Picon : Control Flow Integrity on LLVM

Thomas Coudray, Arnaud Fontaine, Pierre Chifflier

thomas.coudray.fr@gmail.com,{pierre.chifflier,arnaud.fontaine}@ssi.gouv.fr

3 juin 2015

CFI : quesaquo ?

Le *Control Flow Integrity* protège des attaques visant à détourner le flot d'exécution « normal » d'un programme

- S'assure que le programme suit un chemin légitime prédéterminé
- À l'exécution, tout chemin hors du graphe est une compromission
- Cas pratiques : ROP, Ret-to-libc
- Intégrité (pas confidentialité)
- Hypothèse : système durci, binaires intègres
- Plus de détails dans l'article

SSTIC

Cette présentation peut contenir de l'assembleur **et** des maths.

Il a CFI, il a tout compris

- Déjà fait / À la mode ?
  - *Control Flow Guard* (Windows 10)
  - Forward-Edge CFI (Clang)
  - Code Pointer Integrity / Code Pointer Separation
  - Control Flow Locking, BinCFI, mcCFI, CCFIR, KCoFI, cfi-vptr, …
  - Incomplets (*cf* article)

⇒ Tout s'appelle CFI !

Un programme (particulièrement) vulnérable

```
1  void foo(char *s) {
2    char c[12] ;
3    strcpy(c, s) ;
4  }
5
6  int bar(char *s) {
7    printf("[%d]\n", strlen(s)) ;
8    return 0 ;
9  }
10
11 int main(int argc, char *argv[]) {
12   void (*func)(char *) ;
13   func = foo ;
14   (*func)(argv[1]) ;
15
16   bar(argv[1]) ;
17
18   return 0 ;
19 }
```

Points d'intérêts

```
1  void foo(char *s) {
2    char c[12] ;
3    strcpy(c, s) ;
4  }
5
6  int bar(char *s) {
7    printf("[%d]\n", strlen(s)) ;
8    return 0 ;
9  }
10
11 int main(int argc, char *argv[]) {
12   void (*func)(char *) ;
13   func = foo ;
14   (*func)(argv[1]) ;
15
16   bar(argv[1]) ;
17
18   return 0 ;
19 }
```

Appel

(et stack overflow …)

Retour

Appel dynamique

ROP around the clock

- Réutilisation de code
- *gadget* : suite d'instructions terminée par un retour (`ret` sur `x86`)
- *shellcode* : chaînage de gadgets
- Attaques sur **données**, pas sur code
- Marche même si NX ou ASLR

Exemple de gadgets :

```
pop eax
ret

mov eax, ebx
ret
```

Exemple de Forward-Edge CFI

```
1  void foo(char *s) {
2    char c[12] ;
3     strcpy(c, s) ;
4  }
5
6  int bar(char *s) {
7    printf("[%d]\n", strlen(s)) ;
8     return 0 ;
9  }
10
11 int main(int argc, char *argv[]) {
12   void (*func)(char *) ;
13   func = foo ;
14    (*func)(argv[1]) ;
15
16   bar(argv[1]) ;
17
18   return 0 ;
19 }
```

- Protection des appels indirects
- Exemples : Windows 10 CFG, Clang 3.7 CFI

Exemple de Forward-Edge CFI

```
1  void foo(char *s) {
2    char c[12] ;
3     strcpy(c, s) ;
4  }
5
6  int bar(char *s) {
7    printf("[%d]\n", strlen(s)) ;
8     return 0 ;
9  }
10
11 int main(int argc, char *argv[]) {
12   void (*func)(char *) ;
13   func = foo ;
14    (*func)(argv[1]) ;
15
16   bar(argv[1]) ;
17
18   return 0 ;
19 }
```

Appel
(non protégé)

Retour
(non protégé)

Appel dynamique
Vérif. intégrité

Exemple : intégrité des retours

```
1  void foo(char *s) {
2    char c[12] ;
3     strcpy(c, s) ;
4   }
5
6  int bar(char *s) {
7    printf("[%d]\n", strlen(s)) ;
8     return 0 ;
9   }
10
11 int main(int argc, char *argv[]) {
12   void (*func)(char *) ;
13   func = foo ;
14    (*func)(argv[1]) ;
15
16   bar(argv[1]) ;
17
18   return 0 ;
19 }
```

- Protection des retours (vérification d'un identifiant)
- Exemples : BinCFI, Control Flow Locking

Exemple : intégrité des retours

```
1  void foo(char *s) {
2    char c[12] ;
3    strcpy(c, s) ;
4  }
5
6  int bar(char *s) {
7    printf("[%d]\n", strlen(s)) ;
8    return 0 ;
9  }
10
11 int main(int argc, char *argv[]) {
12   void (*func)(char *) ;
13   func = foo ;
14   (*func)(argv[1]) ;
15
16   bar(argv[1]) ;
17
18   return 0 ;
19 }
```

Appel
(non protégé)

Retour
(Vérifié avant le retour)

Appel dynamique
(non protégé)

Fermer le ROP-inet

Existant

- Protections intéressantes, mais incomplètes
- Aucune protection contre le *Jump-Oriented Programming* (JOP)
- Motivation principale : performances

Modèle qui privilégie la sécurité

Modèle le plus complet possible

- Protéger tous les appels de fonctions
- Protéger tous les retours de fonctions
- (optionnel) Protéger tous les branchements

Compromission du CFI

- Détecter « dès que possible » de manière fiable et robuste
- Réagir en terminant le programme compromis
- Disposer d'informations pour une analyse forensique

Décourager le ROP : le modèle grec I

Un moniteur $\mathcal{M}$ est un automate à pile $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ tel que

$$
\begin{aligned}
Q &= \{\text{InFunction}, \text{ExpectCall}, \text{ExpectReturn}, \text{ExpectJump}\} \\
\Sigma &= \{\texttt{cfiCall}\ f, \texttt{cfiEnter}\ f \mid f \in V_{\mathcal{P}}\} \\
&\quad \cup \{\texttt{cfiExit}\ f, \texttt{cfiReturned}\ f \mid f \in V_{\mathcal{P}}\} \\
&\quad \cup \{\texttt{cfiBeforeJump}\ (f, b), \texttt{cfiAfterJump}\ (f, b) \mid f \in V_{\mathcal{P}}, b \in V_f\} \\
\Gamma &= \{\langle f, \sigma \rangle \mid f \in V_{\mathcal{P}}, \sigma \in V_f^*\} \\
q_0 &= \text{ExpectCall} \\
Z_0 &= \langle \texttt{main}, \text{entry}(\texttt{main}) \rangle \\
F &= \{\text{ExpectReturn}\}
\end{aligned}
$$

Décourager le ROP : le modèle grec II

$$\delta(\text{InFunction}, \texttt{cfiCall}\ f', \langle f, b\sigma\rangle) =$$
$$\begin{cases} \{(\text{ExpectCall}, \langle f', \text{entry}(f')\rangle\langle f, b\sigma\rangle) \mid (f, f') \in E_{\mathcal{P}}\} & \text{iff } b \in B_{\mathcal{P}}(f, f') \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta(\text{ExpectReturn}, \texttt{cfiReturned}\ f', \langle f, b\sigma\rangle) =$$
$$\begin{cases} \{(\text{InFunction}, \langle f, b\sigma\rangle) \mid (f, f') \in E_{\mathcal{P}}\} & \text{iff } b \in B_{\mathcal{P}}(f, f') \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta(\text{InFunction}, \texttt{cfiExit}\ f, \langle f, b\rangle) = \begin{cases} \{(\text{ExpectReturn}, \epsilon)\} & \text{iff } b \in \text{exit}(f) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta(\text{ExpectCall}, \texttt{cfiEnter}\ f, \langle f, b\sigma\rangle) = \begin{cases} \{(\text{InFunction}, \langle f, b\sigma\rangle)\} & \text{iff } b = \text{entry}(f) \\ \emptyset & \text{otherwise} \end{cases}$$

