

Triton : Framework d'exécution concolique

Florent Saudel
Jonathan Salwan

SSTIC
Rennes – France

3 juin 2015



Qui sommes-nous ?

- Jonathan Salwan étudiant à l'université de Bordeaux (Master CSI) et employé chez Quarkslab
- Florent Saudel étudiant à l'université de Bordeaux (Master CSI) effectuant un stage chez Amossys

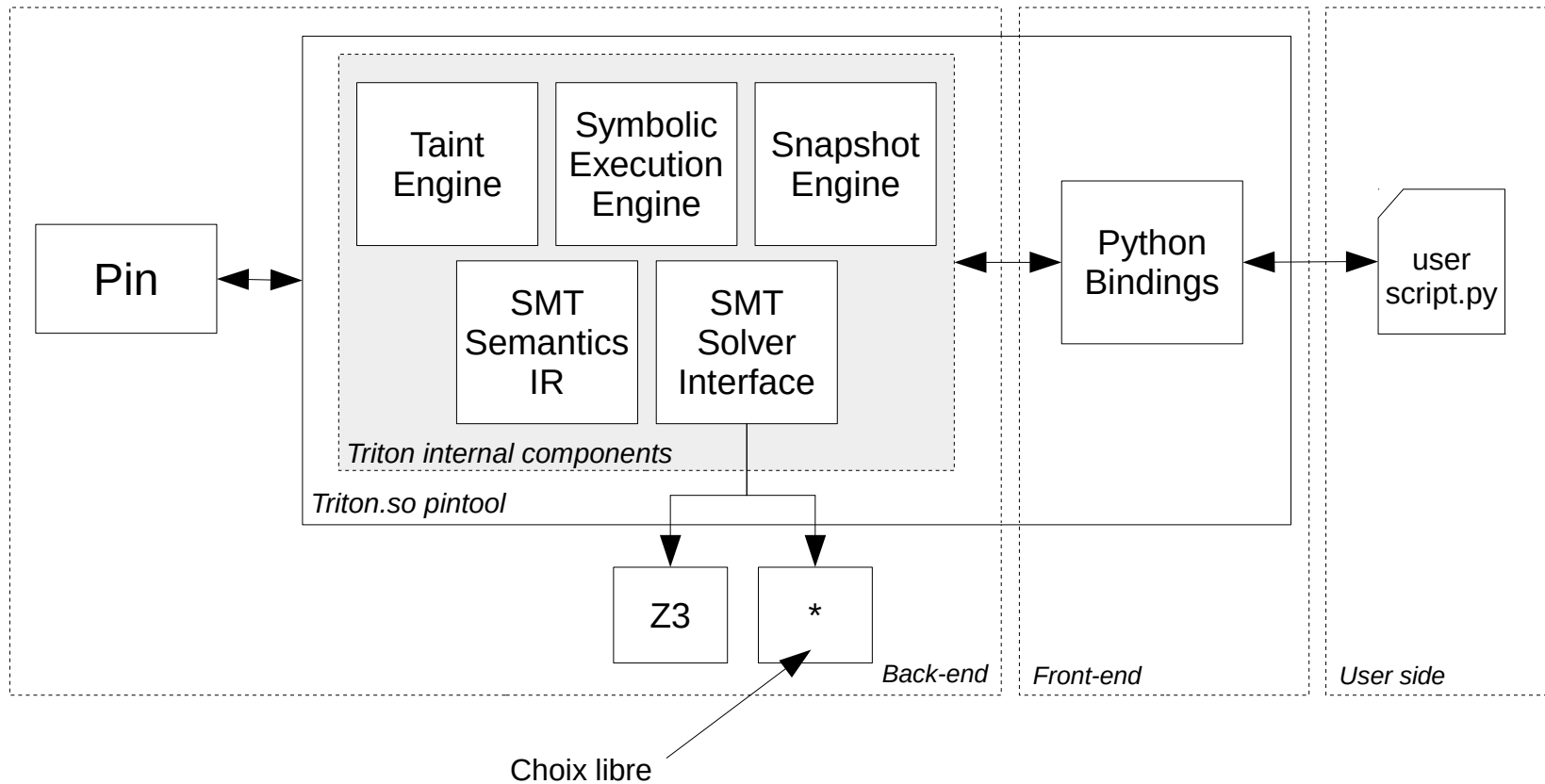
D'où vient Triton ?

