèges

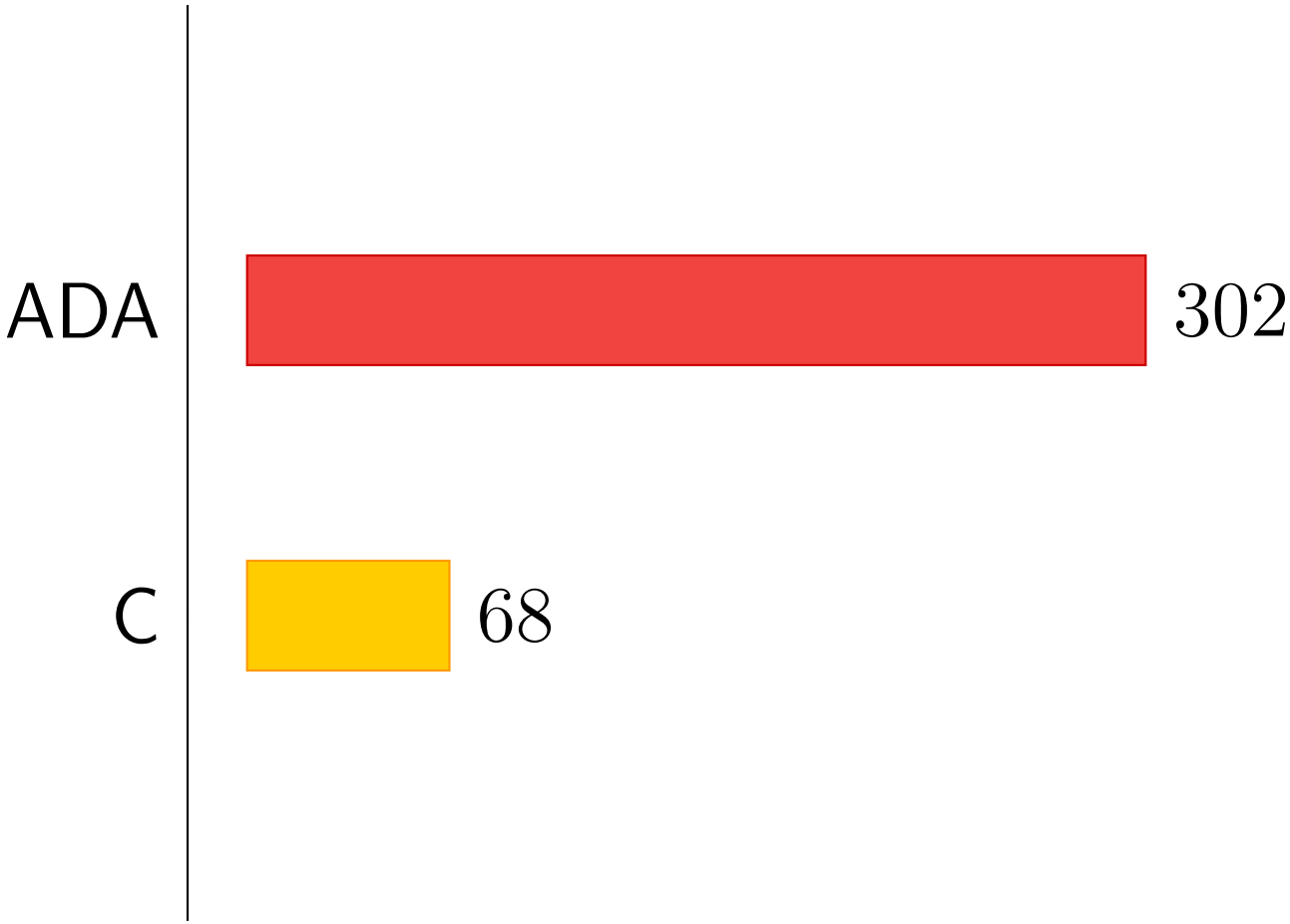
Vérifications dynamiques
Quel coût en performances ?

