



## Analisi e verifica di un framework per l'analisi dinamica di codice binario

*Relatore:*

Dott. Mattia Monga

*Correlatore:*

Dott. Roberto Paleari

*Tesi di Laurea di:*

Luca Giancane  
mat. 675309

## Scenario

- Qualsiasi applicazione di una certa complessità contiene, con buona probabilità al suo interno una o più vulnerabilità;
- molte applicazioni sono disponibili solo in codice binario.

## Motivazione

Alcuni errori di programmazione possono essere sfruttati per sovvertire il comportamento di un'applicazione ed eventualmente prendere il controllo di interi sistemi.

## Smart fuzzer

Il tool di *smart fuzzing*:

- **uso:** analizzare un codice binario;
- **scopo:** rilevamento automatico di vulnerabilità all'interno di codice binario.

## Fuzzing classico

Utilizza come input per una data applicazione dati totalmente random, allo scopo di causare errori di esecuzione.

**Problema:** non consente di percorrere ogni possibile cammino di esecuzione.

- 1 Utilizza tecniche di *program analysis* ibride:
  - **analisi statica:** fornisce informazioni generali sul comportamento di un'applicazione;
  - **analisi dinamica:** monitora il flusso di esecuzione, raffinando le informazioni raccolte dall'analisi precedente.
- 2 Estrapola un insieme di vincoli per la creazione di input in grado di indirizzare il flusso di esecuzione verso stati in cui potrebbero manifestarsi delle vulnerabilità.
- 3 Individua l'alterazione di zone di memoria sensibili, tramite il monitoraggio dell'esecuzione.

perchè serve?

Più efficace del *fuzzing* tradizionale.

## Limite

Il framework Smart fuzzer effettua una gestione incompleta delle funzioni inline.

Problema: generazione di vincoli per la creazione di un nuovo input incompleta.

## Funzioni inline

- Sostituzione della chiamata a funzione con il corpo della funzione stessa.

```
1  void cp(char *src)
2  {
3      char buffer[80];
4      if (src[0]!='c' || src[1]!='p')
5          {
6              printf("error\n");
7          }
8      else if(strlen(src) > 10)
9          {
10         strcpy(buffer, src);
11     }
12 }
```

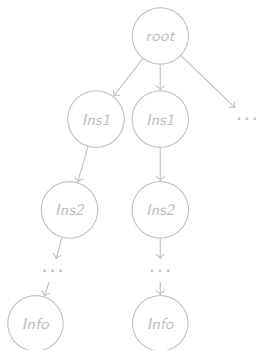
```
1 void cp(char *src)
2 {
3     char buffer[80];
4     if (src[0]!='c' || src[1]!='p')
5     {
6         printf("error\n");
7     }
8     else if(strlen(src) > 10) → funzione inline
9     {
10        strcpy(buffer, src); → vulnerabilità
11    }
12 }
```

Problema: la condizione in linea 8 dipende dall'input.

Soluzione: riconoscimento funzioni inline.

## Funzione strlen()

```
1   r32(ECX) := c32(0xffffffff)
2   m32[(r32(EBP) + None)] := r32(EAX)
3   r32(EAX) := c32(0x0)
4   r1(DF) := c1(0x0)
5   r32(EDI) := m32[(r32(EBP) + None)]
6   JUMP ((r32(ECX) == c32(0x0))) None
7   r8(TMP) := (m32[(r16(ES) + r32(EDI))] - r8(AL))
8   r1(ZF) := (r8(TMP) == c8(0x0))
9   r32(EDI) := ...
10  r32(ECX) := (r32(ECX) + c32(-0x1))
11  JUMP ((r1(ZF) != c32(0x1))) None
12  r32(EAX) := r32(ECX)
13  r32(EAX) := ( r32(EAX))
14  r32(None) := (r32(EAX) + c32(-0x1))
15  INFO r32(None) := (strlen m32[(r32(EBP) + None)])
```



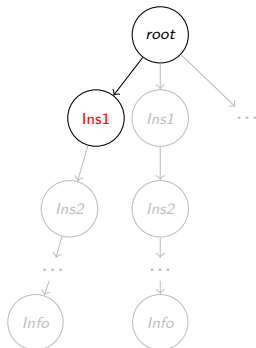
### Passo1: Generazione signature

Insieme di istruzioni in forma intermedia che identificano una funzione.



## Funzione strlen()

```
1  r32(ECX) := c32(0xffffffff)
2  m32[(r32(EBP) + None)] := r32(EAX)
3  r32(EAX) := c32(0x0)
4  r1(DF) := c1(0x0)
5  r32(EDI) := m32[(r32(EBP) + None)]
6  JUMP ((r32(ECX) == c32(0x0))) None
7  r8(TMP) := (m32[(r16(ES) + r32(EDI))] - r8(AL))
8  r1(ZF) := (r8(TMP) == c8(0x0))
9  r32(EDI) := ...
10 r32(ECX) := (r32(ECX) + c32(-0x1))
11 JUMP ((r1(ZF) != c32(0x1))) None
12 r32(EAX) := r32(ECX)
13 r32(EAX) := ( r32(EAX))
14 r32(None) := (r32(EAX) + c32(-0x1))
15 INFO r32(None) := (strlen m32[(r32(EBP) + None)])
```

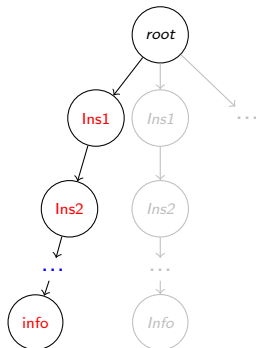


## Passo2: Creazione albero dei suffissi

Struttura dati utilizzata per un veloce confronto tra le istruzioni in forma intermedia.