- Triton est notre projet de fin d'étude pour le Master CSI
 - Supervisé par Emmanuel Fleury du LaBRI
 - Sponsorisé par Quarkslab

Qu'est-ce que Triton ?

- Framework d'exécution concolique basé sur Pin
- Fournit des classes supplémentaires
 - Moteur d'exécution symbolique
 - Représentation des instructions en SMT2-LIB
 - Interface avec un SMT solver
 - Moteur de teinte
 - Moteur de rejeu
 - Une API et des bindings Python

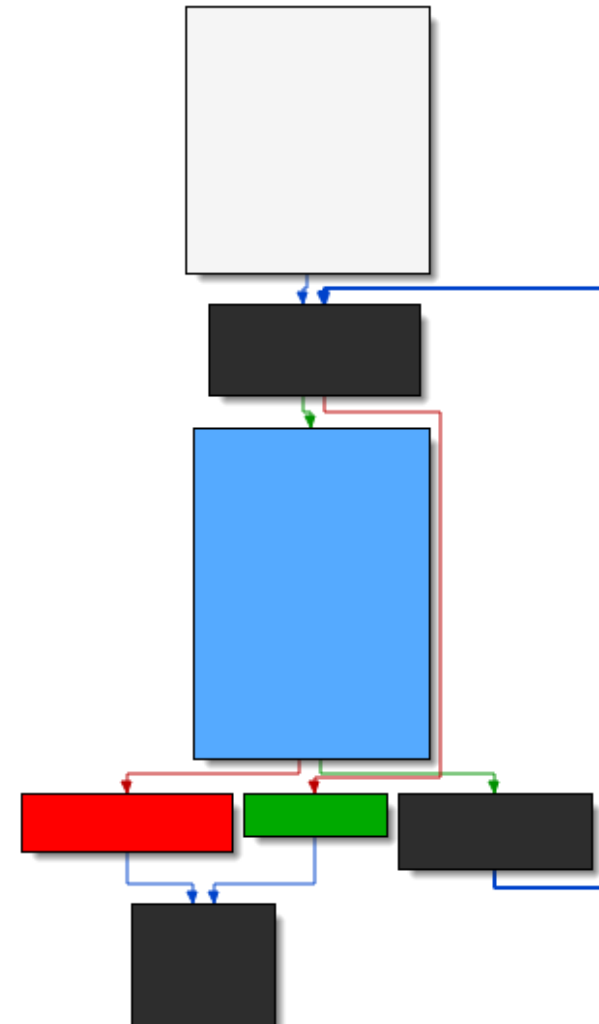
Qu'est-ce que Triton ?



Présentation de l'**API** au travers d'un exemple simple

Présentation de l'API au travers d'un exemple simple

- Exemple
 - Programme qui attend un mot de passe
 - Deux branches ciblées
 - Password valide
 - Password invalide



Roadmap

- 4 questions importantes
 - Qu'est-ce que contrôle l'utilisateur ?
 - Quelles sont les conditions pendant l'exécution ?
 - Quel est le lien entre l'utilisateur et ces conditions ?
 - Comment résoudre ces conditions pour influencer l'exécution ?

Roadmap

- 4 questions importantes
 - **Qu'est-ce que contrôle l'utilisateur ?**
 - Quelles sont les conditions pendant l'exécution ?
 - Quel est le lien entre l'utilisateur et ces conditions ?
 - Comment résoudre ces conditions pour influencer l'exécution ?

Qu'est-ce que contrôle l'utilisateur ?