Cycles d'horloge : +65%



Optimisation : -O2

Taille en octets : x4

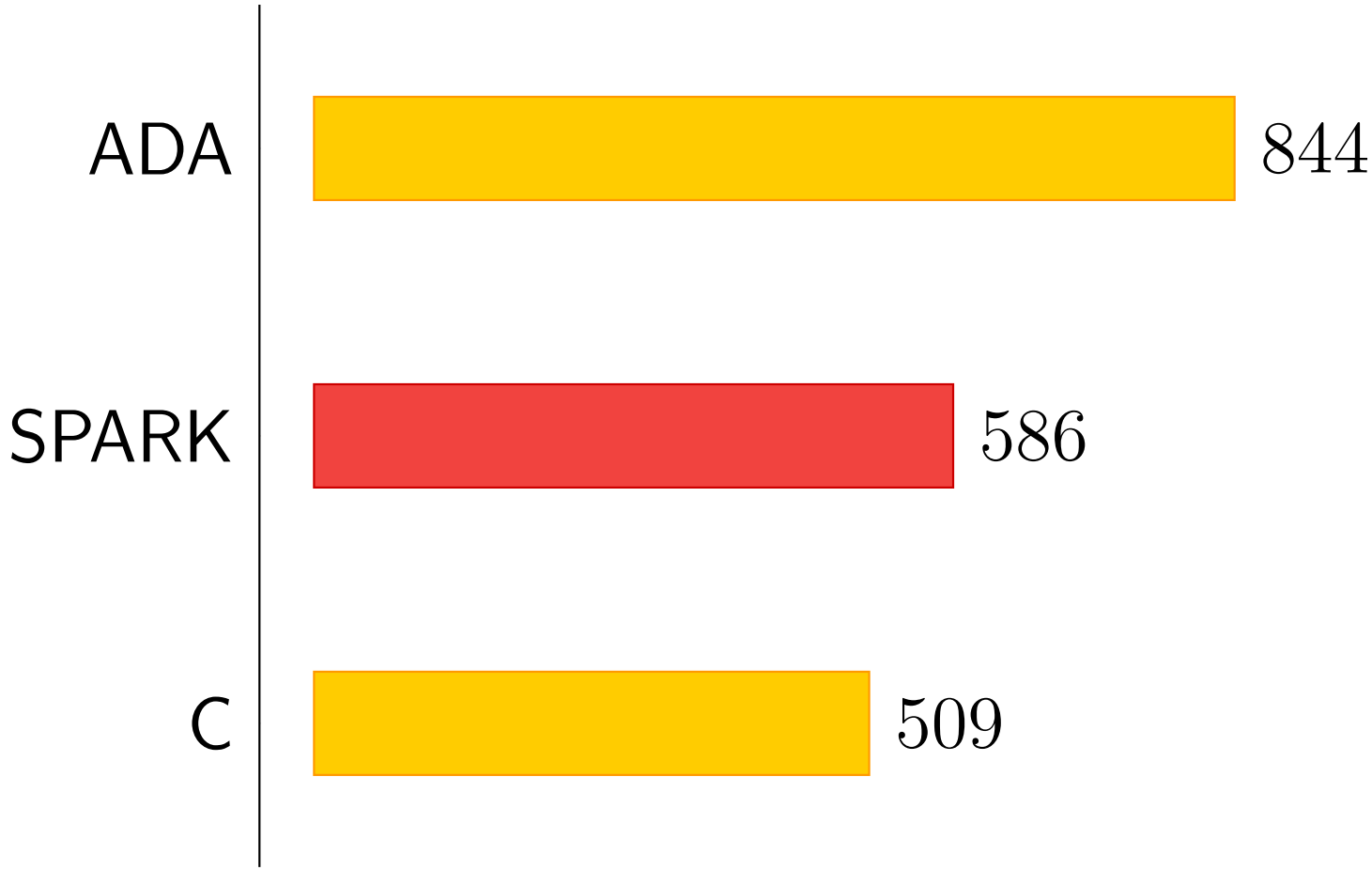


Optimisation : -O2

Désactiver les vérifications ?

