Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics

Erick Bauman\textsuperscript{1}, Zhiqiang Lin\textsuperscript{1,2}, Kevin Hamlen\textsuperscript{1}

\textsuperscript{1}University of Texas at Dallas
\textsuperscript{2}The Ohio State University

NDSS 2018
Static Binary Rewriting

Diagram: Binary code elements

1 0 1 0
1 0 1 0
1 0 1 0
1 0 1 0
1 0 1 0
1 0 1 0
1 0 1 0
Static Binary Rewriting
Many Static Rewriters Have Been Developed Over the Past Decades

<table>
<thead>
<tr>
<th>Systems</th>
<th>Year</th>
<th>L</th>
<th>D</th>
<th>S</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>D</th>
<th>P</th>
<th>O</th>
<th>R</th>
<th>A</th>
<th>F</th>
<th>R</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETCH</td>
<td>1997</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SASI</td>
<td>1999</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PLTO</td>
<td>2001</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VULCAN</td>
<td>2001</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DIABLO</td>
<td>2005</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CFI</td>
<td>2005</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XFI</td>
<td>2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PittSField</td>
<td>2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BIRD</td>
<td>2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NACl</td>
<td>2009</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PEBIL</td>
<td>2010</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SECONDWrite</td>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DYNInst</td>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>StIR/REINS</td>
<td>2012</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CCFIR</td>
<td>2013</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BISTRO</td>
<td>2013</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BinCFI</td>
<td>2013</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PSI</td>
<td>2014</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>UROBOROS</td>
<td>2016</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RAMBLR</td>
<td>2017</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Many Static Rewriters Have Been Developed Over the Past Decades

<table>
<thead>
<tr>
<th>Systems</th>
<th>Year</th>
<th>RL</th>
<th>DS</th>
<th>SH</th>
<th>PH</th>
<th>CHON</th>
<th>PROP</th>
<th>HR</th>
<th>AC</th>
<th>FRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etch</td>
<td>1997</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SASI</td>
<td>1999</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PLTO</td>
<td>2001</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Vulcan</td>
<td>2001</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Diablo</td>
<td>2005</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CFI</td>
<td>2005</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XFI</td>
<td>2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pittsburgh</td>
<td>2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bird</td>
<td>2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NAcl</td>
<td>2009</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PEBIL</td>
<td>2010</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SecondWrite</td>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DYNINST</td>
<td>2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Stir/Reins</td>
<td>2012</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CCFIR</td>
<td>2013</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bistro</td>
<td>2013</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BinCFI</td>
<td>2013</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Psi</td>
<td>2014</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Uroboros</td>
<td>2016</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RAMBLR</td>
<td>2017</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

These tools rely on various assumptions and heuristics!
MULTIVERSE: the first heuristic-free static binary rewriter

“Everything that can happen does happen.” [CF12]
Fundamental Challenges

1. Recognizing and relocating static memory addresses
2. Handling dynamically computed memory addresses
3. Differentiating code and data
4. Handling function pointer arguments (e.g., callbacks)
5. Handling PIC
Working Example

```c
1 // gcc -m32 -o sort cmp.o fstring.o sort.c
2 #include <stdio.h>
3 #include <unistd.h>
4
5 extern char *array[6];
6 int gt(void *, void *);
7 int lt(void *, void *);
8 char* get_fstring(int select);
9
10 void model(void){
11    qsort(array, 5, sizeof(char*), gt);
12 }
13 void mode2(void){
14    qsort(array, 5, sizeof(char*), lt);
15 }
16
17 void (*modes[2])() = {model, mode2};
18
19 void main(void){
20    int p = getpid() & 1;
21    printf(get_fstring(0),p);
22    (*modes[p])();
23    print_array();
24 }
```

(a) Source code of `sort.c`
Working Example

```
1 ;nasm -f elf fstring.asm
2 BITS 32
3 GLOBAL get_fstring
4 SECTION .text
5 get_fstring:
6  mov eax, [esp+4]
7  cmp eax, 0
8  jz after
9  mov eax, msg2
10  ret
11 msg1:
12  db 'mode: %d', 10, 0
13 msg2:
14  db '%s', 10, 0
15 after:
16  mov eax, msg1
17  ret
```