- Analyse par teinte

```
import triton

if __name__ == '__main__':

    # Lancer l'analyse sur la fonction check
    triton.startAnalysisFromSymbol('check')

    # Teinte RAX et RBX à l'adresse 0x40058e
    triton.taintRegFromAddr(0x40058e, [IDREF.REG.RAX, IDREF.REG.RBX])

    # Déteinte RCX à l'adresse 0x40058e
    triton.untaintRegFromAddr(0x40058e, [IDREF.REG.RCX])

    # Exécute le programme
    triton.runProgram()
```

Qu'est-ce que contrôle l'utilisateur ?

- Teindre une adresse en *runtime*

0x40058b: movzx eax, byte ptr [rax]

RAX pointe sur une zone contrôlable
par l'utilisateur

```
def cbeforeSymProc(instruction):
```

```
    if instruction.address == 0x40058b:  
        rax = getRegValue(IDREF.REG.RAX)  
        taintMem(rax)
```

Callback avant le traitement
symbolique

```
if __name__ == '__main__':  
    startAnalysisFromSymbol('check')  
    addCallback(cbeforeSymProc, IDREF.CALLBACK.BEFORE_SYMPROC)  
    runProgram()
```

Qu'est-ce que contrôle l'utilisateur ?

- Triton diffuse la teinte
 - En se basant sur la sémantique de chaque instruction
- Possibilité de savoir ce qui est teinté
 - Registres
 - Mémoire


```
def callback(instruction):  
    if instruction.address == 0x40059a:  
        if isRegTainted(IDREF.REG.RAX):  
            ...  
  
    if instruction.address == 0x40078c:  
        if isMemTainted(memory_address):  
            ...  
  
if __name__ == '__main__':  
    startAnalysisFromSymbol('check')  
    addCallback(callback, IDREF.CALLBACK.BEFORE)  
    runProgram()
```

Roadmap

- 4 questions importantes
 - Qu'est-ce que contrôle l'utilisateur ?
 - **Quelles sont les conditions pendant l'exécution ?**
 - Quel est le lien entre l'utilisateur et ces conditions ?
 - Comment résoudre ces conditions pour influencer l'exécution ?

Quelles sont les conditions pendant l'exécution ?

- Une trace est une séquence d'instructions
 - $T = (ins_1 \wedge ins_2 \wedge ins_3 \wedge ins_x)$
- Pour chaque instruction :
 - Construction d'expression(s) arithmétique(s)
 - SMT2-LIB



```
if __name__ == '__main__':  
    startAnalysisFromSymbol('check')  
    runProgram()
```

Les expressions ?

Exemple :

```
movzx eax, byte ptr [mem]
add eax, 2
mov ebx, eax
```

```
// All refs initialized to -1
Register Reference Table {
    EAX : -1,
    EBX : -1,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
}
```

Les expressions ?

Exemple :

```
► movzx eax, byte ptr [mem]    #0 = ((_ zx 24)0x61)
  add eax, 2
  mov ebx, eax
```

```
// All refs initialized to -1
Register Reference Table {
  EAX : #0,
  EBX : -1,
  ECX : -1,
  ...
}
```

```
// Empty set
Symbolic Expression Set {
  <#0, ((_ zx 24)0x61)>
}
```


Les expressions ?

Exemple :

```
movzx eax, byte ptr [mem]    #0 = ((_zx 24)0x61)
➔ add eax, 2                  #1 = add(#0, 2)
mov ebx, eax
```

```
// All refs initialized to -1
Register Reference Table {
    EAX : #1,
    EBX : -1,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <#1, add(#0, 2)>,
    <#0, ((_zx 24)0x61)>
}
```

Les expressions ?

Exemple :

movzx eax, byte ptr [mem]	#0 = ((_zx 24)0x61)
add eax, 2	#1 = add(#0, 2)
► mov ebx, eax	#2 = #1

```
// All refs initialized to -1
Register Reference Table {
    EAX : #1,
    EBX : #2,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <#2, #1>,
    <#1, add(#0, 2)>,
    <#0, ((_zx 24)0x61)>
}
```

Backward reconstruction

Exemple :

```
movzx eax, byte ptr [mem]  
add eax, 2
```

mov ebx, eax —————> Quelle est l'expression d'EBX ?

```
// All refs initialized to -1  
Register Reference Table {  
    EAX : #1,  
    EBX : #2,  
    ECX : -1,  
    ...  
}
```

```
// Empty set  
Symbolic Expression Set {  
    <#2, #1>,  
    <#1, add(#0, 2)>,  
    <#0, ((_zx 24)0x61)>  
}
```

