L'interprète, le JIT et la licorne

Frédéric Recoules & Sébastien Bardin

35es Journées Francophones des Langages Applicatifs

En quelques mots

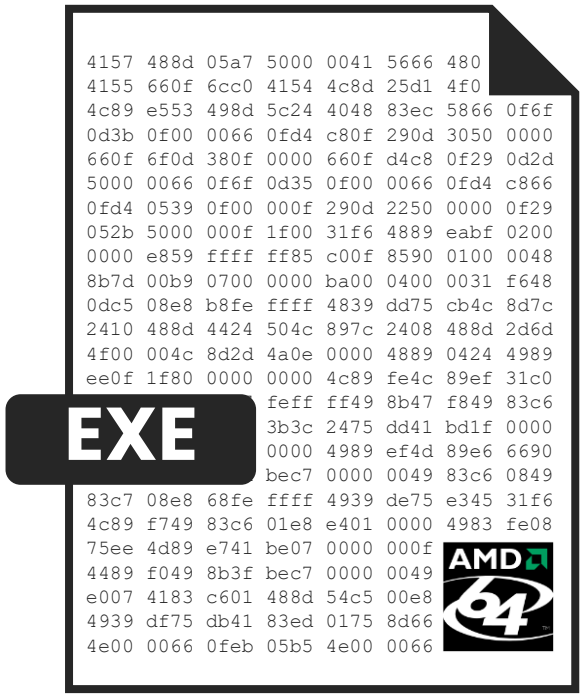
Contexte Une exécution symbolique au niveau binaire codée en OCaml

Problème L'interprétation des instructions qui devient *LE* goulot d'étranglement de l'analyse *dans certains cas*

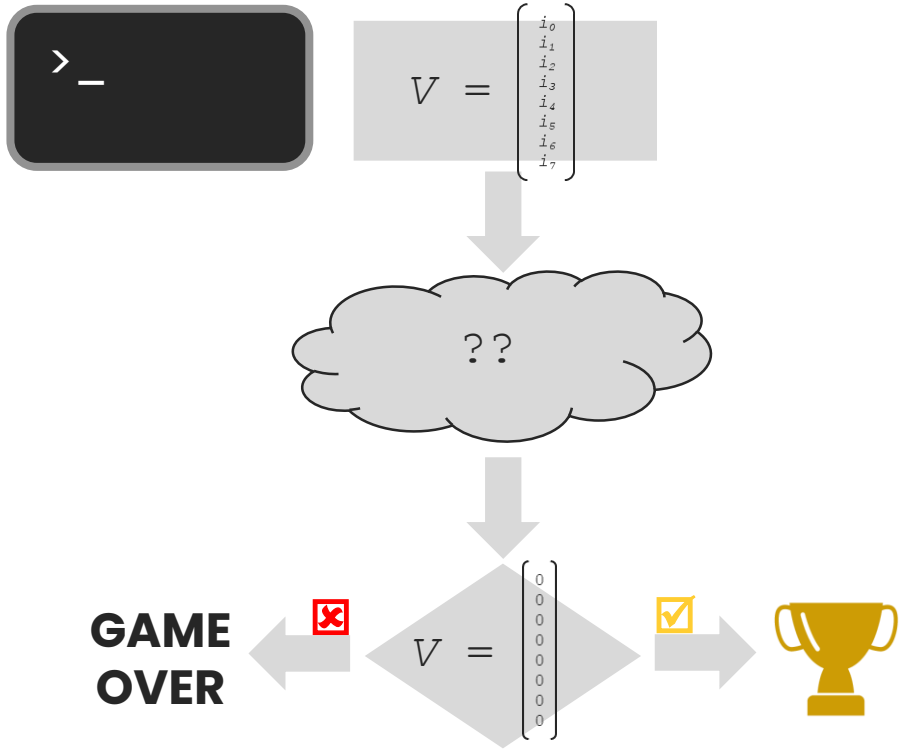
Contribution De l'optimisation des blocs de base à l'aide compilation à la volée en OCaml

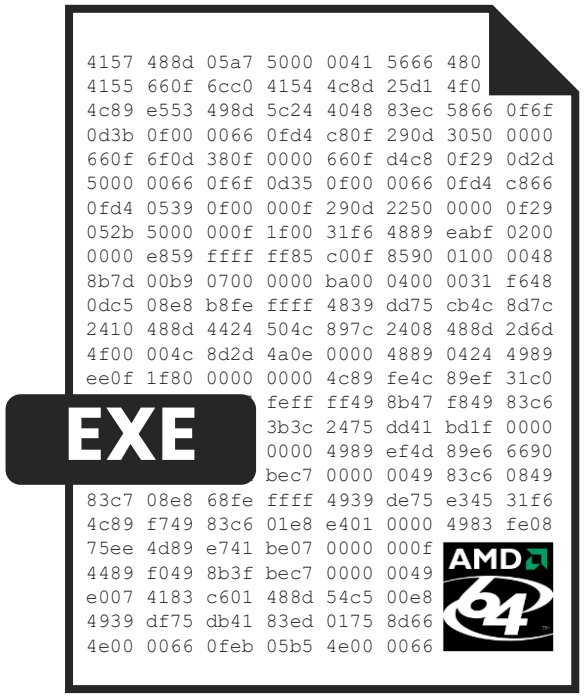
- Sûr et facile d'utilisation
- Efficace et autonome à l'exécution
- Au prix d'une expressivité restreinte

Le cas du *France CyberSecurity Challenge* 2022



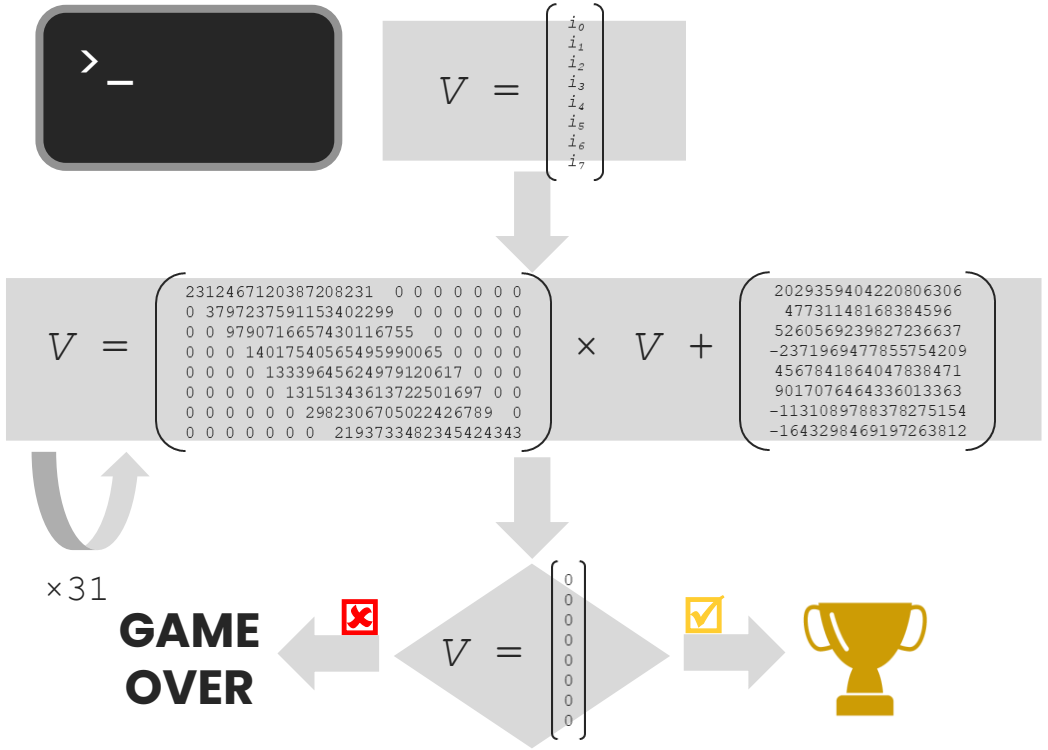
Le cas du *France CyberSecurity Challenge* 2022