(b) Source code of fstring.asm
## Working Example

```c
1 // gcc -m32 -c -o cmp.o cmp.c -fPIC -O2
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
7 char* get_fstring(int select);
8
9 void print_array(){
10    int i;
11    for (i = 0; i < 5; i++){
12       fprintf(stdout, get_fstring(1), array[i]);
13    }
14 }
15
16 int lt(void *a, void *b){
17    return strcmp(*(char **) a, *(char **)b);
18 }
19
20 int gt(void *a, void *b){
21    return strcmp(*(char **) b, *(char **)a);
22 }
```

(c) Source code of `cmp.c`
Challenge (C)1: Recognizing and relocating static addresses

```c
1  // gcc -m32 -o sort cmp.o fstring.o sort.c
2  #include <stdio.h>
3  #include <unistd.h>
4  
5  extern char *array[6];
6  int gt(void *, void *);
7  int lt(void *, void *);
8  char* get_fstring(int select);
9  
10 void model(void){
11    qsort(array, 5, sizeof(char*), gt);
12 }
13 void mode2(void){
14    qsort(array, 5, sizeof(char*), lt);
15 }
16
17 void (*modes[2])() = {model, mode2};
18
19 void main(void){
20    int p = getpid() & 1;
21    printf(get_fstring(0),p);
22    (*modes[p])();
23    print_array();
24 }
```

(a) Source code of `sort.c`
Challenge (C)1: Recognizing and relocating static addresses

Hex dump of section `.data`:

```
0x0804a01c 00000000 00000000 70870408 74870408 ........p...t...
0x0804a02c 78870408 7d870408 81870408 00000000 x...}.........
0x0804a03c f4850408 20860408 .......
```

(f) Hexdump of `.data` section
C1: Recognizing and relocating static memory addresses

- Data may contain function pointers
- Must identify pointers to transformed code
- Difficult to reliably distinguish pointer-like integers from pointers
C1: Recognizing and relocating static memory addresses

- Data may contain function pointers
- Must identify pointers to transformed code
- Difficult to reliably distinguish pointer-like integers from pointers

Keeping original data space intact

- No need to modify data addresses if data unchanged
- Keep read-only copy of code for inline data in original code section [OAK\textsuperscript{+}11, ZS13, WMHL12b, WMHL12a]
C2: Handling dynamically computed memory addresses

```
1 // gcc -m32 -o sort cmp.o fstring.o sort.c
2 #include <stdio.h>
3 #include <unistd.h>
4
5 extern char *array[6];
6 int gt(void *, void *);
7 int lt(void *, void *);
8 char* get_fstring(int select);
9
10 void model(void){
11    qsort(array, 5, sizeof(char*), gt);
12 }
13 void mode2(void){
14    qsort(array, 5, sizeof(char*), lt);
15 }
16
17 void (*modes[2])() = {model, mode2};
18
19 void main(void){
20    int p = getpid() & 1;
21    printf(get_fstring(0), p);
22    (*modes[p])();
23    print_array();
24 }
```
C2: Handling dynamically computed memory addresses

804864c <main>:
...
8048678:  e8 73 fd ff ff  call 80483f0 <printf@plt>
804867d:  8b 44 24 1c  mov 0x1c(%esp),%eax
8048681:  8b 04 85 3c a0 04 08  mov 0x804a03c(%eax,4),%eax
8048688:  ff d0  call *%eax
...