## Funzione strlen()

```
1   r32(ECX) := c32(0xffffffff)
2   m32[(r32(EBP) + None)] := r32(EAX)
3   r32(EAX) := c32(0x0)
4   r1(DF) := c1(0x0)
5   r32(EDI) := m32[(r32(EBP) + None)]
6   JUMP ((r32(ECX) == c32(0x0))) None
7   r8(TMP) := (m32[(r16(ES) + r32(EDI))] - r8(AL))
8   r1(ZF) := (r8(TMP) == c8(0x0))
9   r32(EDI) := ...
10  r32(ECX) := (r32(ECX) + c32(-0x1))
11  JUMP ((r1(ZF) != c32(0x1))) None
12  r32(EAX) := r32(ECX)
13  r32(EAX) := ( r32(EAX))
14  r32(None) := (r32(EAX) + c32(-0x1))
15  INFO r32(None) := (strlen m32[(r32(EBP) + None)])
```

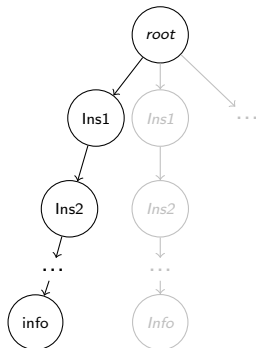


## Passo2: Creazione albero dei suffissi

Struttura dati utilizzata per un veloce confronto tra le istruzioni in forma intermedia.

## Funzione strlen()

```
1   r32(ECX) := c32(0xffffffff)
2   m32[(r32(EBP) + None)] := r32(EAX)
3   r32(EAX) := c32(0x0)
4   r1(DF) := c1(0x0)
5   r32(EDI) := m32[(r32(EBP) + None)]
6   JUMP ((r32(ECX) == c32(0x0))) None
7   r8(TMP) := (m32[(r16(ES) + r32(EDI))] - r8(AL))
8   r1(ZF) := (r8(TMP) == c8(0x0))
9   r32(EDI) := ...
10  r32(ECX) := (r32(ECX) + c32(-0x1))
11  JUMP ((r1(ZF) != c32(0x1))) None
12  r32(EAX) := r32(ECX)
13  r32(EAX) := ( r32(EAX))
14  r32(None) := (r32(EAX) + c32(-0x1))
15  INFO r32(None) := (strlen m32[(r32(EBP) + None)])
```



### Passo3: Riconoscimento funzioni inline

Confronto tra le istruzioni in forma intermedia e individuazione delle funzioni inline.

```
1 void cp(char *src)
2 {
3   char buffer[80];
4   if (src[0]!='c' || src[1]!='p')
5     {
6       printf("error\n");
7     }
8   else if(strlen(src) > 10)
9     {
10    strcpy(buffer, src);
11    }
12 }
```

↓

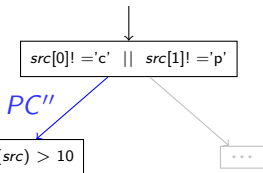
$src[0] \neq 'c' \ \vee \ src[1] \neq 'p'$

$PC = \emptyset$  inizialmente  $PC$  è vuoto.  
 $PC' = PC \cup \{src[0] \neq 'p' \wedge src[1] \neq 'c'\}$   
 $PC'' = PC \cup \{src[0] == 'p' \wedge src[1] == 'c'\}$

# Path condition

```
1 void cp(char *src)
2 {
3   char buffer[80];
4   if (src[0]!='c' || src[1]!='p')
5     {
6       printf("error\n");
7     }
8   else if(strlen(src) > 10)
9     {
10      strcpy(buffer, src);
11    }
12 }
```

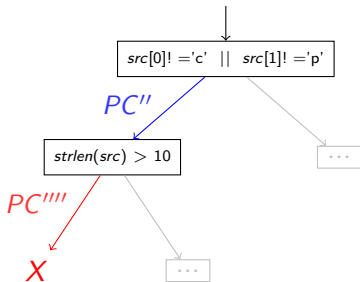
funzione inline



$$PC'' = PC \cup \{src[0] == 'p' \wedge src[1] == 'c'\}$$
$$PC''' = PC'' \cup \{strlen(src) \leq 10\}$$
$$PC'''' = PC'' \cup \{strlen(src) > 10\}$$

# Path condition

```
1 void cp(char *src)
2 {
3   char buffer[80];
4   if (src[0]!='c' || src[1]!='p')
5     {
6       printf("error\n");
7     }
8   else if(strlen(src) > 10)
9     {
10      strcpy(buffer, src);
11    }
12 }
```



$$PC'''' = PC'' \cup \{ strlen(src) > 10 \}$$
$$X = \text{vulnerabilità.}$$

## Contributi

- Implementazione della tecnica di riconoscimento di funzioni inline;
- Modifica della fase di analisi delle *path condition*.

Nel complesso si ritiene che il modello *smart fuzzer* rappresenti un valido punto di partenza da estendere e sviluppare ulteriormente nel prossimo futuro.

Grazie per l'attenzione