Support de l'asm dans Astrée : rétro-ingénierie et flot de programme

Marc Chevalier

26 mars 2018

Quoi ? Pourquoi ?

État actuel

Le frontend

Sauts intra-fonction

Sauts inter-fonction

Les points de programme

Conclusion

Quoi ? Pourquoi ?

État actuel

Le frontend

Sauts intra-fonction

Sauts inter-fonction

Les points de programme

Conclusion

Quoi ?

```
1  int a;
2  void f()
3  {
4      // code C
5      asm {
6          ; code assembleur
7          mov a, 4
8      }
9  }
```

Pourquoi ?

On veut faire de la sécurité !

Beaucoup de protections matérielles mais :

- le système ne doit promouvoir personne en ring 0 ;
- le système ne doit pas faire certaines choses (modifier les registres CR3, par exemple)
- le système ne doit pas permettre des opérations illicites avec le niveau de privilège du ring 0 via un syscall

Propriétés non visibles en C : il faut analyser de l'assembleur.

État actuel

Frontend :

- Yet another parser
- Enfin une compréhension cohérente des noms C dans l'assembleur.
- Une traduction (arbre syntaxique $\rightarrow$ arbre un peu plus sémantique) en chantier

Instructions arithmétiques :

- ensemble d'instructions à compléter ;
- demandez à Stan.

Contrôle de flot :

- saut intra-fonction avec label : ok ;
- saut intra-fonction calculés : plus qu'à tester ;
- inter-fonction en cours.

Quoi ? Pourquoi ?
État actuel
Le frontend
Sauts intra-fonction
Sauts inter-fonction
Les points de programme
Conclusion

Le frontend– Les opérandes simples

Les opérandes assembleur :

- immédiate : `0xd0e0a0d0` (et toute expression arithmétique, opérateurs bizarre, priorité bizarre)
- registre : `EAX`
- fonction C : `main` (dans `call main`)
- label assembleur : `here` (dans `jmp here` ou `here - there` (C))
- opérande mémoire (simple) : `13*4+12[ESP]`
- opérande mémoire (compliqué) : `st_t.ec.eax[ECX]`, `st_t.stackp[tptr]` → `tptr->stackp`

Le frontend– Les autres

D'autres opérandes assembleur :

- locale : `i[EBP]` → `i`
- globale : `tss_table+4` → `*(&tss_table+4)`
- pointeur : `offset istack+STACK_SIZE` → `&istack + STACK_SIZE`

Le frontend– Le dessous de l'assembleur

Comment on devine ça ?

- GNU assembleur (parfois, des reliquats)
- Un peu de doc en vrac
- On essaie de comprendre le code...

Le frontend – Un nouvel AST

```
<operand> ::= <rvalue> | <lvalue>

<rvalues> ::=
    | Immediate of int
    | Asm_label of string
    | C_function of variable
    | Offset of <memory_operand>

<lvalues> ::= <memory_operand> | <register_operand>

<memory_operand> ::=
    | Global_variable of variable
    | Local_ebp of variable
    | Add_offset of <memory_operand> * Z.t
    | st_t.threadspecific_data+4[optr]
    | Field_access of <memory_operand> * path
    | Deref of (Z.t * <register_operand>) list * Z.t
    | Deref_r of <rvalue>

<register_operand> ::=
    | Registers of register
    | Local_register_var of variable
```

Quoi ? Pourquoi ?

État actuel

Le frontend

Sauts intra-fonction

Sauts inter-fonction

Les points de programme

Conclusion

Sauts intra-fonction

```
1  void f()
2  {
3      /* Code C */
4      asm {
5          ; code assembleur
6          jmp label
7          ; encore
8      }
9      /* Code C */
10     asm {
11         ; et
12         label:
13         ; encore de l'assembleur
14     }
15 }
```

Inter ou intra-bloc, avant ou arrière.

Sauts intra-fonction

Comme les goto.

```
1  type flows =
2    {
3      direct : flow; (* ∈ D♯ *)
4      return : flow;
5      (* D'autres flows *)
6      goto : label -> flow;
7      local_jumps : program_point -> flow;
8    }
```

Sauts intra-fonction

Point de programme assembleur $\approx$ pointeur de code

$$(\text{bloc assembleur}, \text{offset})$$

Opérations valides :

- pointeur de code + entier
- différence

Opérations invalides :