(d) Partial binary code of sort
C2: Handling dynamically computed memory addresses

- Indirect control flow transfer (iCFT) targets computed at runtime
- May use base+offset or arbitrary arithmetic
- Difficult to predict iCFT targets statically
C2: Handling dynamically computed memory addresses

- Indirect control flow transfer (iCFT) targets computed at runtime
- May use base+offset or arbitrary arithmetic
- Difficult to predict iCFT targets statically

Creating mapping from old code space to rewritten code space

- Do not attempt to identify original addresses to rewrite
- Ignore how address is computed; only focus on final target
- Rewrite all iCFTs to use mapping to dynamically translate address on use [PCC+04]
C3: Differentiating code and data

```assembly
1 ;nasm -f elf fstring.asm
2 BITS 32
3 GLOBAL get_fstring
4 SECTION .text
5 get_fstring:
6   mov eax, [esp+4]
7   cmp eax, 0
8   jz after
9   mov eax, msg2
10  ret
11 msg1:
12     db 'mode: %d', 10, 0
13 msg2:
14     db '%s', 10, 0
15 after:
16   mov eax, msg1
17  ret
```

(b) Source code of fstring.asm
C3: Differentiating code and data

(d) Partial binary code of sort
C3: Differentiating code and data

- Code and data can be freely interleaved
- Found in hand-written assembly and optimizing compilers
- Linear sweep fails on inline data
- Recursive traversal lacks full coverage
C3: Differentiating code and data

- Code and data can be freely interleaved
- Found in hand-written assembly and optimizing compilers
- Linear sweep fails on inline data
- Recursive traversal lacks full coverage

Brute force disassembling of all possible code

- Disassemble every offset \([\text{KRVV04, WZHKL}^{+15}]\)
- All intended code will be within resulting superset
C4: Handling function pointer arguments (e.g., callbacks)

(a) Source code of sort.c

```c
1 // gcc -m32 -o sort cmp.o fstring.o sort.c
2 #include <stdio.h>
3 #include <unistd.h>
4
5 extern char *array[6];
6 int gt(void *, void *);
7 int lt(void *, void *);
8 char* get_fstring(int select);
9
10 void model(void){
11     qsort(array, 5, sizeof(char*), gt);
12 }
13 void mode2(void){
14     qsort(array, 5, sizeof(char*), lt);
15 }
16
17 void (*modes[2])() = {model, mode2};
18
19 void main(void){
20     int p = getpid() & 1;
21     printf(get_fstring(0),p);
22     (*modes[p])();
23     print_array();
24 }
```
C4: Handling function pointer arguments (e.g., callbacks)

(d) Partial binary code of sort
C4: Handling function pointer arguments (e.g., callbacks)

- Callbacks will fail if function pointer not updated
- Library code uses callbacks
- Difficult to identify function pointer arguments
C4: Handling function pointer arguments (e.g., callbacks)

- Callbacks will fail if function pointer not updated
- Library code uses callbacks
- Difficult to identify function pointer arguments

Rewriting all user level code including libraries

- Hard to automatically identify all function pointer arguments
- Instead, rewrite everything [ZS13]
- Use mapping (from Solution 2) to translate callback upon use
C5: Handling PIC

```c
1 // gcc -m32 -c -o cmp.o cmp.c -fPIC -O2
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
7 char* get_fstring(int select);
8
9 void print_array()
10 {
11     int i;
12     for (i = 0; i < 5; i++){
13         printf(stdout, get_fstring(1), array[i]);
14     }
15 }
16 int lt(void *a, void *b){
17     return strcmp(*(char **) a, *(char **)b);
18 }
19 int gt(void *a, void *b){
20     return strcmp(*(char **) b, *(char **)a);
21 }
```

(c) Source code of cmp.c
C5: Handling PIC

(d) Partial binary code of sort
C5: Handling PIC

- Position-independent code (PIC) can be loaded at arbitrary address
- Dynamically calculates relative offsets
- Offsets different for modified code
C5: Handling PIC

- Position-independent code (PIC) can be loaded at arbitrary address
- Dynamically calculates relative offsets
- Offsets different for modified code

Rewriting all call instructions

- For x86-32 instructions, only `call` reveals instruction pointer
- Rewrite `call` to `push/jmp` and push old return address [ZS13, CBG17]
- Offsets computed based on old address
- From Solution 2, rewritten `ret` instructions translate return address with mapping
MULTIVERSE

Mapping Phase

1. Mapping Phase
   - Disassemble starting from every byte
   - Determine lengths of rewritten instructions
   - Create mapping from original address to rewritten address