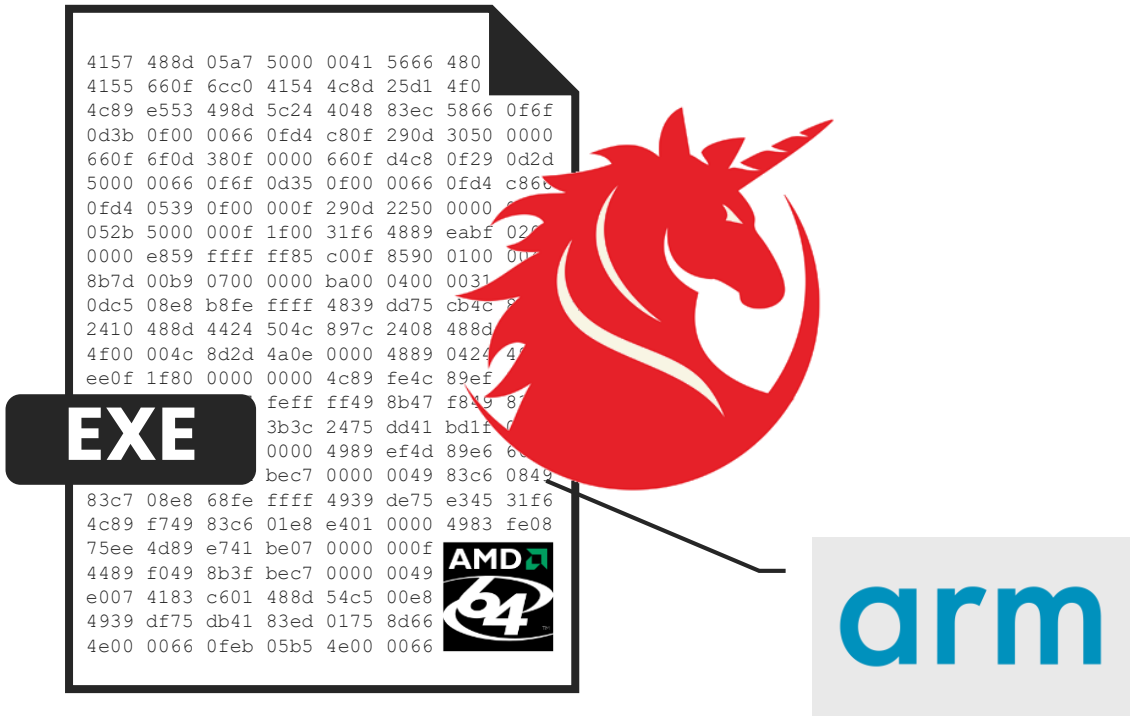




BINSEC

Le cas du *France CyberSecurity Challenge* 2022

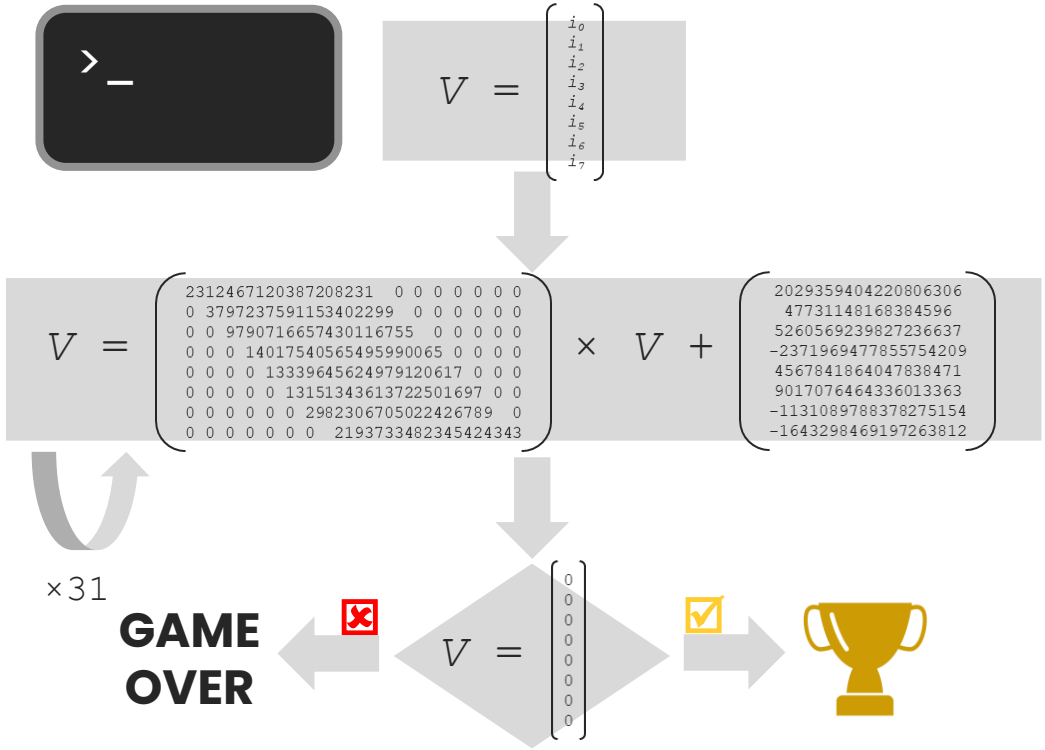


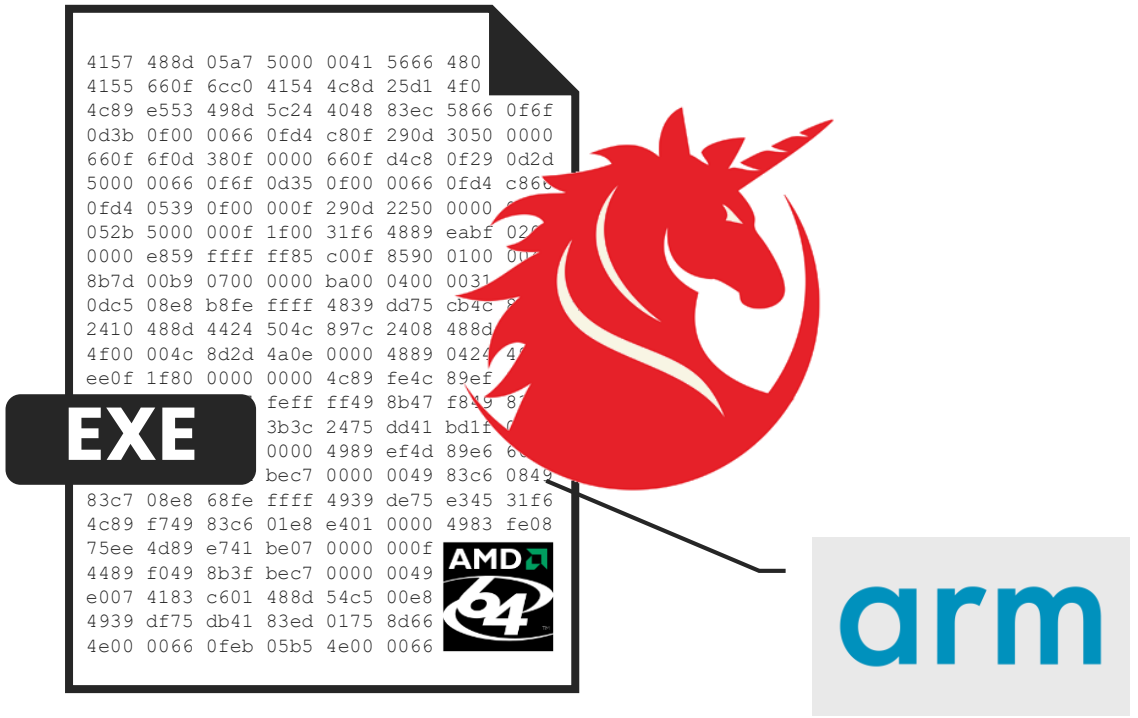


999 859 832 instructions

BINSEC

Le cas du *France CyberSecurity Challenge* 2022





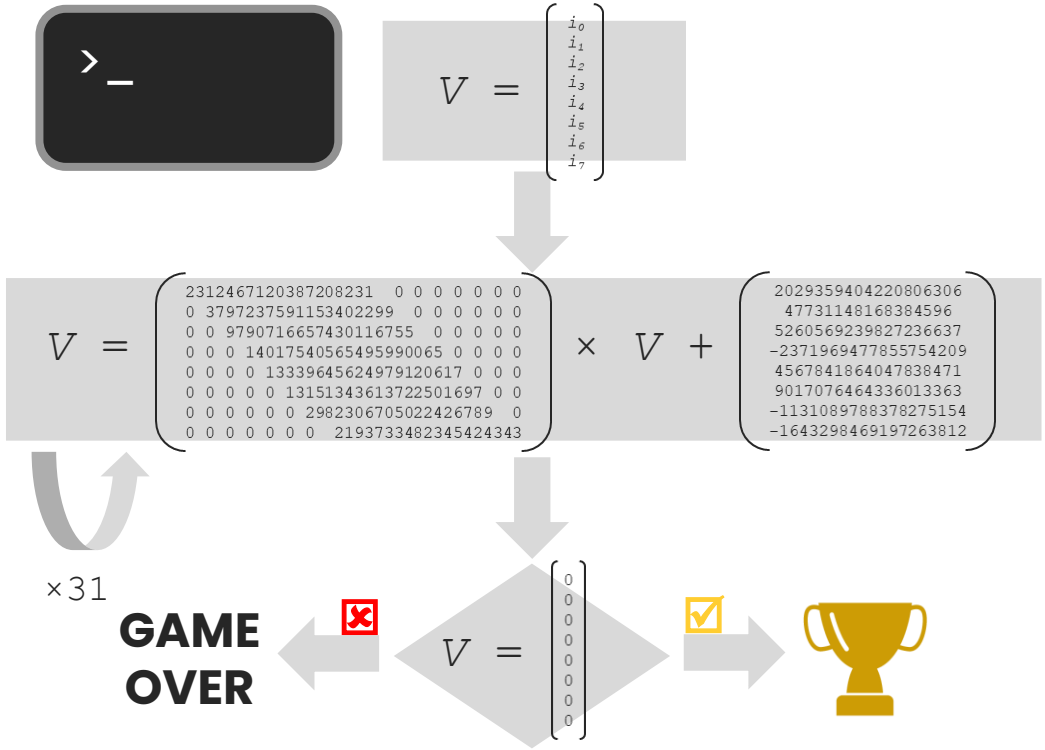
999 859 832 instructions **0.6**

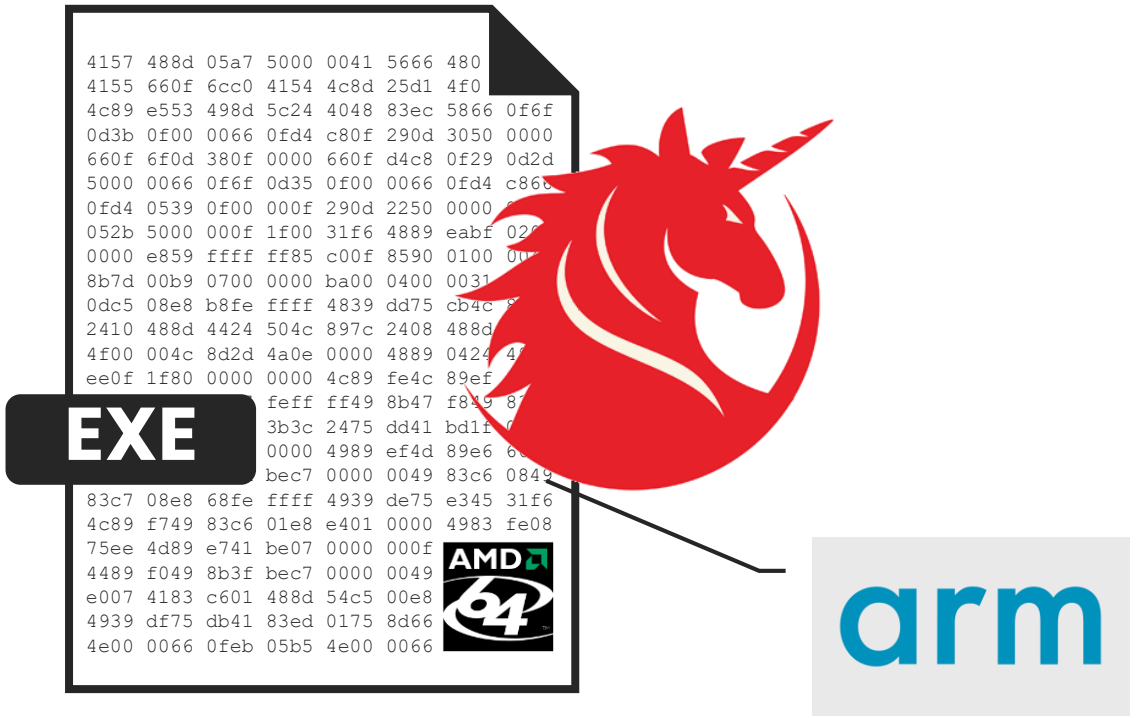
BINSEC

RÉSOLUTION ✓

~3h

Le cas du *France CyberSecurity Challenge* 2022



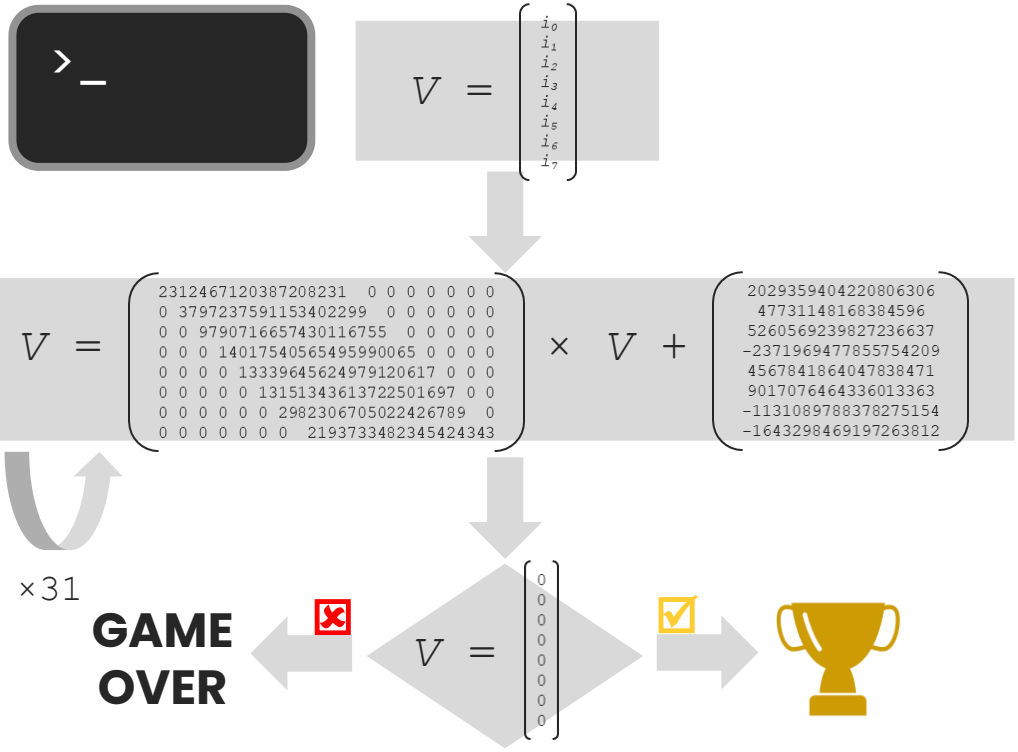


999 859 832 instructions

0.6	**0.8**
RÉSOLUTION ✓	**RÉSOLUTION** ✓
~3h	10m24

BINSEC

Le cas du *France CyberSecurity Challenge* 2022



arm

999 859 832 instructions

0.6

RÉSOLUTION ✓

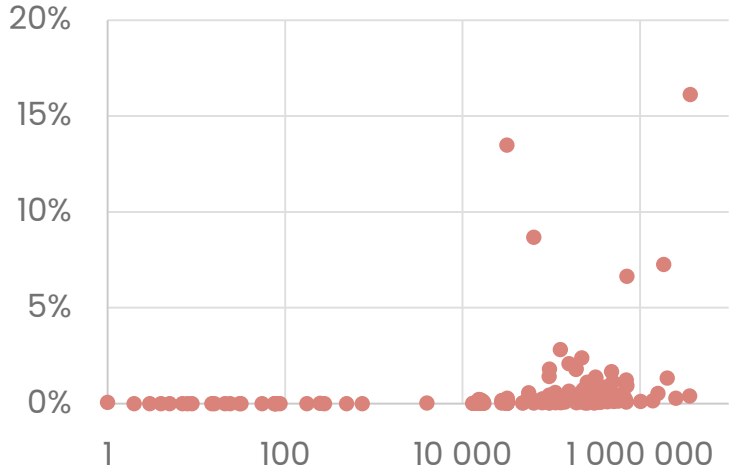
~3h