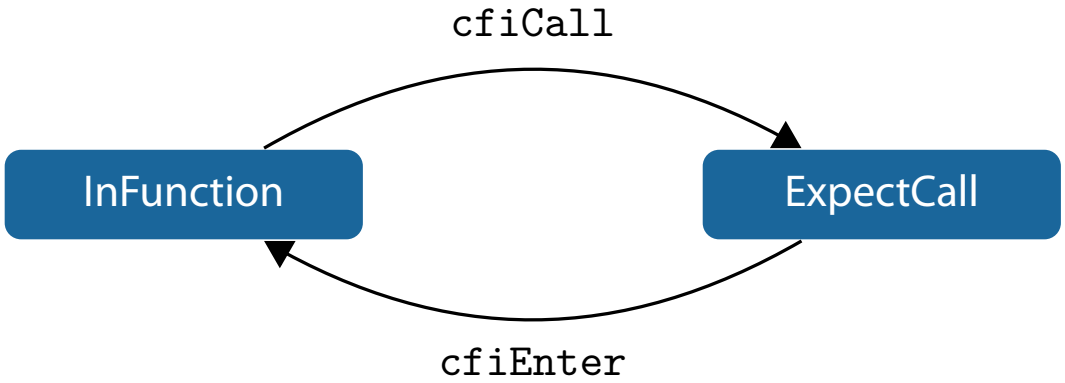
$$\delta(\text{InFunction}, \texttt{cfiBeforeJump}\ (f, b), \langle f, b\sigma\rangle) = \{(\text{ExpectJump}, \langle f, b\sigma\rangle)\}$$

$$\delta(\text{ExpectJump}, \texttt{cfiAfterJump}\ (f, b'), \langle f, b\sigma\rangle) = \{(\text{InFunction}, \langle f, b'b\sigma\rangle) \mid (b, b') \in E_f\}$$

Le moniteur dans tous ses états

InFunction

Le moniteur dans tous ses états



Instrumentation

```
1  void foo(char *s) {
2
3    char c[12] ;
4
5    strcpy(c, s) ;
6
7
8  }
9
10 int bar(char *s) {
11
12
13
14
15   printf("[%d]\n", strlen(s)) ;
16
17
18   return 0 ;
19 }
```

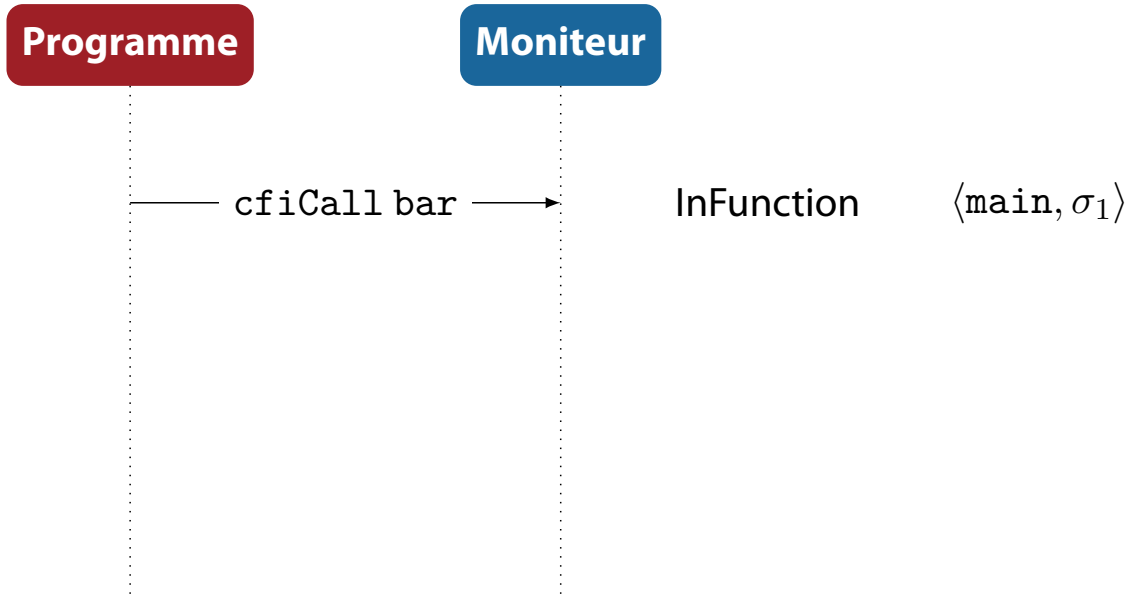
```
1  int main(int argc, char *argv[]) {
2
3    void (*func)(char *) ;
4    func = foo ;
5
6    (*func)(argv[1]) ;
7
8
9    bar(argv[1]) ;
10
11
12   return 0 ;
13 }
```

Instrumentation

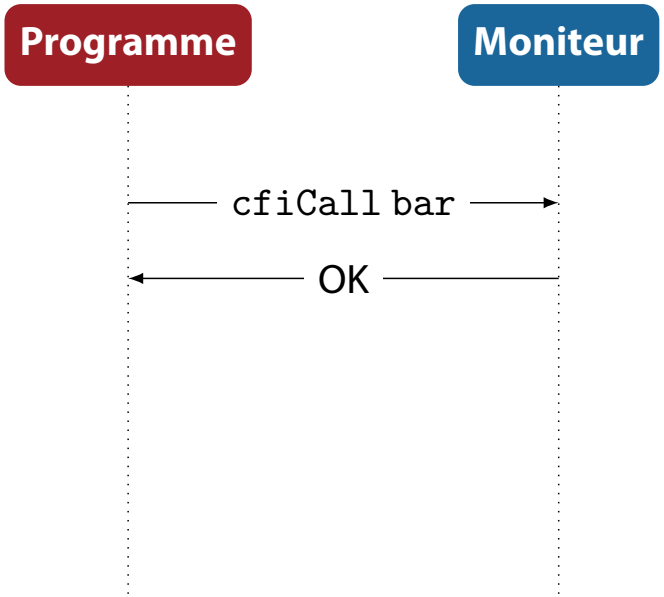
```
1  void foo(char *s) {
2    cfiEnter(foo) ;
3    char c[12] ;
4    cfiCall(strcpy) ;
5    strcpy(c, s) ;
6
7
8  }
9
10 int bar(char *s) {
11   cfiEnter(bar) ;
12   cfiCall(strlen) ;
13
14   cfiCall(printf) ;
15   printf("[%d]\n", strlen(s)) ;
16
17
18   return 0 ;
19 }
```

```
1  int main(int argc, char *argv[]) {
2    cfiEnter(main) ;
3    void (*func)(char *) ;
4    func = foo ;
5    cfiCall(foo) ;
6    (*func)(argv[1]) ;
7
8    cfiCall(bar) ;
9    bar(argv[1]) ;
10
11
12   return 0 ;
13 }
```