2. Rewriting Phase
   - Translate instructions to rewritten forms
   - Use mapping to determine final addresses

Instruction Rewriter

Superset Disassembler

Original Executable, Shared Library

New Executable, Shared Library

ELF
- .text
- .rodata
- .got
- .got.plt
- .data

ELF
- .text
- .rodata
- .got
- .got.plt
- .data

.newtext

.localmapping
MULTIVERSE

1. Mapping Phase
   ▶ Disassemble starting from every byte

- **Original Executable, Shared Library**
  - ELF
    - .text
    - .rodata
    - .got
    - .got.plt
    - .data

- **Mapping Phase**
  - Superset Disassembler
    - Instruction Rewriter
      - Rewriting Phase
        - New Executable, Shared Library
          - ELF
            - .text
            - .rodata
            - .got
            - .got.plt
            - .data
            - .newtext
            - .localmapping
**MULTIVERSE**

**Original Executable, Shared Library**

- ELF
  - .text
  - .rodata
  - .got
  - .got.plt
  - .data

**Mapping Phase**

- Superset Disassembler
- Instruction Rewriter

**New Executable, Shared Library**

- ELF
  - .text
  - .rodata
  - .got
  - .got.plt
  - .data
  - .newtext
  - .localmapping

**Rewriting Phase**

**Mapping Phase**

- Disassemble starting from every byte
- Determine lengths of rewritten instructions
MULTIVERSE

Original Executable, Shared Library

Mapping Phase

Superset Disassembler

Instruction Rewriter

Rewriting Phase

New Executable, Shared Library

Mapping Phase

- Disassemble starting from every byte
- Determine lengths of rewritten instructions
- Create mapping from original address to rewritten address
MULTIVERSE

1. Mapping Phase
   - Disassemble starting from every byte
   - Determine lengths of rewritten instructions
   - Create mapping from original address to rewritten address

2. Rewriting Phase
**MULTIVERSE**

- **Original Executable, Shared Library**
  - ELF
    - .text
    - .rodata
    - .got
    - .got.plt
    - .data

- **Mapping Phase**
  - Superset Disassembler
  - Instruction Rewriter

- **New Executable, Shared Library**
  - ELF
    - .text
    - .rodata
    - .got
    - .got.plt
    - .data
    - .newtext
    - .localmapping

---

1. **Mapping Phase**
   - Disassemble starting from every byte
   - Determine lengths of rewritten instructions
   - Create mapping from original address to rewritten address

2. **Rewriting Phase**
   - Translate instructions to rewritten forms
MULTIVERSE

Original Executable, Shared Library → Mapping Phase

Superset Disassembler

Instruction Rewriter → Rewriting Phase

New Executable, Shared Library

1. Mapping Phase
   - Disassemble starting from every byte
   - Determine lengths of rewritten instructions
   - Create mapping from original address to rewritten address

2. Rewriting Phase
   - Translate instructions to rewritten forms
   - Use mapping to determine final addresses
Superset Disassembly

Algorithm 1: Superset Disassembly

```
input : empty two-dimensional list instructions
input : string of raw bytes of text section bytes
output: all disassembled instructions are in instructions
1 for start_offset ← 0 to length(bytes) do
2     offset ← start_offset;
3     while legal(offset) and offset ∉ instructions and
4         offset < length(bytes) do
5         instruction ← disassemble(offset);
6         instructions[start_offset][offset] ← instruction;
7         offset ← offset + length(instruction);
8     if offset ∈ instructions then
9         instructions[start_offset][offset] ← “jmp
10        offset”;
```
Superset Disassembly

The Algorithm

1. **Start disassembly at first byte**
2. **Disassemble until encounters one of:**
   - Invalid instruction encoding
   - Already disassembled offset
   - End of byte sequence