0.8

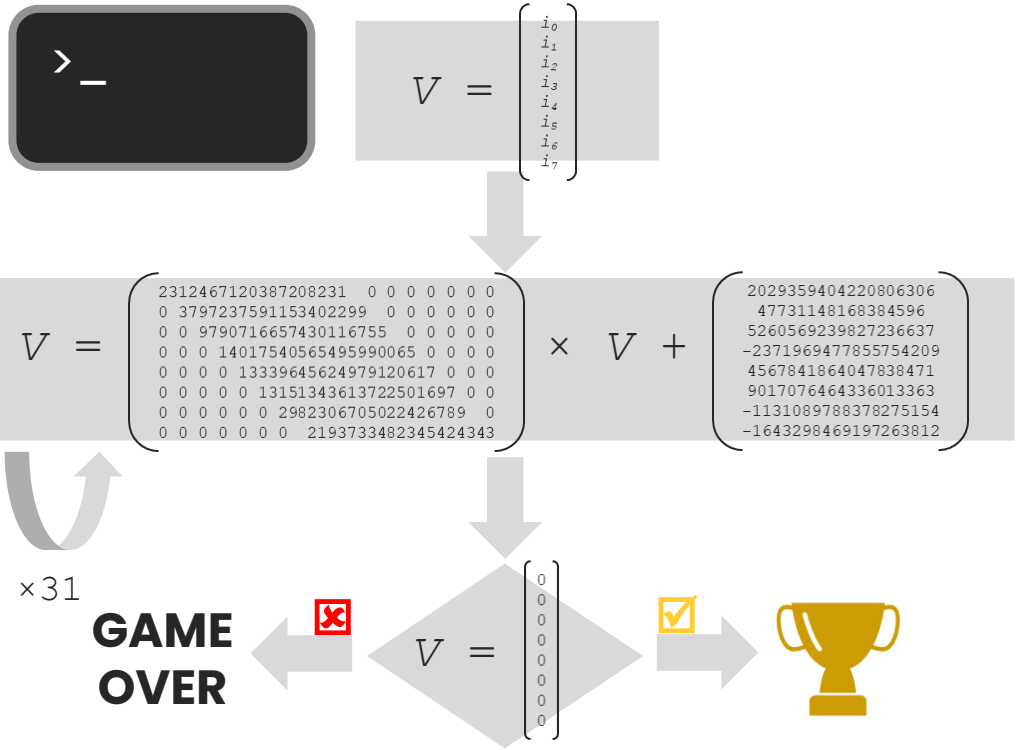
RÉSOLUTION ✓

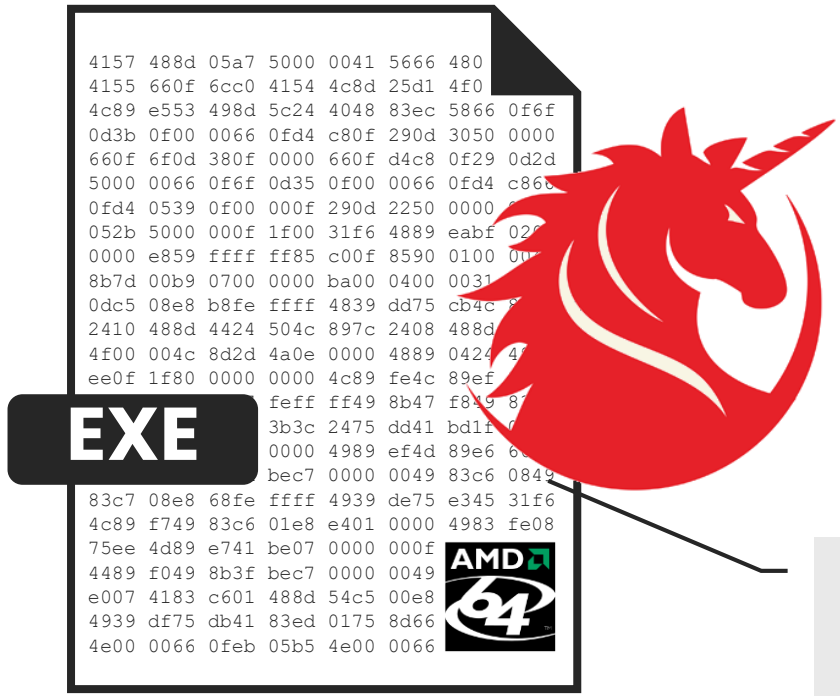
10m24

BINSEC

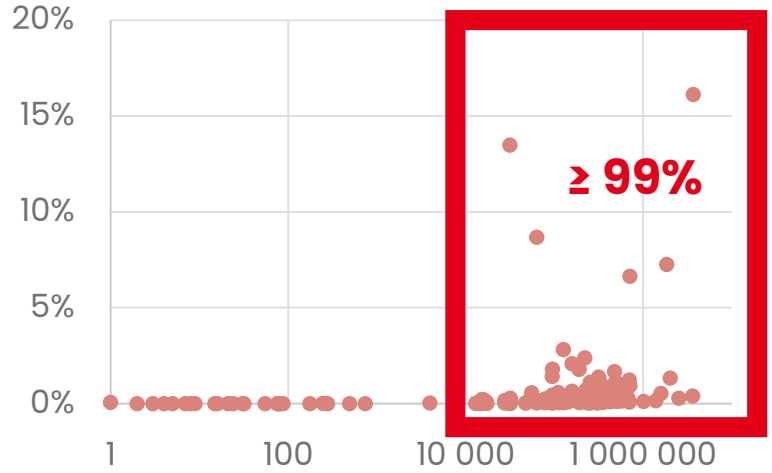


Le cas du *France CyberSecurity Challenge* 2022





arm



999 859 832 instructions

0.6

RÉSOLUTION ✓

~3h

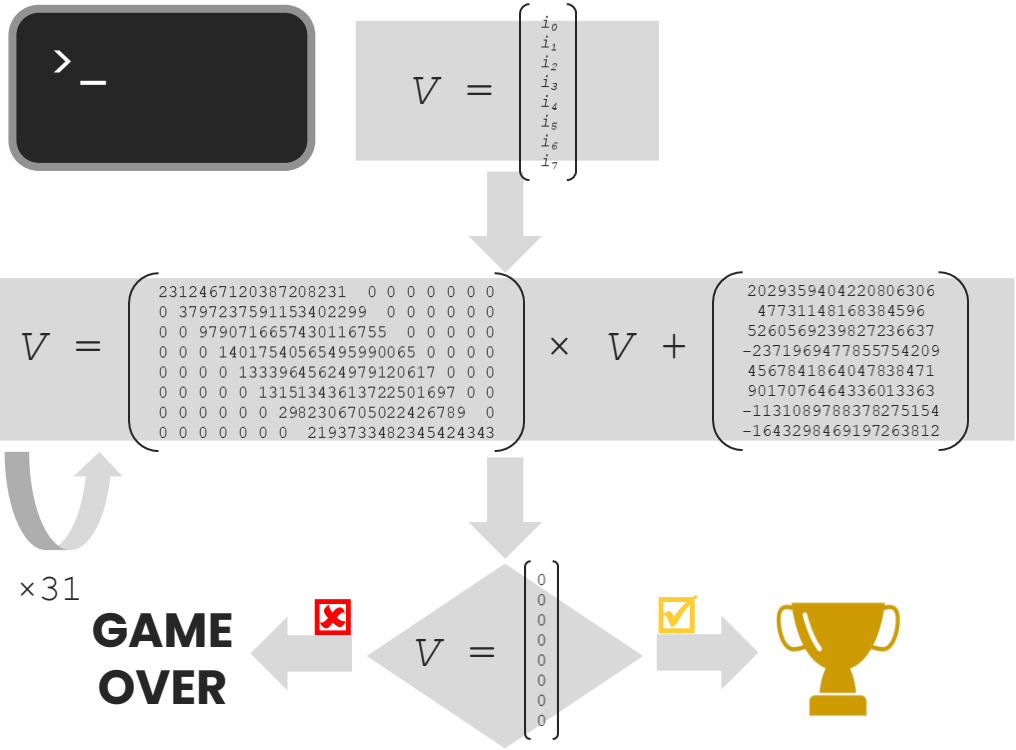
0.8

RÉSOLUTION ✓

10m24

BINSEC

Le cas du *France CyberSecurity Challenge* 2022

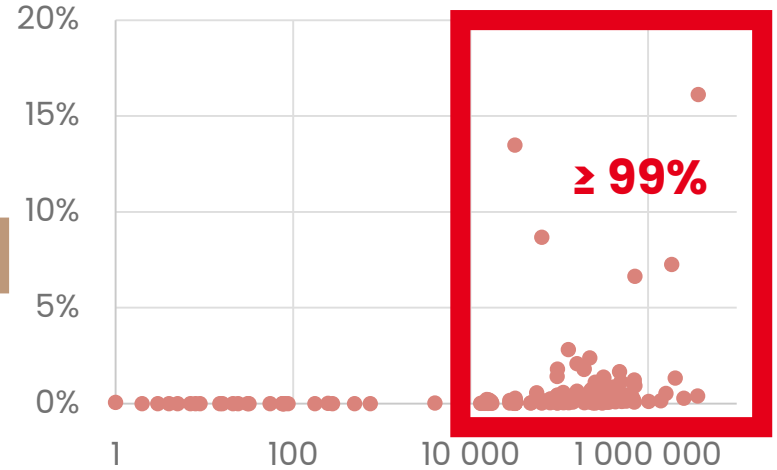


arm

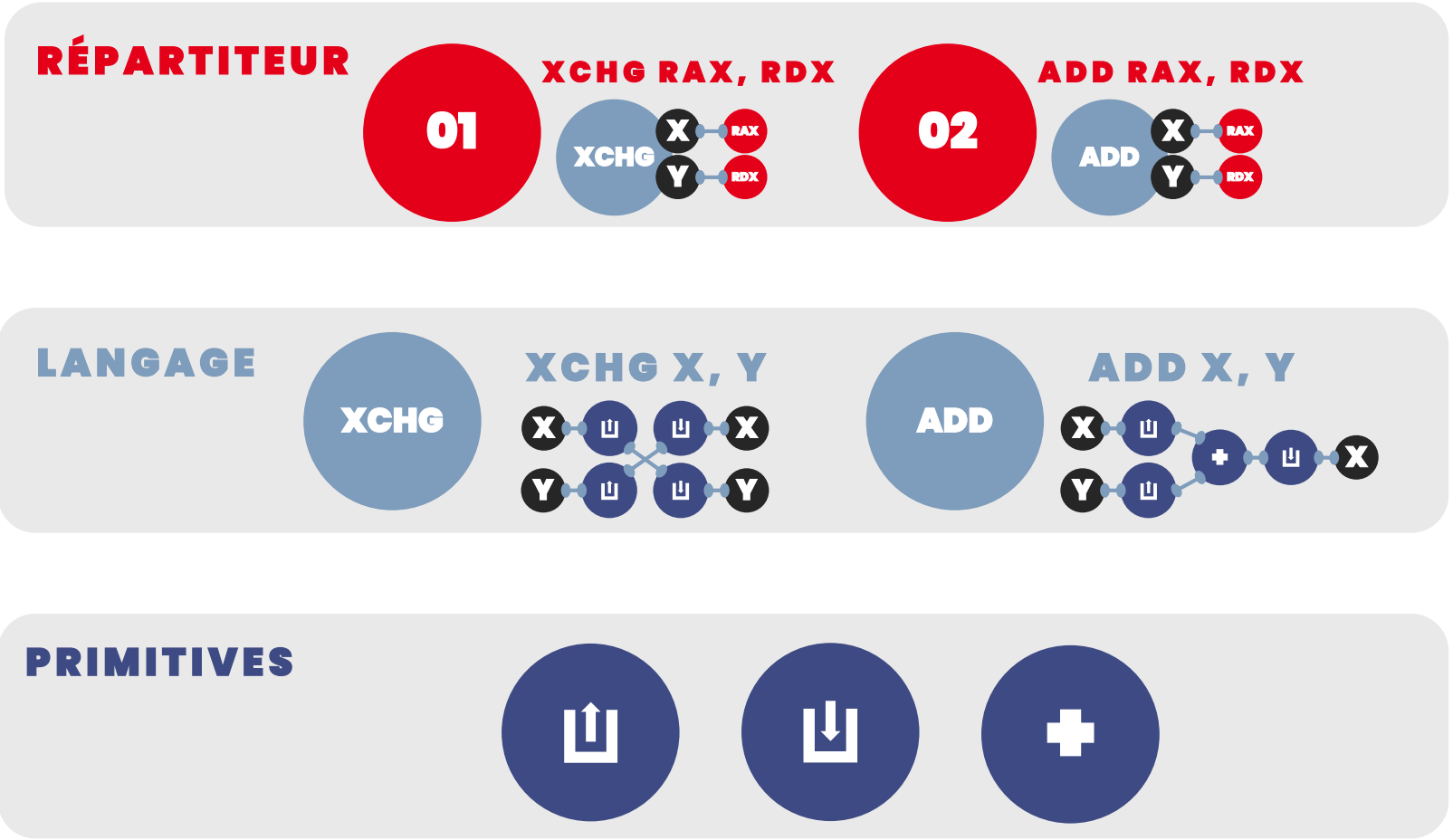
999 859 832 instructions

0.6	0.8	0.8+JIT
RÉSOLUTION ✓	RÉSOLUTION ✓	RÉSOLUTION ✓
~3h	10m24	7m15

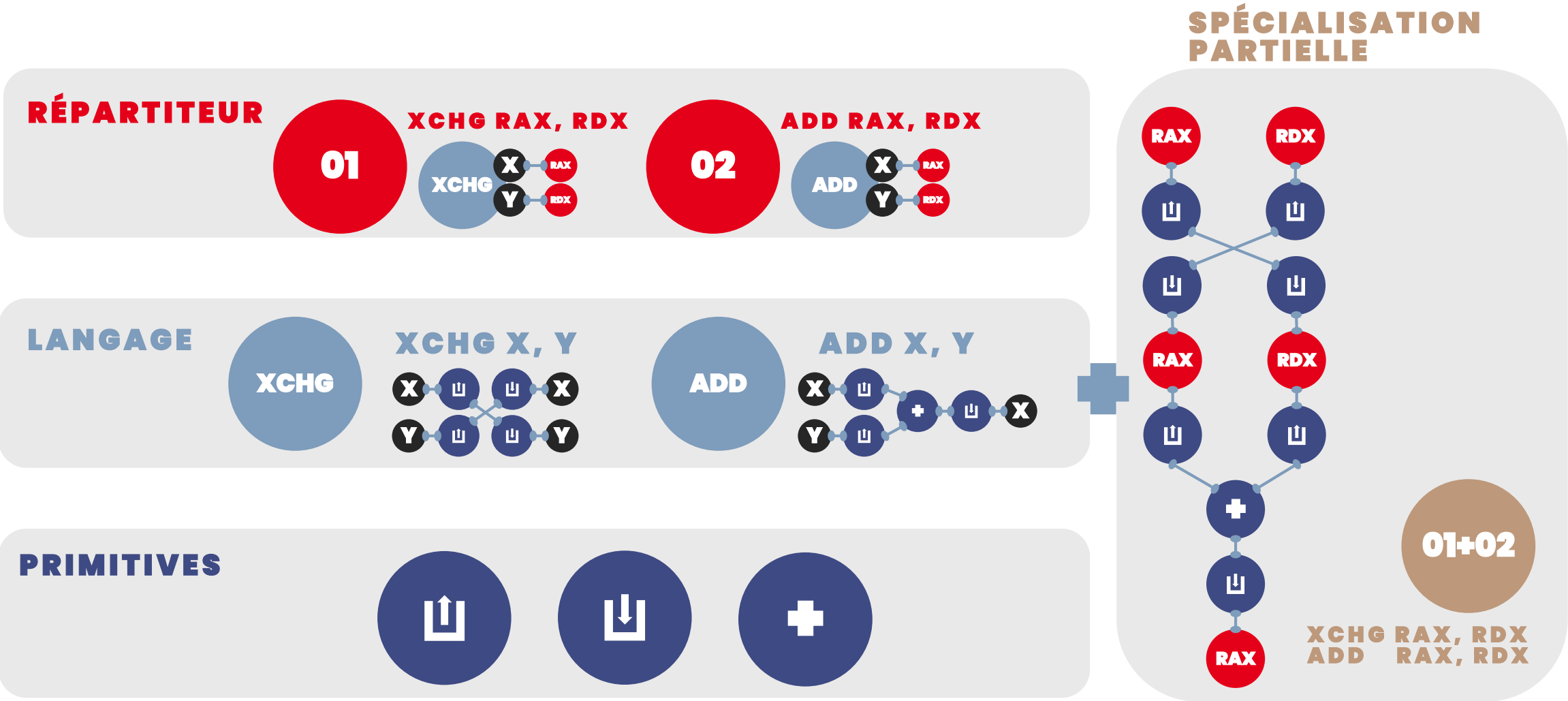
BINSEC



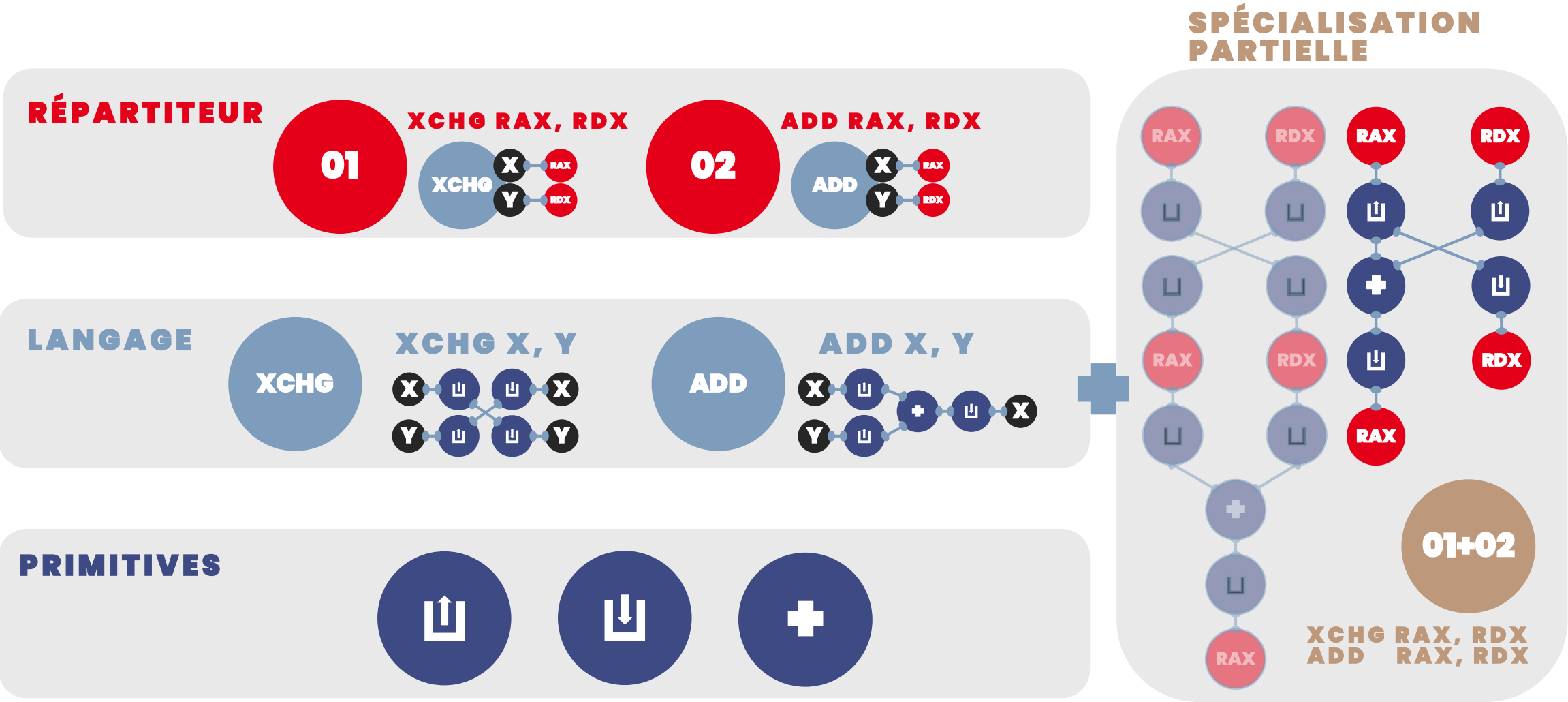
Spécialisation d'un interpréteur



Spécialisation d'un interpréteur