Exemple


Programme
Moniteur
cfiCall(bar) ;
cfiCall bar
InFunction
⟨main, σ1⟩

Exemple

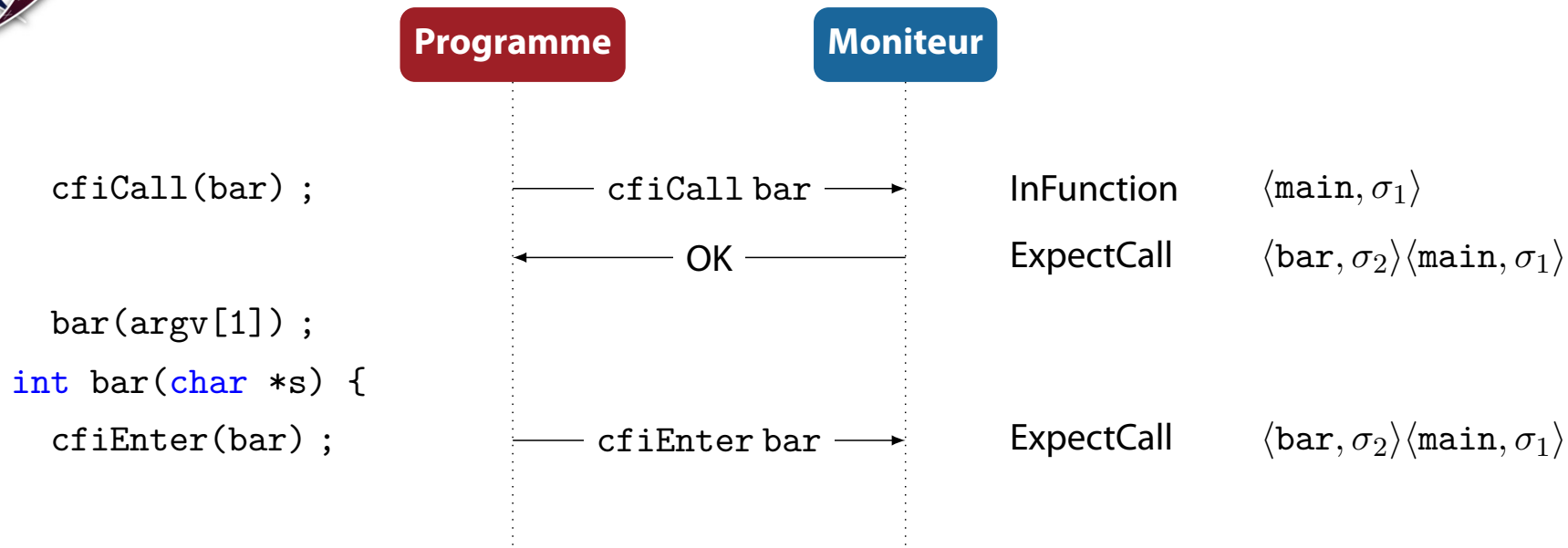



```
cfiCall(bar) ;
```

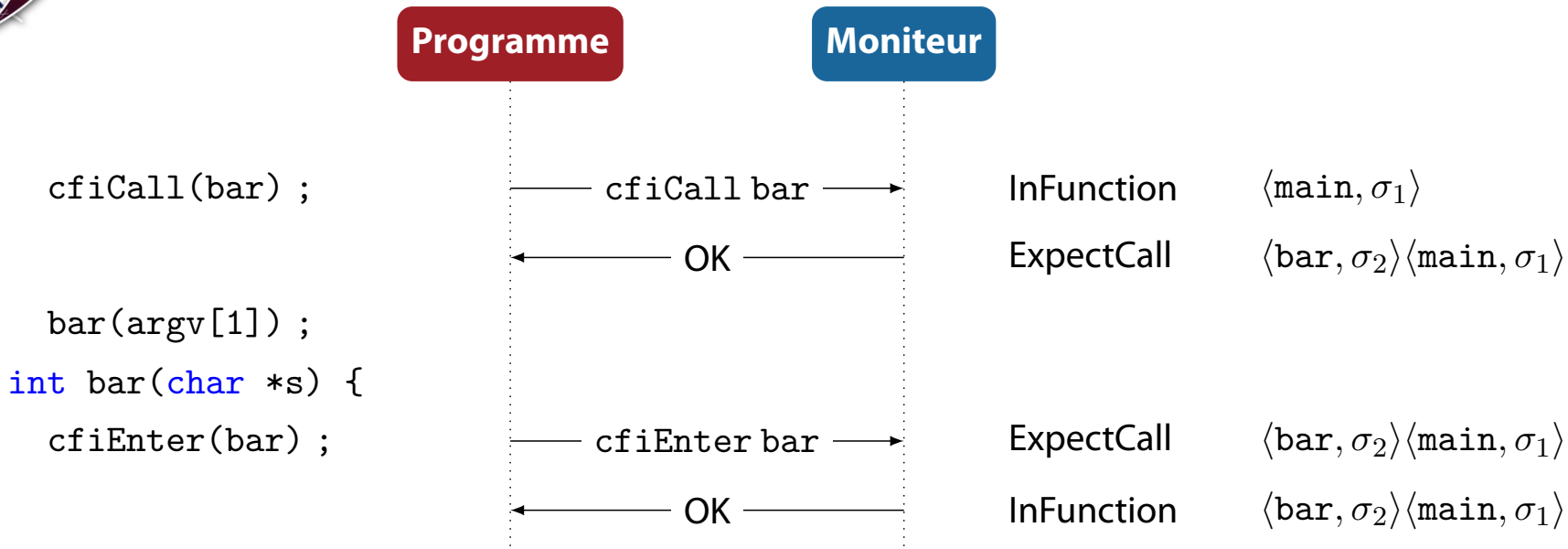
InFunction $\langle \texttt{main}, \sigma_1 \rangle$

ExpectCall $\langle \texttt{bar}, \sigma_2 \rangle \langle \texttt{main}, \sigma_1 \rangle$

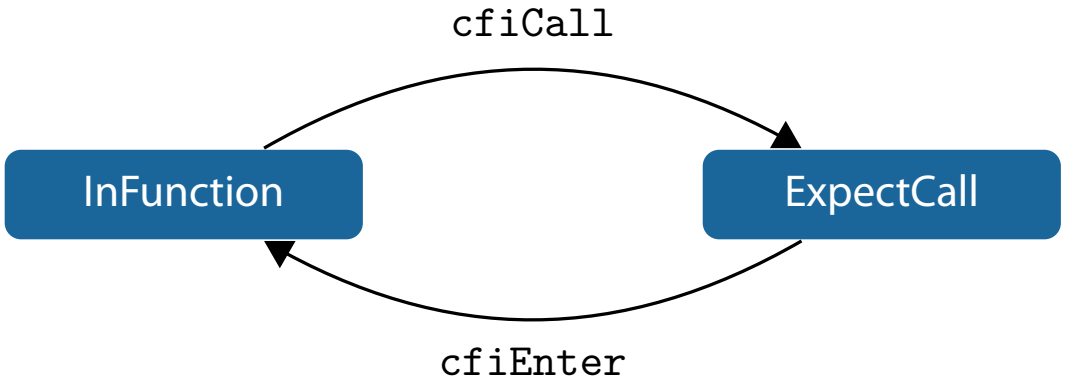
Exemple


Programme
Moniteur
cfiCall(bar) ;
cfiCall bar
InFunction
⟨main, σ1⟩
OK
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩
bar(argv[1]) ;
int bar(char *s) {
cfiEnter(bar) ;
cfiEnter bar
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩

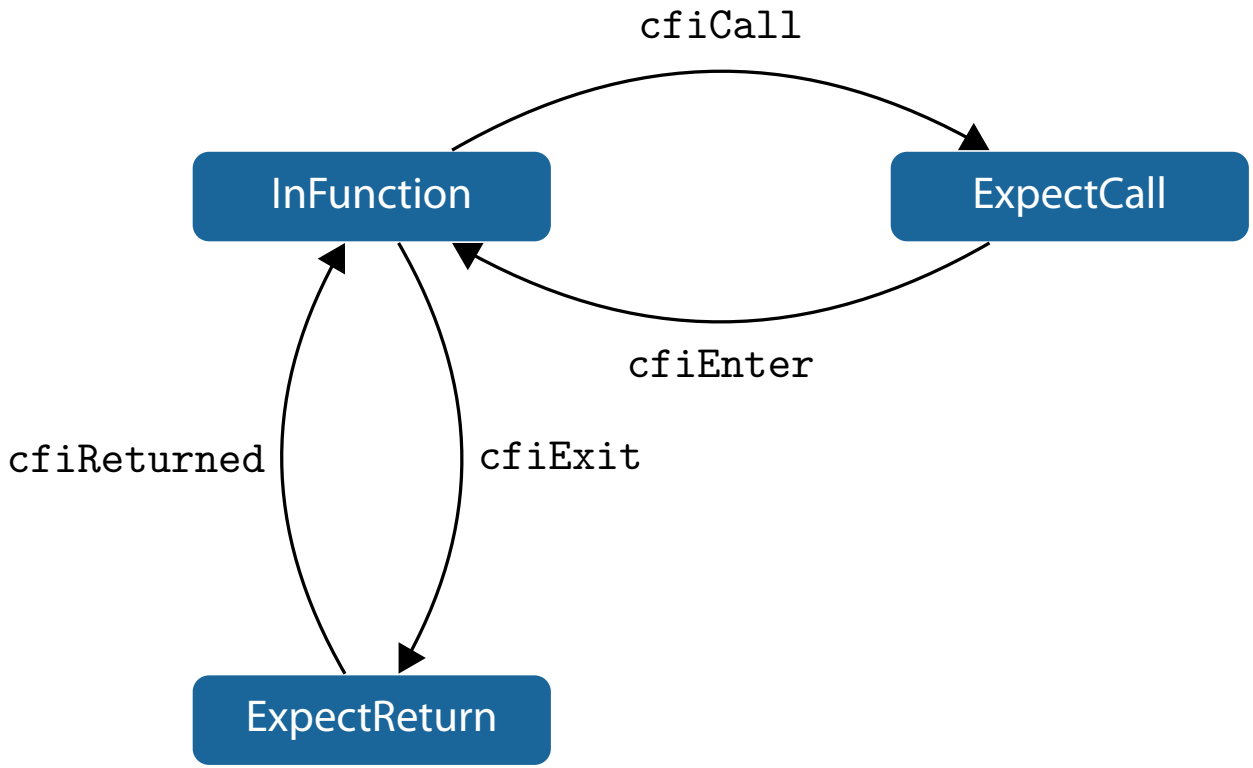
Exemple


Programme
Moniteur
cfiCall(bar) ;
cfiCall bar
InFunction
⟨main, σ1⟩
OK
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩
bar(argv[1]) ;
int bar(char *s) {
cfiEnter(bar) ;
cfiEnter bar
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩
OK
InFunction
⟨bar, σ2⟩⟨main, σ1⟩