**Algorithm 1: Superset Disassembly**

- **input**: empty two-dimensional list *instructions*
- **input**: string of raw bytes of text section *bytes*
- **output**: all disassembled instructions are in *instructions*

```plaintext
1 for start_offset ← 0 to length(bytes) do
    offset ← start_offset;
    2 while legal(offset) and offset ∉ instructions and
        offset < length(bytes) do
        3 instruction ← disassemble(offset);
        4 instructions[start_offset][offset] ← instruction;
        5 offset ← offset + length(instruction);
    6 if offset ∈ instructions then
        7 instructions[start_offset][offset] ← “jmp
        8 offset”;
```
# Superset Disassembly

## Algorithm 1: Superset Disassembly

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Start disassembly at first byte</td>
</tr>
<tr>
<td>2.</td>
<td>Disassemble until encounters one of:</td>
</tr>
<tr>
<td></td>
<td>- Invalid instruction encoding</td>
</tr>
<tr>
<td></td>
<td>- Already disassembled offset</td>
</tr>
<tr>
<td></td>
<td>- End of byte sequence</td>
</tr>
<tr>
<td>3.</td>
<td>If offset in previous sequence, jump to the sequence</td>
</tr>
</tbody>
</table>

The Algorithm

1. Start disassembly at first byte
2. Disassemble until encounters one of:
   - Invalid instruction encoding
   - Already disassembled offset
   - End of byte sequence
3. If offset in previous sequence, jump to the sequence
Superset Disassembly

The Algorithm

1. Start disassembly at first byte
2. Disassemble until encounters one of:
   - Invalid instruction encoding
   - Already disassembled offset
   - End of byte sequence
3. If offset in previous sequence, jump to the sequence
4. If not at end of byte sequence, start disassembly from next byte

Algorithm 1: Superset Disassembly

\[
\text{input} : \text{empty two-dimensional list instructions} \\
\text{input} : \text{string of raw bytes of text section bytes} \\
\text{output} : \text{all disassembled instructions are in instructions}
\]

1. \(\text{for} \ \text{start_offset} \leftarrow 0 \ \text{to length(bytes)} \ \text{do}\)
2. \(\text{offset} \leftarrow \text{start_offset};\)
3. \(\text{while} \ \text{legal(offset)} \ \text{and} \ \text{offset} \notin \text{instructions and} \ \text{offset} < \text{length(bytes)} \ \text{do}\)
   4. \(\text{instruction} \leftarrow \text{disassemble(offset)};\)
   5. \(\text{instructions[start_offset][offset]} \leftarrow \text{instruction};\)
   6. \(\text{offset} \leftarrow \text{offset} + \text{length(instruction)};\)
4. \(\text{if} \ \text{offset} \in \text{instructions then}\)
5. \(\text{instructions[start_offset][offset]} \leftarrow \text{“jmp offset”;}\)
Superset Disassembly

The Algorithm

1. Start disassembly at first byte
2. Disassemble until encounters one of:
   - Invalid instruction encoding
   - Already disassembled offset
   - End of byte sequence
3. If offset in previous sequence, jump to the sequence
4. If not at end of byte sequence, start disassembly from next byte
5. Go to ②
Superset Disassembly

Offset 0
Superset Disassembly
Superset Disassembly
Superset Disassembly
Superset Disassembly
Superset Disassembly
Superset Disassembly
Superset Disassembly
Mapping Lookups
Mapping Lookups

1.

.globalmapping
.localmapping
.local_lookup
.global_lookup
.newtext
.data
.text
Mapping Lookups

1. local_lookup
2. local_lookup

.globalmapping
.localmapping

.text
.data
.newtext

.globalmapping
.localmapping

.text (libc)
.data (libc)
.newtext (libc)
.localmapping (libc)
Mapping Lookups

1. local_lookup
2. global_lookup
3. .text (libc)
   .data (libc)
   .newtext
   .localmapping

- .text
- .data
- global_lookup
  .globalmapping
- local_lookup
  .newtext (libc)
  .localmapping (libc)
Mapping Lookups

Diagram showing the flow of lookup operations in different sections of the program:

1. `.text` to `local_lookup`
2. `global_lookup` to `.globalmapping`
3. `.globalmapping` to `local_lookup` in `.newtext`
4. `local_lookup` in `.globalmapping` to `local_lookup` in `.newtext (libc)`
Mapping Lookups