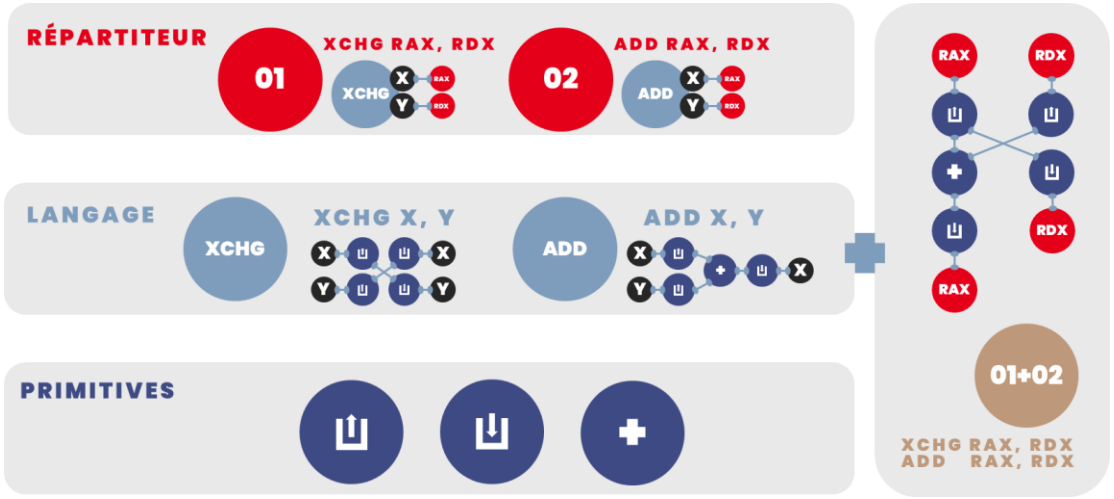
Spécialisation d'un interpréteur



Spécialisation d'un interpréteur en

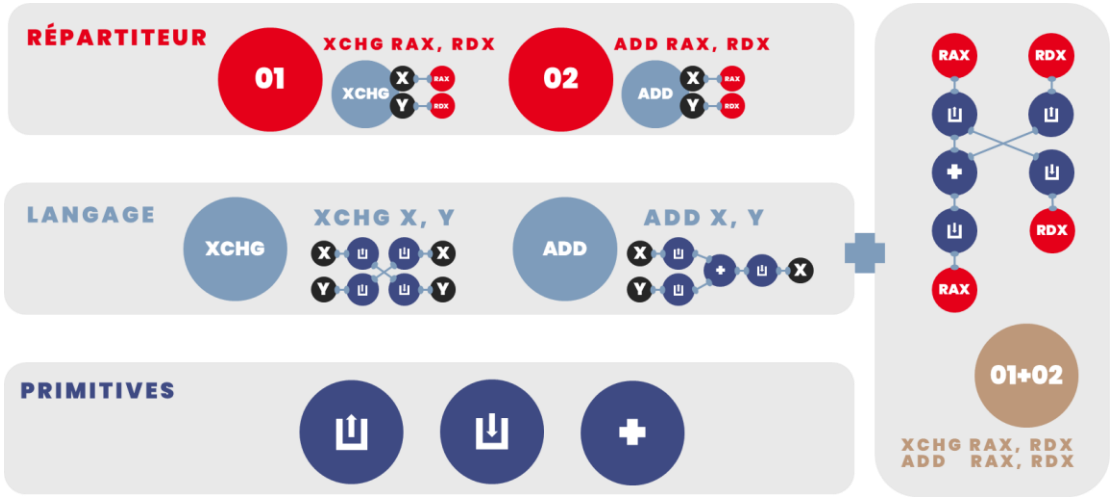
```
let xchg x y registers =
  let vx = read_reg x registers in
  let vy= read_reg y registers in
  let registers = write_reg x vy registers in
  let registers = write_reg y vx registers in
  registers
```

```
let add x y registers =
  let vx = read_reg x registers in
  let vy= read_reg y registers in
  let vp = make_plus vx vy in
  let registers = write_reg x vp registers in
  registers
```



Spécialisation d'un interpréteur en

```
let xchg x y registers =
  let vx = read_reg x registers in
  let vy= read_reg y registers in
  let registers = write_reg x vy registers in