Le moniteur dans tous ses états



Le moniteur dans tous ses états



Instrumentation

```
1  void foo(char *s) {
2    cfiEnter(foo) ;
3    char c[12] ;
4    cfiCall(strcpy) ;
5    strcpy(c, s) ;
6
7
8  }
9
10 int bar(char *s) {
11   cfiEnter(bar) ;
12   cfiCall(strlen) ;
13
14   cfiCall(printf) ;
15   printf("[%d]\n", strlen(s)) ;
16
17
18   return 0 ;
19 }
```

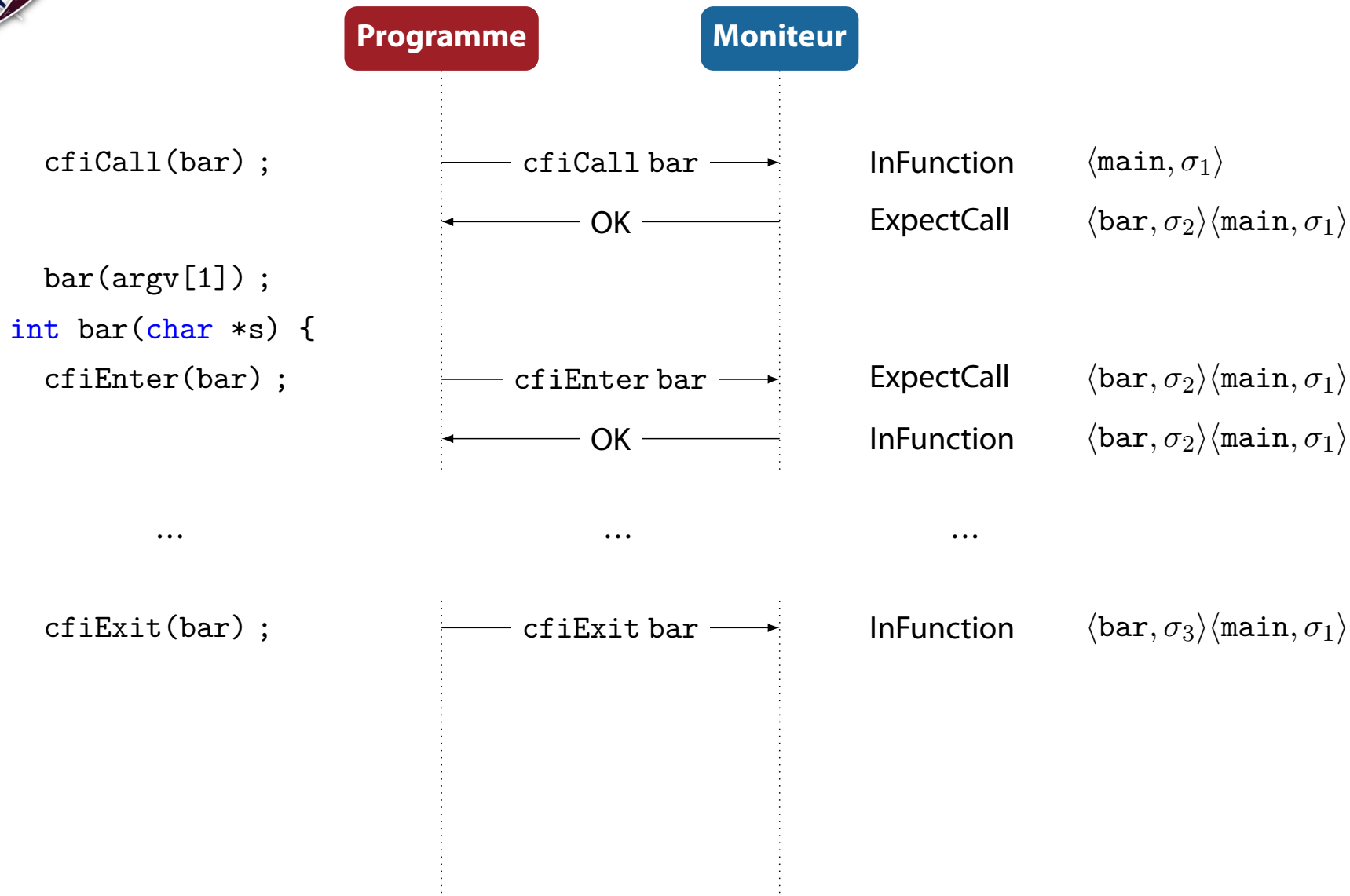
```
1  int main(int argc, char *argv[]) {
2    cfiEnter(main) ;
3    void (*func)(char *) ;
4    func = foo ;
5    cfiCall(foo) ;
6    (*func)(argv[1]) ;
7
8    cfiCall(bar) ;
9    bar(argv[1]) ;
10
11
12   return 0 ;
13 }
```

Instrumentation

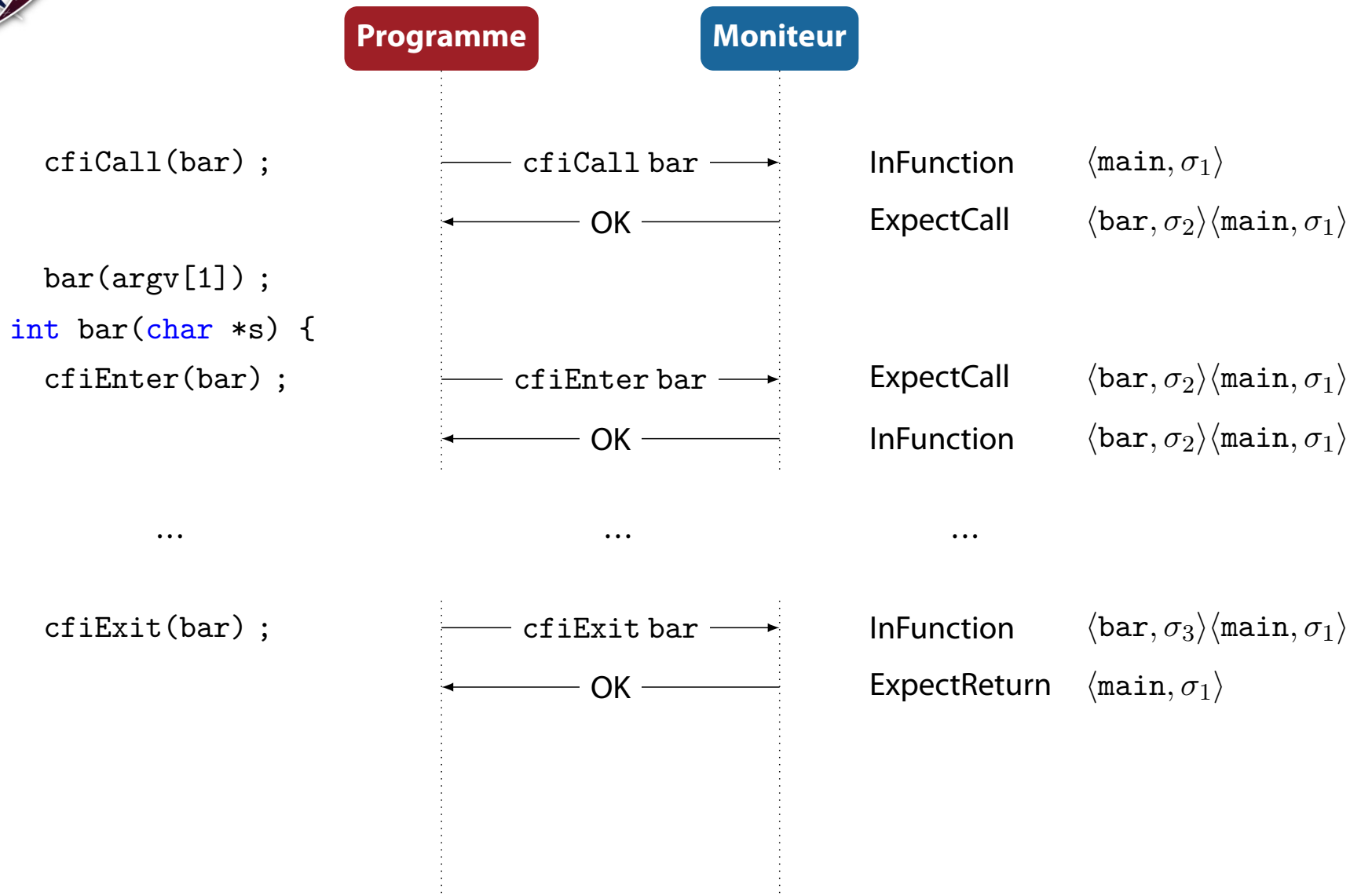
```
1  void foo(char *s) {
2    cfiEnter(foo) ;
3    char c[12] ;
4    cfiCall(strcpy) ;
5    strcpy(c, s) ;
6    cfiReturned(strcpy) ;
7    cfiExit(foo) ;
8  }
9
10 int bar(char *s) {
11   cfiEnter(bar) ;
12   cfiCall(strlen) ;
13   cfiReturned(strlen) ;
14   cfiCall(printf) ;
15   printf("[%d]\n", strlen(s)) ;
16   cfiReturned(printf) ;
17   cfiExit(bar) ;
18   return 0 ;
19 }
```

```
1  int main(int argc, char *argv[]) {
2    cfiEnter(main) ;
3    void (*func)(char *) ;
4    func = foo ;
5    cfiCall(foo) ;
6    (*func)(argv[1]) ;
7    cfiReturned(foo) ;
8    cfiCall(bar) ;
9    bar(argv[1]) ;
10   cfiReturned(bar) ;
11   cfiExit(main) ;
12   return 0 ;
13 }
```