- produit, division...
- déréférencement

Sauts intra-fonction

$$func^\sharp : f \mapsto f'$$

$$c \quad : f \mapsto \begin{cases} direct & \mapsto c^\sharp(f(direct)) \\ x & \mapsto f(x) \end{cases}$$

$$\texttt{return e} \quad : f \mapsto \begin{cases} return & \mapsto f(direct) \vee f(return) \\ direct & \mapsto \bot \\ x & \mapsto f(x) \end{cases}$$

$$\texttt{jmp pp} \quad : f \mapsto \begin{cases} pp & \mapsto f(direct) \\ direct & \mapsto \bot \\ x & \mapsto f(x) \end{cases}$$

$$\texttt{pp:} \quad : f \mapsto \begin{cases} pp & \mapsto \bot \\ direct & \mapsto f(direct) \vee f(pp) \\ x & \mapsto f(x) \end{cases}$$

Sauts intra-fonction

$$func^\sharp : f \mapsto f'$$

- $f'(direct)$: tout ce qui a atteint la fin sans rencontrer de return (erreur dans une fonction non void).
- $f'(return)$: tout ce qui a rencontré un return.
- $\{f'(pp) \mid pp\}$: tous les sauts qui n'ont pas trouvé leurs labels dans la suite : saut arrière.

Sauts intra-fonction

$$func^\sharp : f \mapsto f'$$

Flot entrant : $f_{in} = (direct \mapsto \text{contexte d'appel})$ pour un appel C vers C.

$$f_{out} = lfp^\sharp \left(F \mapsto func^\sharp \left(\begin{cases} direct & \mapsto f_{in} \\ pp & \mapsto F(pp) \\ label & \mapsto F(label) \end{cases} \right) \right)$$

Sauts locaux $\approx$ goto

Quoi ? Pourquoi ?

État actuel

Le frontend

Sauts intra-fonction

Sauts inter-fonction

Les points de programme

Conclusion

Sauts inter-fonction

```c
void b()
{
    /* Du C */
    asm {
        ; Du x86
        l:
        ; Du x86
    }
    /* Du C */
}
```

```c
void a()
{
    /* Du C */
    asm {
        ; Du x86
        jmp l
        ; Du x86
    }
    /* Du C */
}
```

Sauts inter-fonction

Un début familier

```
1  type flows =
2  {
3      direct : flow;
4      return : flow;
5      (* D'autres flows *)
6      goto : label -> flow;
7      local_jumps : program_point -> flow;
8      distant_jumps : program_point -> flow;
9  }
```

Sauts inter-fonction

$$func^\sharp : f \mapsto f'$$

$$\texttt{jmp pp\_d} \quad : f \mapsto \begin{cases} pp & \mapsto f(direct) \\ direct & \mapsto \bot \\ x & \mapsto f(x) \end{cases}$$

$$\texttt{pp\_d:} \quad : f \mapsto \begin{cases} direct & \mapsto f(direct) \vee f(pp\_d) \\ pp\_d & \mapsto \bot \\ x & \mapsto f(x) \end{cases}$$

Sauts inter-fonction

$$func^\sharp : f \mapsto f'$$

- $f'(direct)$: comme avant
- $f'(return)$: ...
- $\{f'(pp) \mid pp \text{ distant}\}$: tout les sauts inter-fonction.

Sauts inter-fonction

distant `jmp` assembleur vs appel C :

- L'appel C retourne toujours $\Rightarrow$ on suspend l'analyse le temps que l'analyse de l'appel termine, puis on continue.
- L'appel asm peut ne jamais terminer (ou revenir par ailleurs) $\Rightarrow$ une autre stratégie.

Sauts inter-fonction

$$\begin{aligned} a : f^a &\mapsto f'^a \\ b : f^b &\mapsto f'^b \end{aligned}$$

- On collecte les contexte d'appels ($f'^a[pp]$)
- On exécute les sauts/appels. $f^b_{in} = (pp \mapsto f'^a[pp])$
- On utilise également une pile des fonctions analysées (pour éviter les récursions et assurer la terminaison).
- On collecte les retours (`ret` et `return`) et on les injecte en entrée.

$$f^a_{in} = \begin{cases} l & \mapsto f^b_{out}(return) \\ x & \mapsto f^a_{in}(x) \end{cases}$$

- Point fixe.

Quoi ? Pourquoi ?
État actuel
Le frontend
Sauts intra-fonction
Sauts inter-fonction
Les points de programme
Conclusion

Les points de programme

Point de programme assembleur vs. pointeur de code :

Point de programme = entre deux instructions (offset ordinal). L'analyse a besoin de ça.

Pointeur de code = position dans le code physique (offset en octets). L'assembleur fonctionne avec ça.

Dépend de la taille des instructions
Tous les pointeurs de code ne sont pas valides.
Il faut calculer toutes les tailles possibles des instructions (en cours).

Conclusion

- saut intra-fonction avec label ok ;
- déduction de la taille des instructions finie (réparer avec le nouvel AST) ;
- saut intra-fonction calculés à tester ;
- inter-fonction en cours.