Backward reconstruction

Exemple :

```
movzx eax, byte ptr [mem]
add eax, 2
```

mov ebx, eax —————> Quelle est l'expression d'EBX ?

```
// All refs initialized to -1
Register Reference Table {
  EAX : #1,
  EBX : #2,
  ECX : -1,
  ...
}
```

```
// Empty set
Symbolic Expression Set {
  <#2, #1>,
  <#1, add(#0, 2)>,
  <#0, ((_ zx 24)0x61)>
}
```

EBX possède la référence **#2**

Backward reconstruction

Exemple :

```
movzx eax, byte ptr [mem]
add eax, 2
mov ebx, eax —————> Quelle est l'expression d'EBX ?
```

```
// All refs initialized to -1
Register Reference Table {
  EAX : #1,
  EBX : #2,
  ECX : -1,
  ...
}
```

```
// Empty set
Symbolic Expression Set {
  <#2, #1>,
  <#1, add(#0, 2)>,
  <#0, ((_ zx 24)0x61)>
}
```

EBX possède la référence **#2**
Qu'est-ce que **#2** ?

Backward reconstruction

Exemple :

```
movzx eax, byte ptr [mem]
add eax, 2
mov ebx, eax —————> Quelle est l'expression d'EBX ?
```

```
// All refs initialized to -1
Register Reference Table {
  EAX : #1,
  EBX : #2,
  ECX : -1,
  ...
}
```

```
// Empty set
Symbolic Expression Set {
  <#2, #1>, ←
  <#1, add(#0, 2)>,
  <#0, ((_ zx 24)0x61)>
}
```

EBX possède la référence **#2**

Qu'est-ce que **#2** ?

Reconstruction: EBX = **#2**

Backward reconstruction

Exemple :

```
movzx eax, byte ptr [mem]
add eax, 2
mov ebx, eax —————> Quelle est l'expression d'EBX ?
```

```
// All refs initialized to -1
Register Reference Table {
  EAX : #1,
  EBX : #2,
  ECX : -1,
  ...
}
```

```
// Empty set
Symbolic Expression Set {
  <#2, #1>, ←
  <#1, add(#0, 2)>,
  <#0, ((_ zx 24)0x61)>
}
```

EBX possède la référence **#2**

Qu'est-ce que **#2** ?

Reconstruction: EBX = **#1**

Backward reconstruction

Exemple :

```
movzx eax, byte ptr [mem]
add eax, 2
mov ebx, eax —————> Quelle est l'expression d'EBX ?
```

```
// All refs initialized to -1
Register Reference Table {
  EAX : #1,
  EBX : #2,
  ECX : -1,
  ...
}
```

```
// Empty set
Symbolic Expression Set {
  <#2, #1>,
  <#1, add(#0, 2)>,
  <#0, ((_ zx 24)0x61)>
}
```

EBX possède la référence **#2**

Qu'est-ce que **#2** ?

Reconstruction: EBX = **add(#0, 2)**

Backward reconstruction

Exemple :

```
movzx eax, byte ptr [mem]
add eax, 2
mov ebx, eax —————> Quelle est l'expression d'EBX ?
```

```
// All refs initialized to -1
Register Reference Table {
  EAX : #1,
  EBX : #2,
  ECX : -1,
  ...
}
```

```
// Empty set
Symbolic Expression Set {
  <#2, #1>,
  <#1, add(#0, 2)>,
  <#0, ((_ zx 24)0x61)> ←
```

EBX possède la référence **#2**

Qu'est-ce que **#2** ?