  let registers = write_reg y vx registers in
  registers
```




```
let add x y registers =
  let vx = read_reg x registers in
  let vy= read_reg y registers in
  let vp = make_plus vx vy in
  let registers = write_reg x vp registers in
  registers
```

```
let xchg_rax_rdx_add_rax_rdx registers =
  let va = read_reg Rax registers in
  let vd = read_reg Rdx registers in
  let vp = make_plus va vd in
  let registers = write_reg Rdx va registers in
  let registers = write_reg Rax vp registers in
  registers
```


“Compiler *harmonieusement*, directement depuis OCaml, une sequence d’appels de fonctions ”
Votre serviteur

Solutions existantes ou ressemblantes

	 LLVM	MetaOCaml	OCaml-JIT
Compatibilité 	✗	✓	✓
Autonomie à l'exécutions	N/A	✗	✗
Temps d'émission de code	N/A		⚡
Multi-architectures	N/A	✓	✗

Contributions de l'article

- Conception de la bibliothèque **JITPSI**
  - Compilation à la volée d'une séquence d'appel de fonctions
  - Simplicité, efficacité et compatibilité
- Intégration dans la plateforme BINSEC
  - Accélération de la résolution du défis **licorne** (-30%)

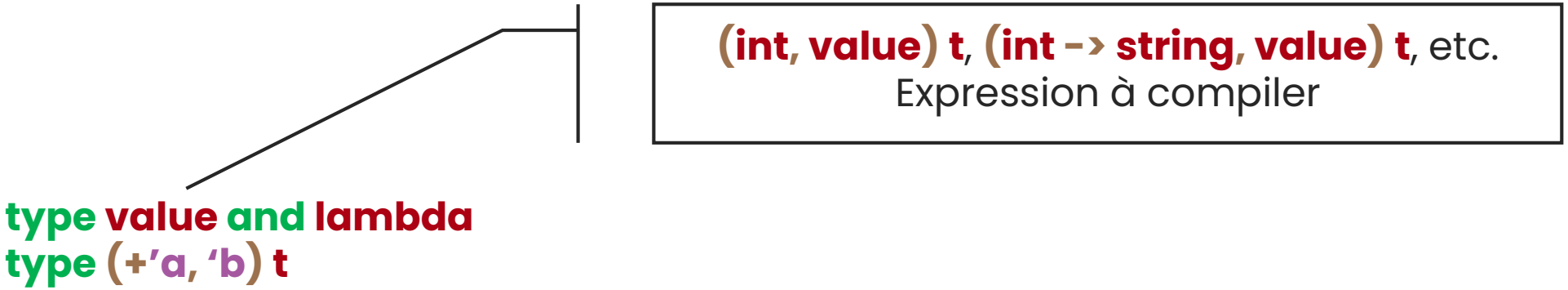
- Code source disponible sur GitHub

JITPSI, une interface sobre mais effective

```
type value and lambda
type (+'a, 'b) t
```

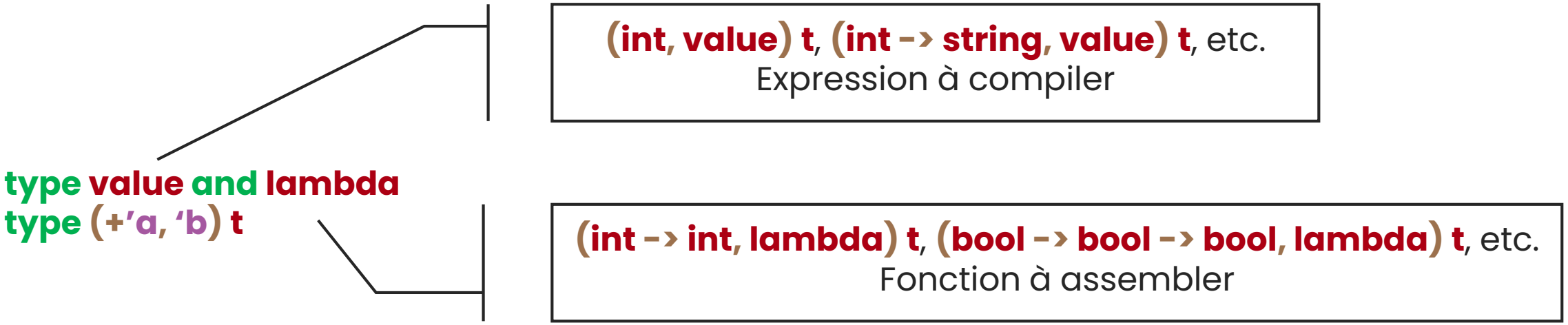
```
val const : 'a -> ('a, value) t

val apply : ('a -> 'b, value) t -> ('a, value) t -> ('b, value) t

val lambda : (('a, value) t -> ('b, 'c) t) -> ('a -> 'b, lambda) t

val return : ('a, lambda) t -> 'a
```