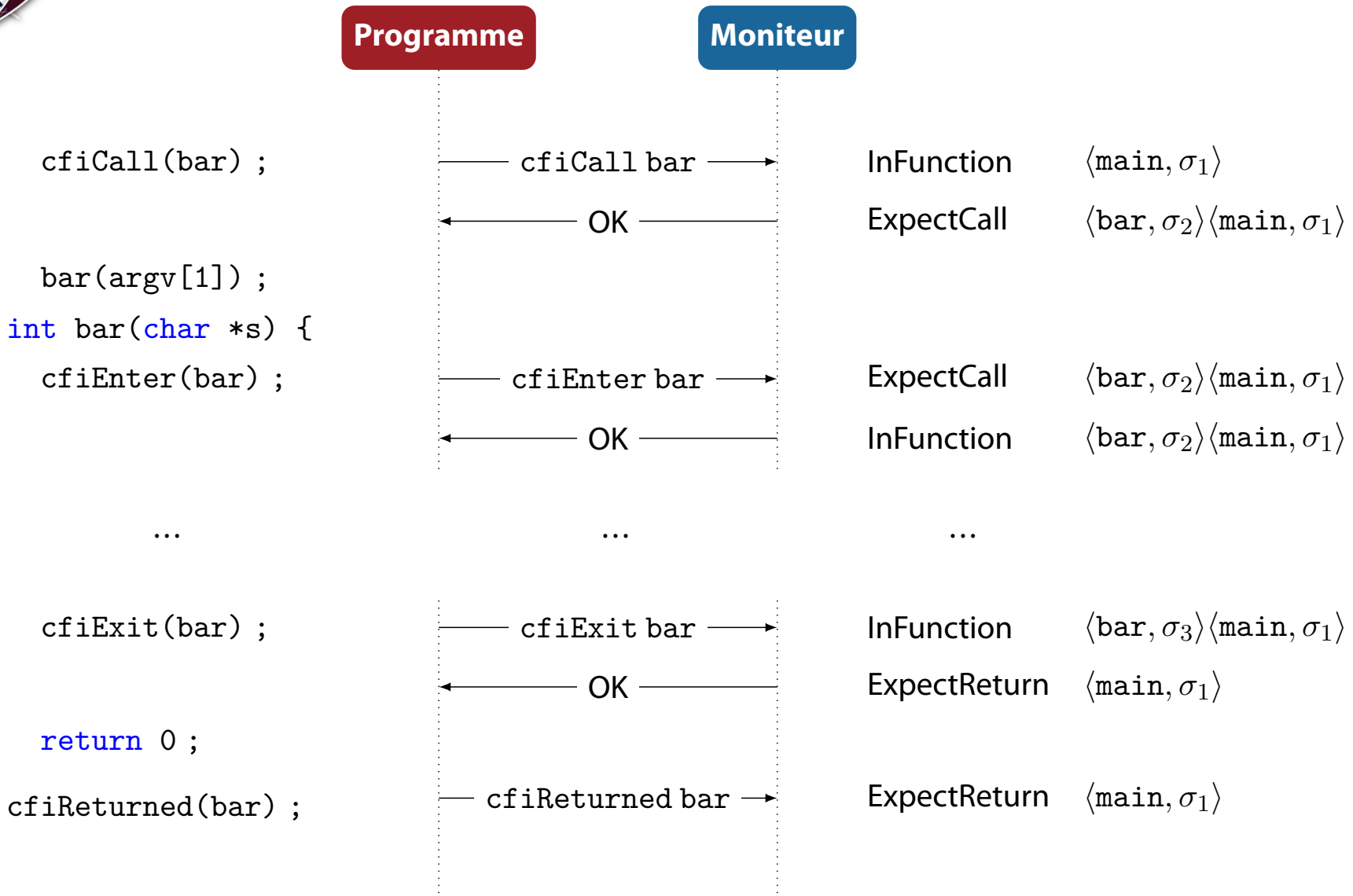
Exemple


Programme
Moniteur
cfiCall(bar) ;
cfiCall bar
InFunction
⟨main, σ1⟩
OK
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩
bar(argv[1]) ;
int bar(char *s) {
cfiEnter(bar) ;
cfiEnter bar
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩
OK
InFunction
⟨bar, σ2⟩⟨main, σ1⟩
...
...
...
cfiExit(bar) ;
cfiExit bar
InFunction
⟨bar, σ3⟩⟨main, σ1⟩