Diagram showing the process of mapping lookups in a program, with labels for `.text`, `.data`, `.newtext`, `.localmapping`, `.globalmapping`, and `.text (libc)`.
Mapping Lookups

1. local_lookup
2. global_lookup
3. .globalmapping
4. .newtext
5. .localmapping (libc)
6. .newtext (libc)
7. .data (libc)
8. .text (libc)
Optimizations

- Lack of assumptions increases overhead
- For well-behaved binaries it is safe to relax constraints
Optimizations

- Lack of assumptions increases overhead
- For well-behaved binaries it is safe to relax constraints

Optimization 1: Only Rewrite Main Binary
- If only the main binary is of interest
- Requires list of library callback functions
Optimizations

- Lack of assumptions increases overhead
- For well-behaved binaries it is safe to relax constraints

Optimization 1: Only Rewrite Main Binary
- If only the main binary is of interest
- Requires list of library callback functions

Optimization 2: No Generic PIC
- Assume only PIC is via `get_pc_thunk`
- True for many binaries
- Significant performance increase for compatible binaries
Instruction Counting

- Ultimate purpose of a rewriter is to insert instrumentation code
Instrumentation Evaluation

Instruction Counting

- Ultimate purpose of a rewriter is to insert instrumentation code
- Created straightforward instrumentation API
Instrumentation Evaluation

Instruction Counting

- Ultimate purpose of a rewriter is to insert instrumentation code
- Created straightforward instrumentation API
- For evaluation created instruction counting instrumentation in MULTIVERSE
Instrumentation Evaluation

**Instruction Counting**

- Ultimate purpose of a rewriter is to insert instrumentation code
- Created straightforward instrumentation API
- For evaluation created instruction counting instrumentation in **MULTIVERSE**
- Compared with instruction counting Pintools
Instrumentation Overhead

The graph shows the instrumentation overhead for various benchmarks under different conditions. The y-axis represents the overhead in terms of speedup, with values ranging from 0x to 25.3x. The x-axis lists the benchmarks.

- **MULTIVERSE w/ Binary Only**
- **MULTIVERSE w/ Binary Only w/o Generic PIC**
- **Pintool**
- **Pintool w/ Binary Only**

The overhead values vary significantly across different benchmarks, with some showing a slight increase and others a more substantial drop. The overhead is measured compared to a baseline performance without instrumentation.
An appealing application of rewriters is binary hardening.
Security Applications Evaluation

Shadow Stack

- An appealing application of rewriters is binary hardening
- Shadow stacks implement a form of backward-edge CFI
An appealing application of rewriters is binary hardening.
Shadow stacks implement a form of backward-edge CFI.
Implemented a simple shadow stack in MULTIVERSE.
### Security Applications Evaluation

#### Shadow Stack

- An appealing application of rewriters is binary hardening
- Shadow stacks implement a form of backward-edge CFI
- Implemented a simple shadow stack in **MULTIVERSE**
- Compared with same type of shadow stack using **PIN**
Limitations and Future Work

x86-64 Support

- Paper only covers 32-bit support
- MULTIVERSE now supports 64-bit applications
Limitations and Future Work

x86-64 Support

- Paper only covers 32-bit support
- MULTIVERSE now supports 64-bit applications

Optimization

- MULTIVERSE focuses on generality
- Overhead in some cases is high
- Still room for performance improvements in future
Limitations and Future Work

**x86-64 Support**
- Paper only covers 32-bit support
- MULTIVERSE now supports 64-bit applications

**Optimization**
- MULTIVERSE focuses on generality
- Overhead in some cases is high
- Still room for performance improvements in future

**Instrumentation API**
- For paper, used simple instruction-level API
- Currently working on more robust API
Conclusion

**MULTIVERSE**
- Heuristic-free static rewriter
- Works for x86/64 binaries
- Useful for many security applications (e.g., hardening)

**MULTIVERSE Source Code**
github.com/utds3lab/multiverse
Thank You
References I


References II