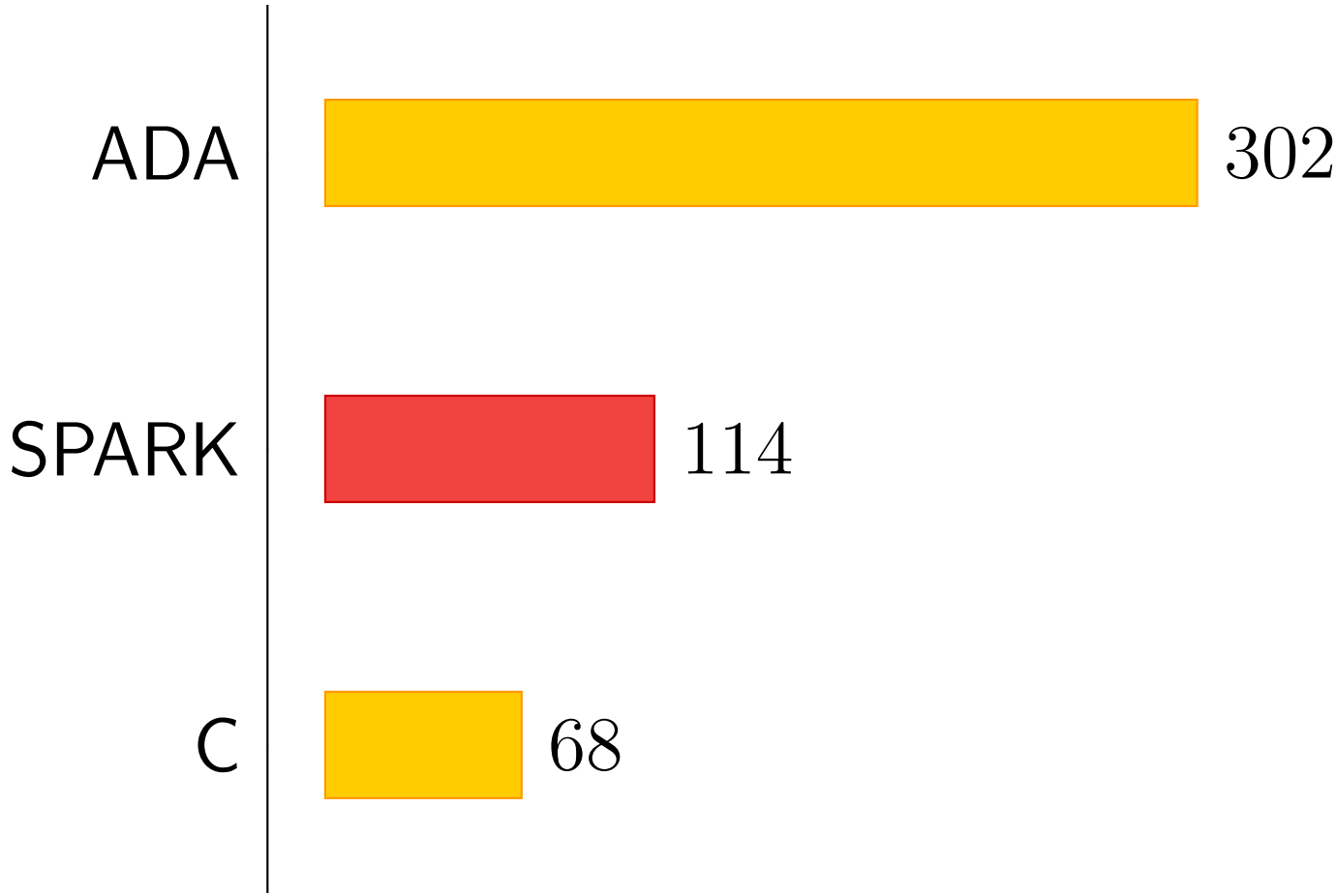
- SPARK !
- Ensemble d'annotations et d'outils de preuve

Cycles d'horloge : +15%



Optimisation : -O2

Taille en octets : +70%



Optimisation : -O2

Conclusion

Une méthode simple, un noyau sûr

- Absence de *buffer overflows*, Integer overflows et déréférencement de pointeurs nuls
- 80% des vulnérabilités potentielles supprimées
- Visibilité des erreurs
- Facilité pour déboguer

Démo !