JITPSI, une interface sobre mais effective



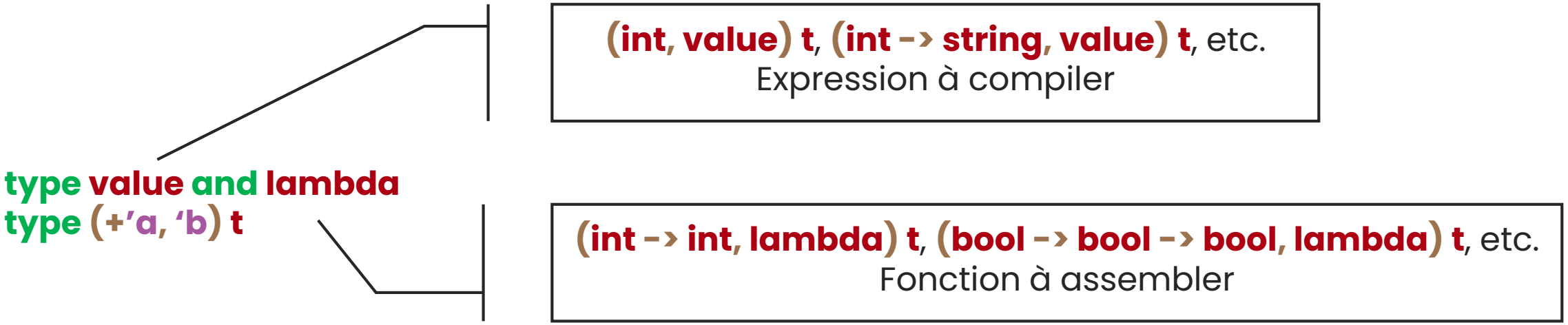

```
val const : 'a -> ('a, value) t

val apply : ('a -> 'b, value) t -> ('a, value) t -> ('b, value) t

val lambda : (('a, value) t -> ('b, 'c) t) -> ('a -> 'b, lambda) t

val return : ('a, lambda) t -> 'a
```

JITPSI, une interface sobre mais effective




```
val const : 'a -> ('a, value) t

val apply : ('a -> 'b, value) t -> ('a, value) t -> ('b, value) t

val lambda : (('a, value) t -> ('b, 'c) t) -> ('a -> 'b, lambda) t

val return : ('a, lambda) t -> 'a
```

JITPSI, une interface sobre mais effective



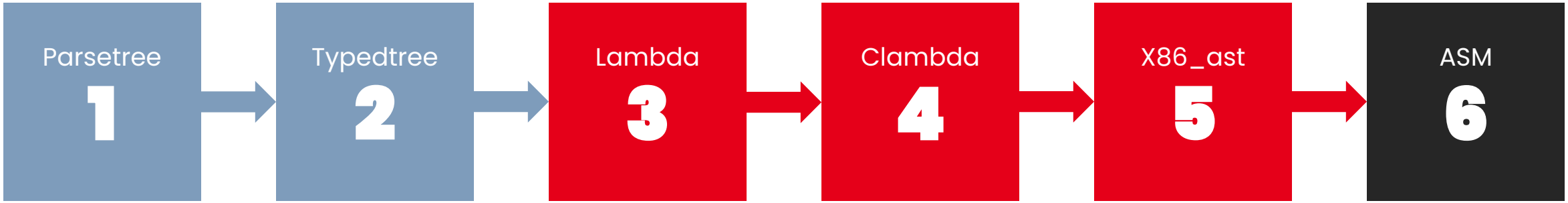

```
val const : 'a -> ('a, value) t

val apply : ('a -> 'b, value) t -> ('a, value) t -> ('b, value) t

val lambda : (('a, value) t -> ('b, 'c) t) -> ('a -> 'b, lambda) t

val return : ('a, lambda) t -> 'a
```

```
let apply2 f x y = apply (apply f x) y

let lambda2 body =
  lambda (fun x -> lambda (body x))
```

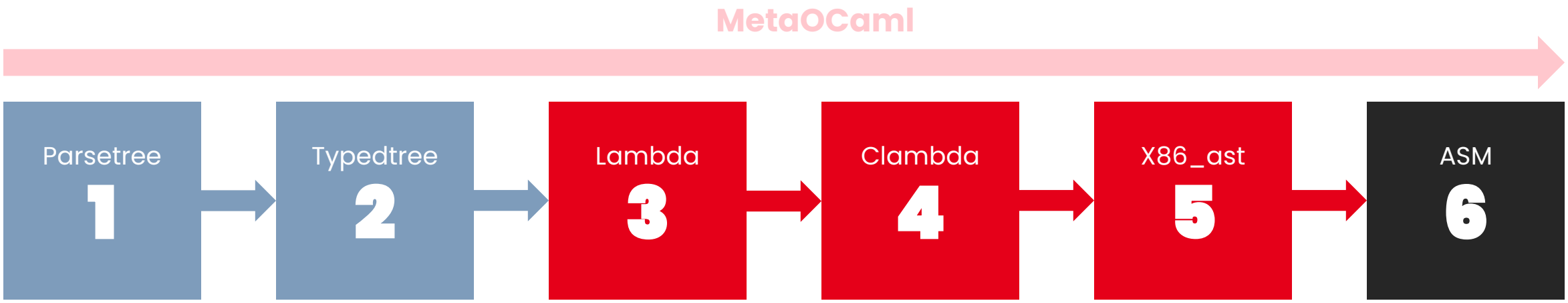
Pour l'exemple

```
let one : unit -> int = return
    (lambda (fun _ -> const 1))

let identity : 'a -> 'a = return
    (lambda (fun x -> x))

let curry (f : 'a -> 'b -> 'c) (x : 'a) : 'b -> 'c = return
    (lambda (fun y -> apply2 (const f) (const x) y))

let flip (f : 'a -> 'b -> 'c) : 'b -> 'a -> 'c = return
    (lambda2 (fun x y -> apply2 (const f) y x))
```