Reconstruction: $EBX = \text{add}((_ \text{zx } 24)0x61), 2)$

Les expressions depuis les bindings Python

Code

```
def my_callback_after(instruction):
    print '%#x: %s' % (instruction.address, instruction.assembly)
    for se in instruction.symbolicElements:
        print '\t -> ', se.expression
    print

if __name__ == '__main__':
    startAnalysisFromSymbol('check')
    addCallback(my_callback_after, IDREF.CALLBACK.AFTER)
    runProgram()
```

Résultat

```
0x4005ab: movsx eax, al
        -> #70 = ((_ sign_extend 24) ((_ extract 7 0) #68))
        -> #71 = (_ bv4195758 64)

0x4005ae: cmp ecx, eax
        -> #72 = (bvsb ((_ extract 31 0) #52) ((_ extract 31 0) #70))
        ...
        -> #77 = (ite (= ((_ extract 31 31) #72) (_ bv1 1)) (_ bv1 1) (_ bv0 1))
        -> #78 = (ite (= #72 (_ bv0 32)) (_ bv1 1) (_ bv0 1))
        -> #79 = (_ bv4195760 64)

0x4005b0: jz 0x4005b9
        -> #80 = (ite (= #78 (_ bv1 1)) (_ bv4195769 64) (_ bv4195762 64))
```

Roadmap

- 4 questions importantes
 - Qu'est-ce que contrôle l'utilisateur ?
 - Quelles sont les conditions pendant l'exécution ?
 - **Quel est le lien entre l'utilisateur et ces conditions ?**
 - Comment résoudre ces conditions pour influencer l'exécution ?

Quel est le lien entre l'utilisateur et ces conditions ?

Exemple :

```
► movzx eax, byte ptr [mem]    #0 = Symvar_0
  add eax, 2
  mov ebx, eax
```

```
// All refs initialized to -1
Register Reference Table {
  EAX : #0,
  EBX : -1,
  ECX : -1,
  ...
}
```

```
// Empty set
Symbolic Expression Set {
  <#0, Symvar_0>
}
```

Quel est le lien entre l'utilisateur et ces conditions ?

Assembleur

```
0x40058b: movzx eax, byte ptr [rax]
...
...
0x4005ae: cmp ecx, eax
```

Code

```
def callback_after(instruction):
    if instruction.address == 0x4005ae:
        # Get the symbolic expression ID of ZF
        zfId = getRegSymbolicID(IDREF.FLAG.ZF)

        # Backtrack the symbolic expression ZF
        zfExpr = getBacktrackedSymExpr(zfId)

        # Craft a new expression over the ZF expression : (assert (= zfExpr True))
        expr = smt2lib.smtAssert(smt2lib.equal(zfExpr, smt2lib.bvtrue()))
        print expr
```

Résultat

```
(assert (= (ite (= (bvsb (( _ extract 31 0) (( _ extract 31 0) (bvxor (( _ extract 31 0)
(bvsb (( _ extract 31 0) (( _ sign_extend 24) (( _ extract 7 0) SymVar_0)) (( _ bv1 32)))
( _ bv85 32)))) (( _ extract 31 0) (( _ sign_extend 24) (( _ extract 7 0) (( _ zero_extend 24)
( _ bv49 8))))) (( _ bv0 32)) ( _ bv1 1) ( _ bv0 1) ( _ bv1 1)))
```

Roadmap

- 4 questions importantes
 - Qu'est-ce que contrôle l'utilisateur ?
 - Quelles sont les conditions pendant l'exécution ?
 - Quel est le lien entre l'utilisateur et ces conditions ?
 - **Comment résoudre ces conditions pour influencer l'exécution ?**

Comment résoudre ces conditions pour influencer l'exécution ?

- SMT solver
- Injection de la solution en mémoire
- Utilisation des *snapshots*

Utilisation du SMT solver

Code

```
...
zfExpr = getBacktrackedSymExpr(zfId)

# Customiser son expression
expr = smt2lib.smtAssert(
    smt2lib.equal(
        zfExpr,
        smt2lib.bvtrue()
    )
)

model = getModel(expr)
print model
```