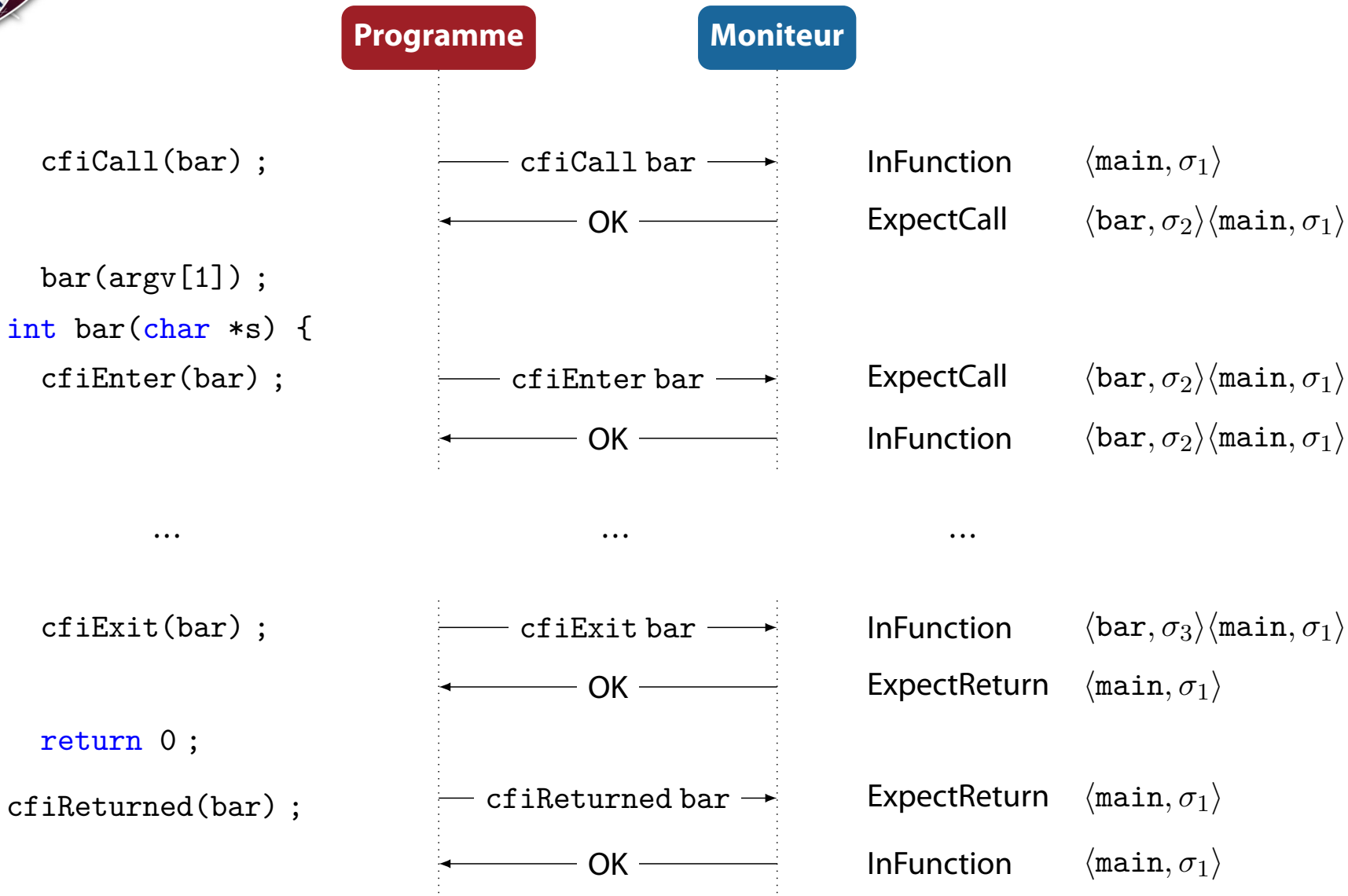
Exemple


Programme
Moniteur
cfiCall(bar) ;
cfiCall bar
InFunction ⟨main, σ1⟩
OK
ExpectCall ⟨bar, σ2⟩⟨main, σ1⟩
bar(argv[1]) ;
int bar(char *s) {
cfiEnter(bar) ;
cfiEnter bar
ExpectCall ⟨bar, σ2⟩⟨main, σ1⟩
OK
InFunction ⟨bar, σ2⟩⟨main, σ1⟩
...
...
...
cfiExit(bar) ;
cfiExit bar
InFunction ⟨bar, σ3⟩⟨main, σ1⟩
OK
ExpectReturn ⟨main, σ1⟩

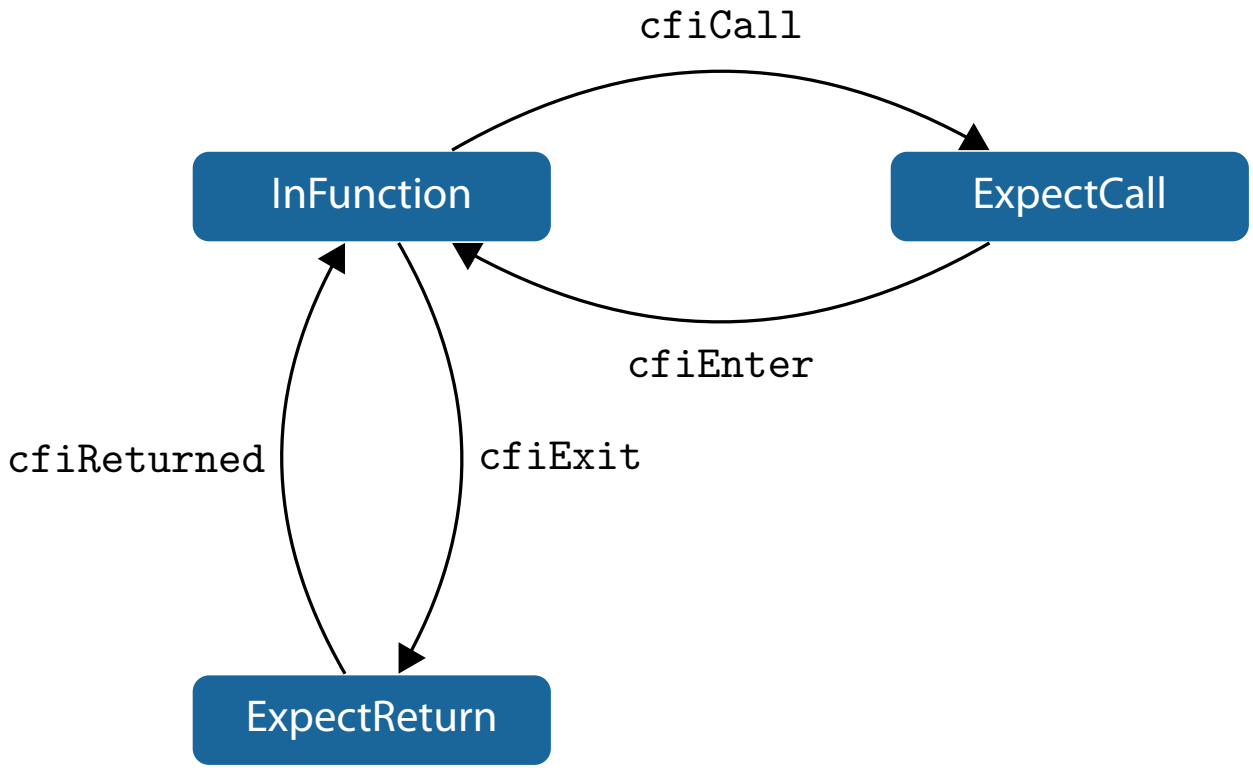
Exemple


Programme
Moniteur
cfiCall(bar) ;
cfiCall bar
InFunction
⟨main, σ1⟩
OK
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩
bar(argv[1]) ;
int bar(char *s) {
cfiEnter(bar) ;
cfiEnter bar
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩
OK
InFunction
⟨bar, σ2⟩⟨main, σ1⟩
...
...
...
cfiExit(bar) ;
cfiExit bar
InFunction
⟨bar, σ3⟩⟨main, σ1⟩
OK
ExpectReturn
⟨main, σ1⟩
return 0 ;
cfiReturned(bar) ;
cfiReturned bar
ExpectReturn
⟨main, σ1⟩