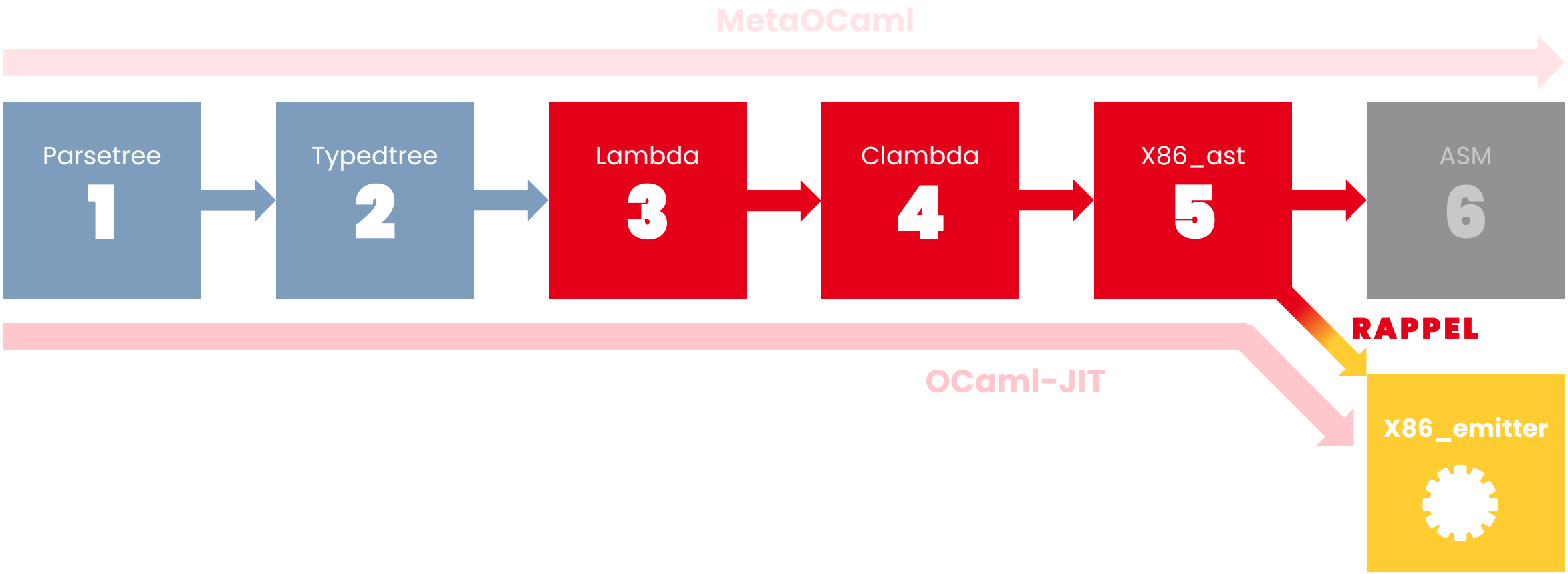
Positionnement de JITPSI



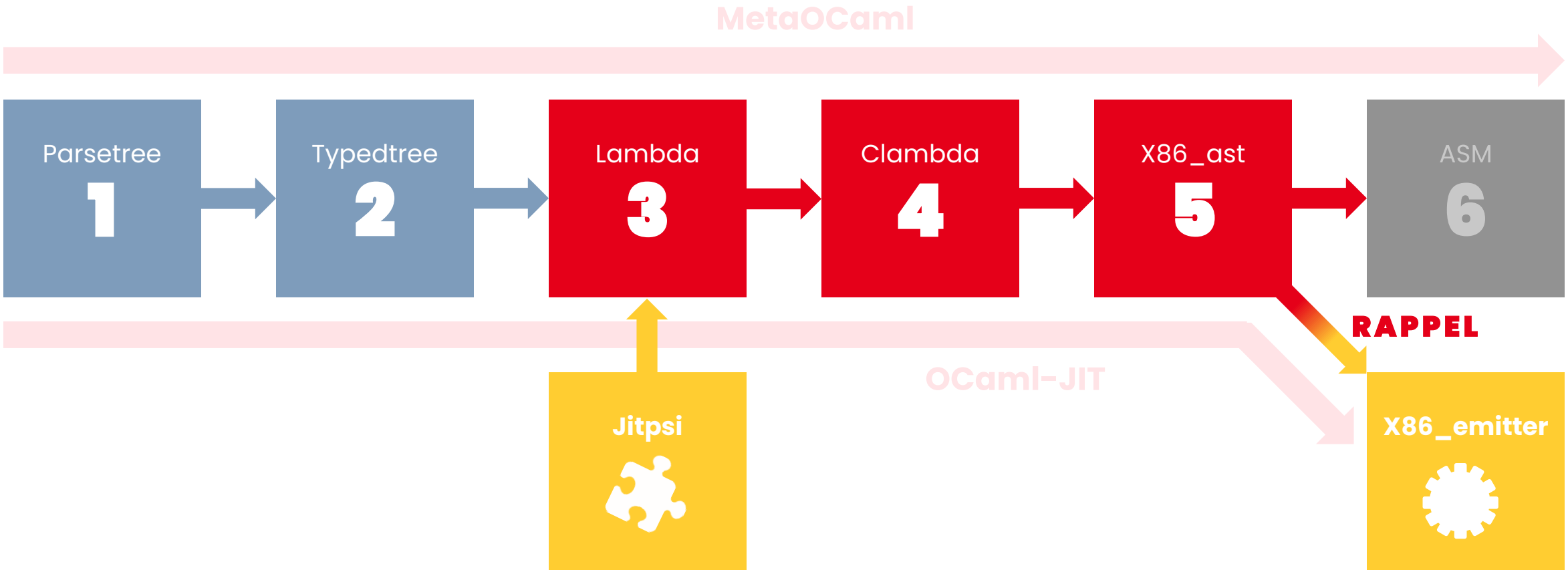
Positionnement de JITPSI



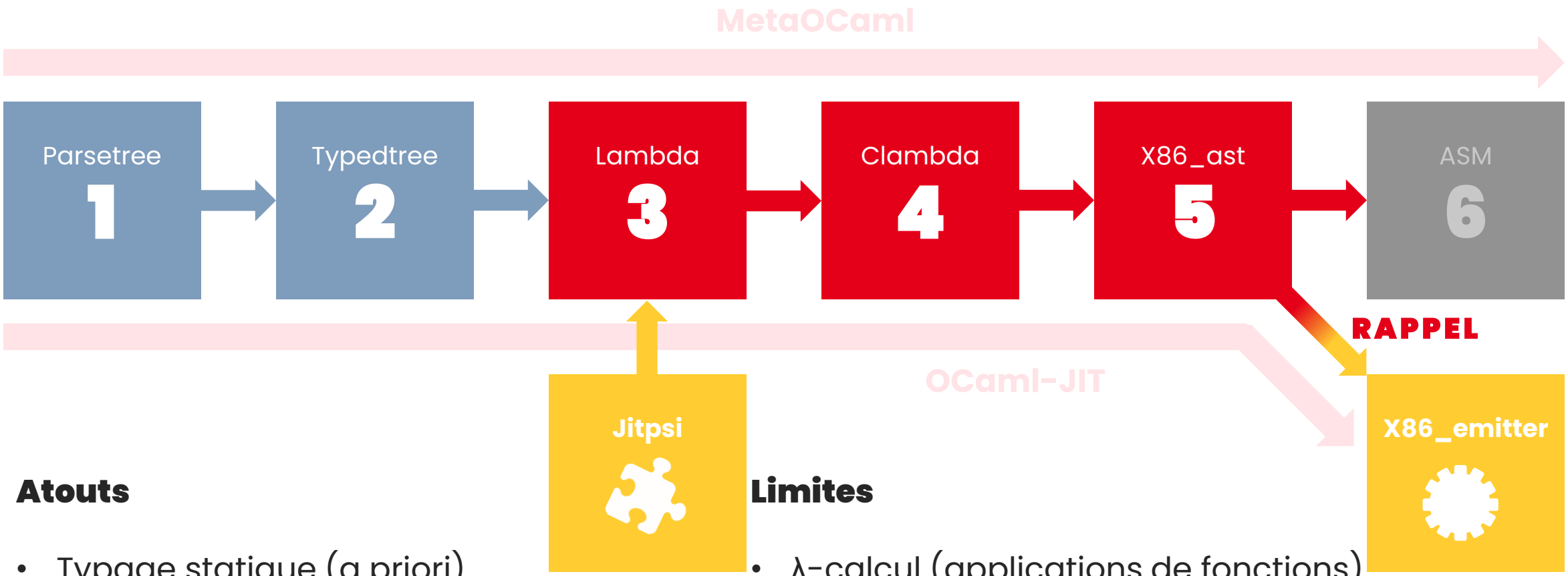
Positionnement de JITPSI



Positionnement de JITPSI



Positionnement de JITPSI



Atouts

- Typage statique (a priori)
- Indépendance à l'exécution
- Émission de code en mémoire
- *Collectable* par le GC

Limites

- λ-calcul (applications de fonctions)
  - Absence de structures de contrôle
  - Absence de primitives de bases

Récapitulatif final

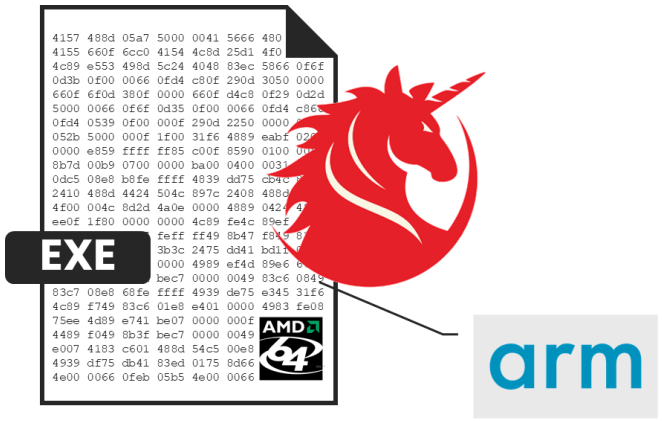
	 LLVM	MetaOCaml	OCaml-JIT	JITPSI
Compatibilité	✗	✓	✓	✓
Autonomie à l'exécutions	N/A	✗	✗	✓
Expressivité	N/A	✓	✓	✗
Temps d'émission de code	N/A	⏳	⚡	⚡
Multi-architectures	N/A	✓	✗	✗

Merci pour votre attention

Frédéric Recoules Sébastien Bardin

- Conception de la bibliothèque **JITPSI**
  - Compilation à la volée d'une séquence d'appel de fonctions
  - Simplicité, efficacité et compatibilité
- Optimisation d'interpréteur « *au sens large* »
  (séquence d'instructions répétées)
  - Émulateurs, Machines virtuelles
  - Exécution symbolique (BINSEC)
  - Interprétation abstraite ?
  - Autres ?
- Code source disponible sur GitHub

France CyberSecurity Challenge 2022



La main est à l'audience

Qui est qui, JITPSI ou code OCaml ?

```
sub     rsp, 18h
mov     [rsp+8], rbx
mov     rdi, [rbx+16]
mov     rbx, rax
mov     [rsp], rbx
mov     eax, 5
call    camlStdlib__Map__find_440
call    camlStdint__pred_4052
mov     rbx, rax
mov     rax, [rsp+8]
mov     rsi, [rax+32]
mov     eax, 5
mov     rdi, [rsp]
add     rsp, 18h
jmp     camlStdlib__Map__add_428
```

```
sub     rsp, 18h
mov     rbx, rax
mov     [rsp], rbx
cmp     r15, [r14]
jbe     1f
0:  lea     rax, camlLibexec
mov     rax, [rax]
mov     rdi, [rax+18h]
mov     [rsp+8], rdi
mov     rdi, [rax+d8h]
mov     eax, 5
call    caml_apply2
call    camlStdint__pred_4052
mov     rbx, rax
mov     eax, 5
mov     rdi, [rsp]
mov     rsi, [rsp+8]
add     rsp, 18h
jmp     caml_apply3
1:  call    caml_call_gc
jmp     0b
```