Résultat

```
{'SymVar_0': 0x65}
```


Injection de la solution en mémoire

Extrait du code précédent

```
...  
model = getModel(expr)  
print model
```

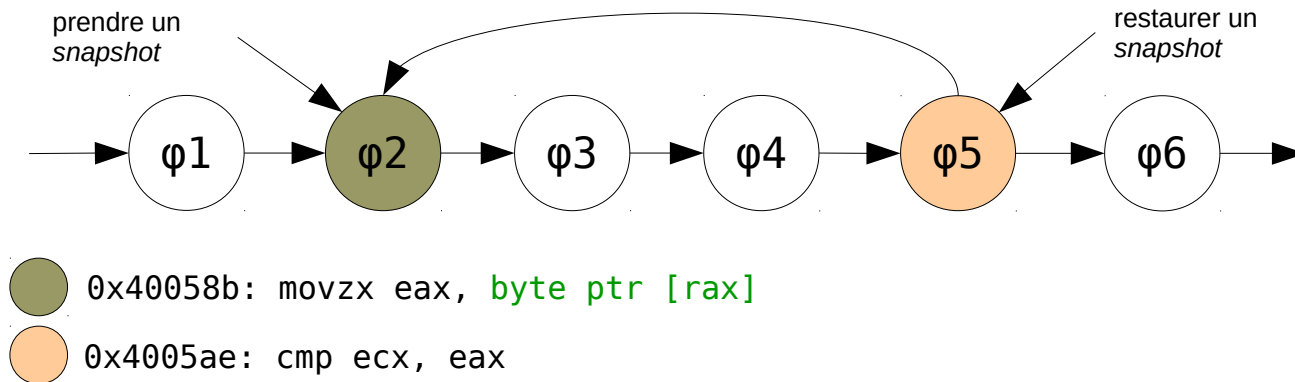
Résultat

```
{'SymVar_0': 0x65}
```

Injecter le modèle en mémoire

```
for k, v in model.items():  
    setMemValue(getMemoryFromSymVar(k), getSymVarSize(k), v)
```

Utilisation des *snapshots*



```
def callback(instruction):  
    if instruction.address == 0x40058b and isSnapshotEnable() == False:  
        takeSnapshot()  
  
    if instruction.address == 0x4005ae:  
        if getFlagValue(IDREF.FLAG.ZF) == 0:  
            zfExpr = getBacktrackedSymExpr(...)  
            expr = smt2lib.smtAssert(...zfExpr...)  
            for k, v in getModel(expr).items():  
                setMemValue(...)  
            restoreSnapshot()
```

Demo en ' $2s_\pi$

Vue d'ensemble

Vue d'ensemble

- ~80 fonctions exportées vers des bindings Python
- Grossièrement :
 - Teindre et déteindre chaque partie de la mémoire et les registres
 - Modifier la mémoire et les registres à la volée
 - Ajouter des callbacks à chaque point du programme
 - Assigner des variables symboliques
 - Construire et “*customiser*” nos expressions symboliques
 - Résoudre des contraintes
 - Effectuer des rejeux de trace (snapshot)
 - Tout en python

Que peut-on faire avec Triton ?

- Analyser une trace avec des informations concrètes
- Effectuer une analyse symbolique
- Générer et résoudre des contraintes
- Effectuer de la couverture de code
- Modifier les données à la volée
- Rejouer des traces directement en mémoire
- Scripter du débogage
- Accéder à Pin depuis des *bindings* Python
- Et probablement pleins d'autres choses :)

Conclusion

Conclusion

- Triton:
 - Pintool qui fournit des classes supplémentaires pour du DBA
 - Framework d'exécution concolique
 - API de haut niveau (bindings Python)
 - Analyse que des binaires x86-64
 - ~100 sémantiques
 - Big up à Kevin `wisk` Szkudlapski et Francis `gg` Gabriel pour le fichier x86.yaml du projet Medusa
 - Gratuit et open-source <3
 - Disponible ici : github.com/JonathanSalwan/Triton

Merci pour votre attention

Question(s)?

- Contacts

- fsaudel@gmail.com
- jsalwan@quarkslab.com

- Remerciements

- Rolf Rolles, Sean Heelan, Sébastien Bardin, Fred Raynal et Serge Guelton pour leurs conseils avisés !
- Quarkslab pour le sponsor !