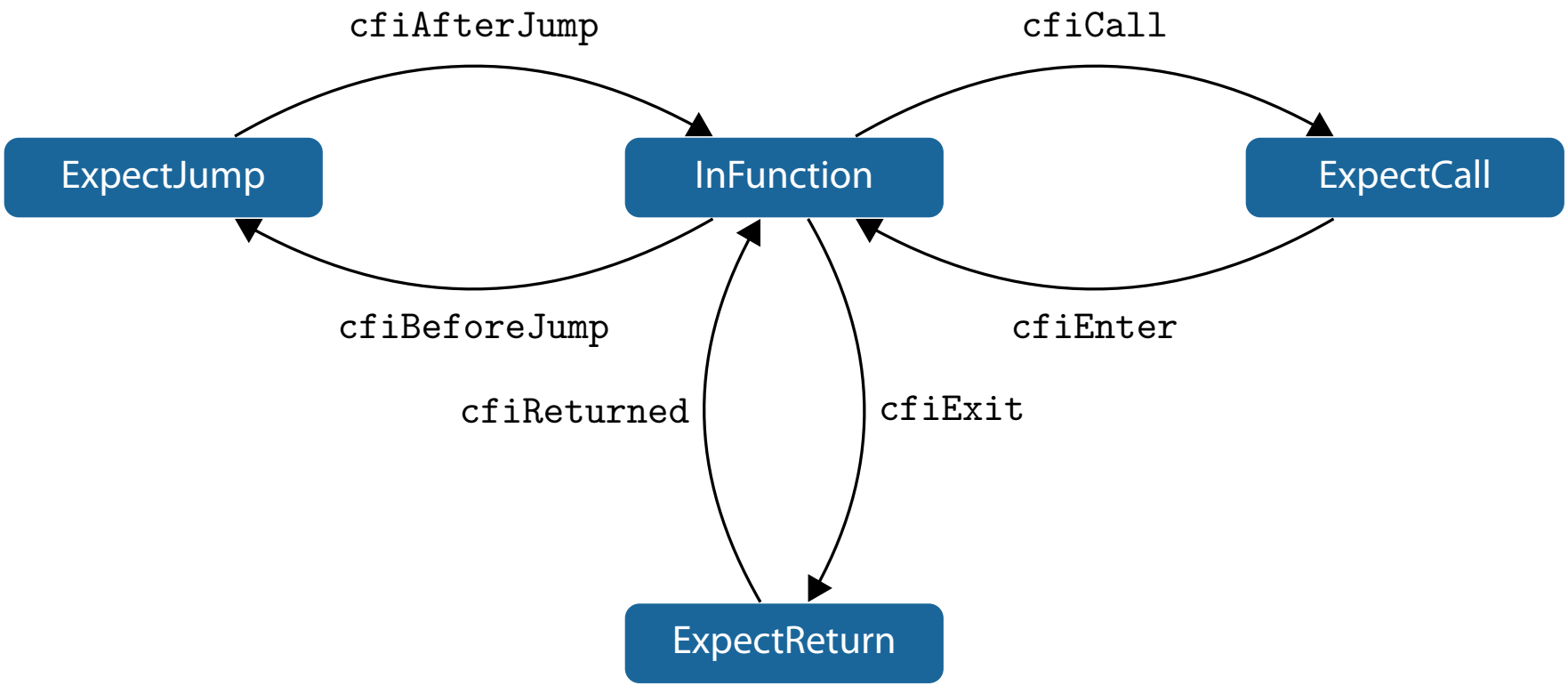
Exemple


Programme
Moniteur
cfiCall(bar) ;
cfiCall bar
InFunction
⟨main, σ1⟩
OK
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩
bar(argv[1]) ;
int bar(char *s) {
cfiEnter(bar) ;
cfiEnter bar
ExpectCall
⟨bar, σ2⟩⟨main, σ1⟩
OK
InFunction
⟨bar, σ2⟩⟨main, σ1⟩
...
...
...
cfiExit(bar) ;
cfiExit bar
InFunction
⟨bar, σ3⟩⟨main, σ1⟩
OK
ExpectReturn
⟨main, σ1⟩
return 0 ;
cfiReturned(bar) ;
cfiReturned bar
ExpectReturn
⟨main, σ1⟩
OK
InFunction
⟨main, σ1⟩

Le moniteur dans tous ses états



Le moniteur dans tous ses états



Détection de compromission du CFI

Proposition (Détection de compromission du CFI)

Si l'intégrité du flot de contrôle d'un programme instrumenté est compromise, alors elle détectée au prochain signal reçu par le moniteur.

- Preuve dans le papier
- Compromission $\Longrightarrow$ terminaison

Comment implémenter ce modèle ?

Instrumenter le source ?

- Portable, mais dépendant du langage
- C'est pénible (parser C/C++, optimisations du compilateur, …)

Instrumenter le binaire ?

- Pas portable
- C'est pénible (parser x86/ARM/Thumb, construction du CFG, …)

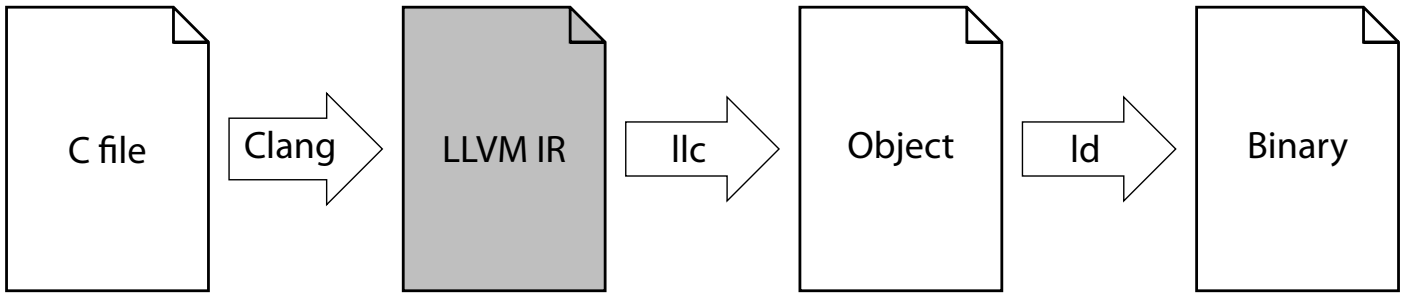
Instrumenter de l'IR LLVM

- Langage source agnostique (C,C++, …)
- Architecture agnostique (x86,x86_64,ARM,ARM64, …)
- API riche (CFG disponible, …)

Implémentation basée sur LLVM

Bonnes propriétés requises par le modèle

- Point d'entrée unique pour une fonction
- Points de sortie bien identifiés
- Partitionnement en blocs de base par fonction



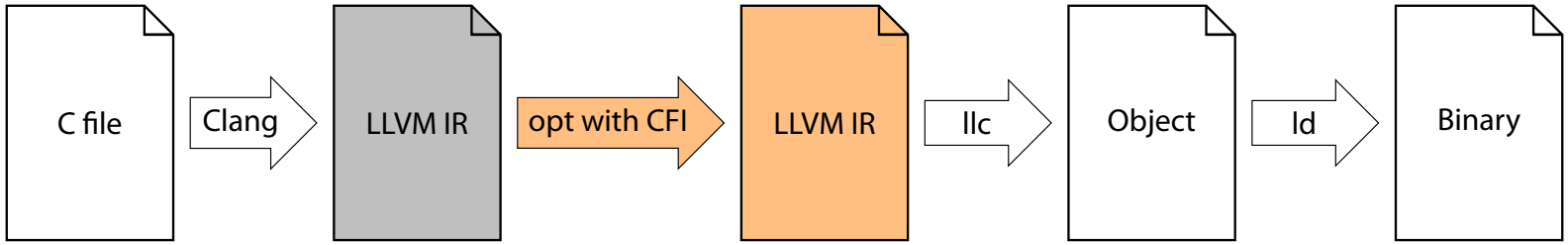
Picon

Picon : Protect Integrity of CONtrol flow

- Un plugin LLVM
- Un moniteur externe

Picon-pilation

- Instrumentation à la compilation
- Injection de code
- Deux niveaux de granularité disponibles
- C++ (non, pas d'OCaml)



Compilation et instrumentation

```
1  void foo(char *s) {
2    cfiEnter(foo) ;
3    char c[12] ;
4    cfiCall(strcpy) ;
5    strcpy(c, s) ;
6    cfiReturned(strcpy) ;
7    cfiExit(foo) ;
8  }
9
10 int bar(char *s) {
11   cfiEnter(bar) ;
12   cfiCall(strlen) ;
13   cfiReturned(strlen) ;
14   cfiCall(printf) ;
15   printf("[%d]\n", strlen(s)) ;
16   cfiReturned(printf) ;
17   cfiExit(bar) ;
18   return 0 ;
19 }
```

```
define void @foo(i8* %s) {
  call void @cfiEnter(i32 42)
  %c = alloca [12 x i8], align 8
  call void @cfiCall(i32 3)
  %1 = call i8* @strcpy(i8* %c, i8* %s)
  ...
```

- Identification des fonctions / appels
- Attribution d'ID uniques
- Injection du code

Moniteur

Moniteur interne

- Communication rapide
- Simple à implementer

Moniteur externe

- Séparation du programme et du code de vérification
- Maintenance simplifiée (bugs, vulnérabilités, mises à jour)
- Besoin d'intégrité et d'authenfication des échanges

Picon-trôle à l'exécution

Implémentation

- Preuve de concept
- `fork` + `exec`
- Communication *via* `pipe`
- Besoin d'aucun privilège particulier (sauf `kill`)
- Fichiers de transition embarqués dans les sections ELF

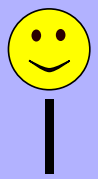
Devenir ex-static

- CFI implique souvent binaire statique
  - Mises à jour coûteuses
  - Duplication de code
- Comment gérer les `.so` ?
  - sans risques de conflits d'identifiants
  - sans faire d'exceptions

Picon dynamique

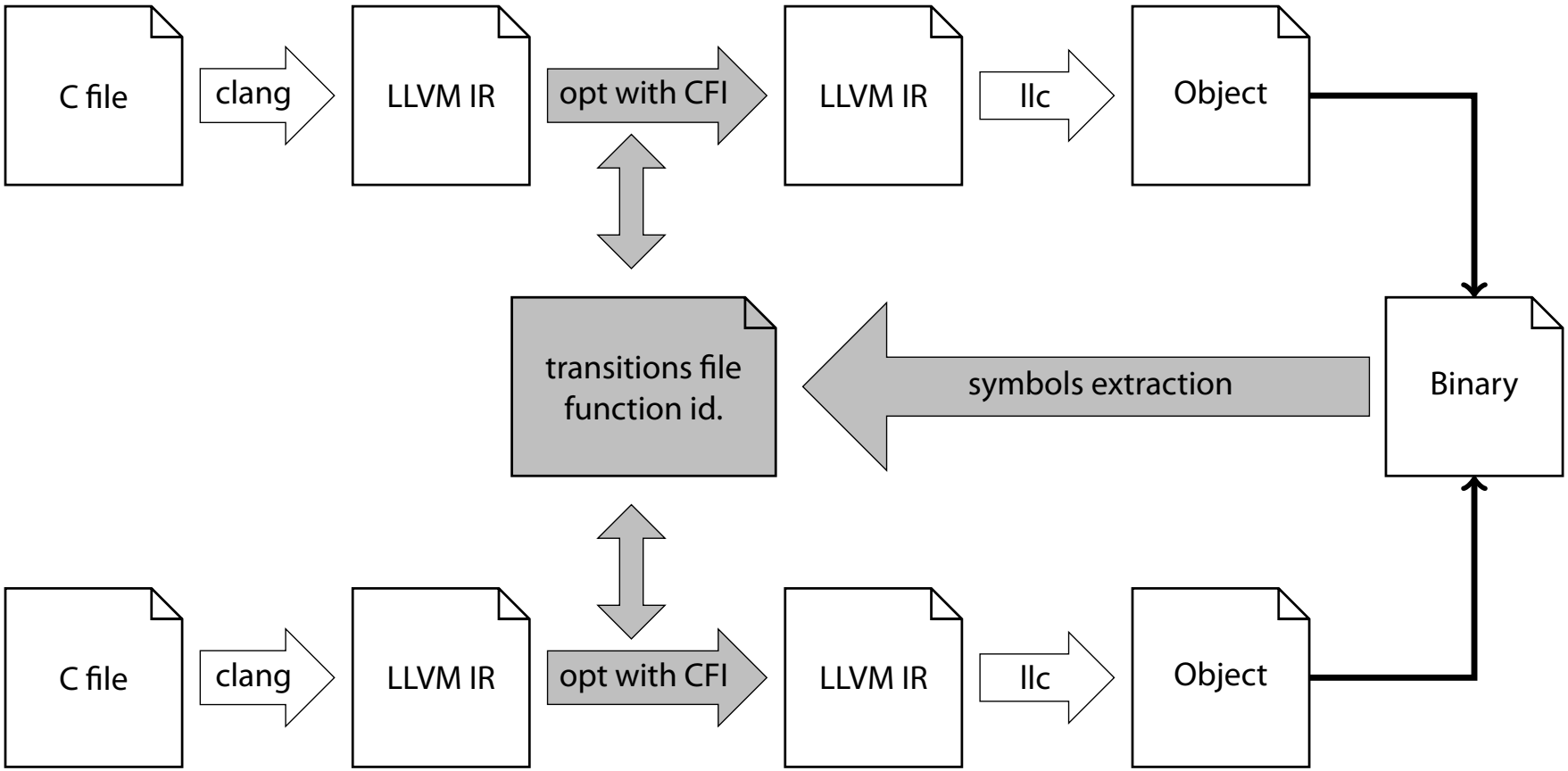
Bibliothèques dynamiques et symboles externes

- L'identifiant devient un couple (*idMod*,*idFun*)
- Utilisation d'`objdump` et de `ld` récursivement pour trouver les *idMod* contenants les fonctions appelables
- Les dépendances ont été compilées avec Picon
- Le programme contient sa liste de dépendances



Le programme, ses dépendances, **et** les transitions sont protégés.

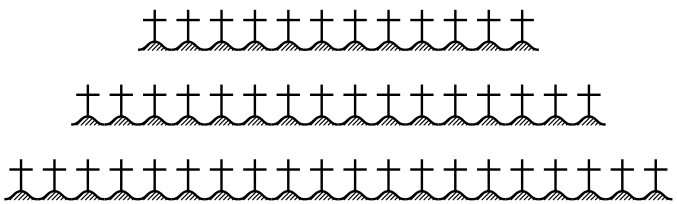
Compilation



Ça marche ?

PEGI-18

Des programmes **ont** été tués pendant nos tests



- Oui, mais c'est lent
- Immense majorité des gadgets éliminés (< 20 gadgets restants)
- Gadgets dans le runtime C et dans le loader
- Impossible de reconstruire une *ROP-chain* (avec nos exemples)

Aucune protection n'est parfaite !

Limitations

- Appels dynamiques : seulement si résolus par Clang / LLVM
  - pas de C++ (vtables, exceptions)
- Code parallèle (processus, threads)

Exemple 1 : serveur web

Serveur web minimal

- `quark` (`http://git.suckless.org/quark/`)
- Candidat adapté (réseau)

- Fonctions : $< 1\%$ overhead
- Fonctions + BB : $\sim 3\%$ overhead
- Explication : majorité d'appels systèmes, peu de fonctions, peu de BB

Exemple 2 : parseur

Parseur de code

- `sltar` (`http://git.suckless.org/sltar/`)
- Candidat adapté (lecture, parsing)

- Fonctions : $< 1\%$ overhead
- Fonctions + BB : $\sim 2231\%$ overhead
- Explications : peu de fonctions, beaucoup de sauts entre BB (+ 2,2 millions)

Futur

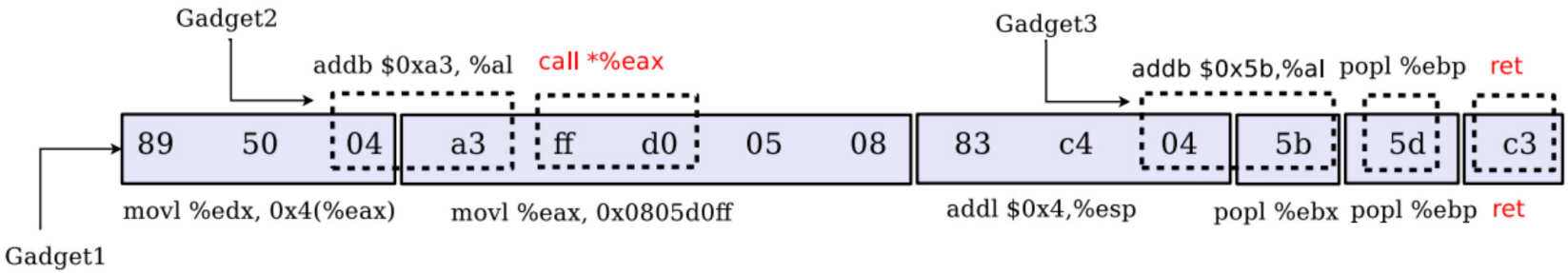
Pistes

- Code parallèle (`fork`, `clone`)
- Code dynamique (JIT, `dlopen`, appels dynamiques)
- Instrumentation des binaires *via* décompilation
- *Link Time Optimization* (LTO)
- Complémentaire à d'autres plugins LLVM (mais attention à l'ordre)

Conclusion (CFIni)

- Protection quasi-complète possible
  - Deux niveaux de granularité (détection “au plus tôt”)
  - Implémentation : protection d’abord
  - Coûteux, mais améliorable
- Implémentation hardware possible
- En cours de publication `https://github.com/ANSSI-FR/picon`

Backup slide : Gadgets non alignés



Randomisation d'opodes

- Smashing the Gadgets : Hindering Return-Oriented Programming Using In-Place Code Randomization *SP 2012*