Static Binary Rewriting

Zhiqiang Lin

Department of Computer Science and Engineering
The Ohio State University

7/19/2018

Static binary rewriting is important

Applications

1. Software fault isolation (SFI) [WLAG93]
2. Control Flow Integrity (CFI) [ABEL09]
3. Binary code hardening (e.g., STIR [WMHL12a])
4. Binary code reuse (e.g., BCR [CJMS10])
5. Platform-specific optimizations [ASE+13]

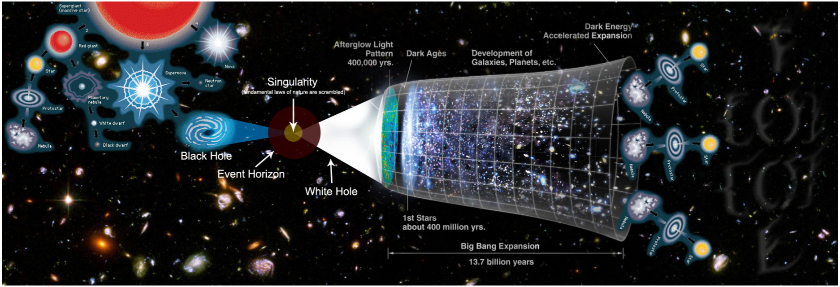
Challenges in disassembling

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

Existing static rewriters: w/ assumptions and heuristics

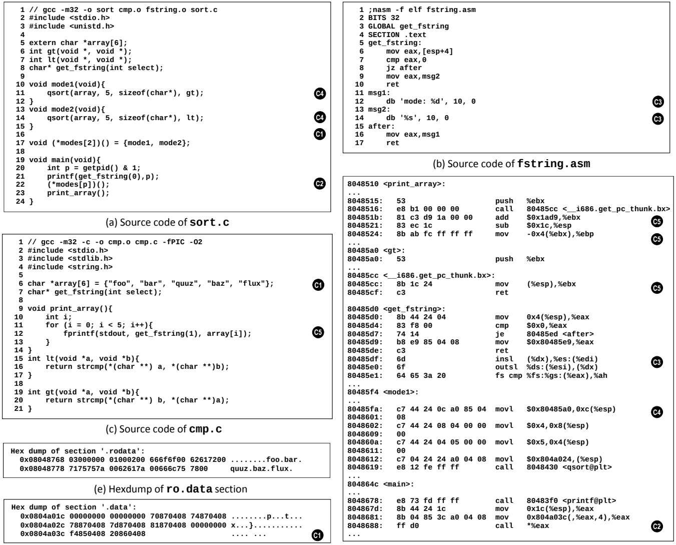
1. Assume certain compiler (e.g., gcc) generated binaries
2. Assume having debugging symbols
3. Assume knowledge of APIs (call backs)
4. Assume no code and data interleaving
5. Assume relocation metadata
6. Assume integer and pointer can be differentiated
7. ...

MULTIVERSE: the first heuristic-free static binary rewriter


Supergiant (massive star)
Red giant
Star
Protostar
Nebula
Planetary nebula
White dwarf
Black dwarf
Supernova
Neutron star
Nova
1
2
3
4
Black Hole
Event Horizon
Singularity
(fundamental laws of nature are scrambled)
White Hole
Afterglow Light Pattern 400,000 yrs.
Dark Ages
Development of Galaxies, Planets, etc.
Dark Energy Accelerated Expansion
1st Stars about 400 million yrs.
Big Bang Expansion
13.7 billion years
Star
Protostar
Nebula

"Everything that can happen does happen." [CF12]

A Running Example


C4
C4
C1
C2
C1
C5
C1
C3
C3
C5
C5
C5
C3
C4
C2

```
// gcc -m32 -o sort cmp.o fstring.o sort.c
#include <stdio.h>
#include <unistd.h>

extern char *array[6];
int gt(void *, void *);
int lt(void *, void *);
char* get_fstring(int select);

void mode1(void){
    qsort(array, 5, sizeof(char*), gt);
}
void mode2(void){
    qsort(array, 5, sizeof(char*), lt);
}

void (*modes[2])() = {mode1, mode2};

void main(void){
    int p = getpid() & 1;
    printf(get_fstring(0),p);
    (*modes[p])();
    print_array();
}
```

(a) Source code of **sort.c**

```
// gcc -m32 -c -o cmp.o cmp.c -fPIC -O2
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *array[6] = {"foo", "bar", "quuz", "baz", "flux"};
char* get_fstring(int select);

void print_array(){
    int i;
    for (i = 0; i < 5; i++){
        fprintf(stdout, get_fstring(1), array[i]);
    }
}
int lt(void *a, void *b){
    return strcmp(*(char **) a, *(char **)b);
}

int gt(void *a, void *b){
    return strcmp(*(char **) b, *(char **)a);
}
```

(c) Source code of **cmp.c**

```
Hex dump of section '.rodata':
  0x08048768 03000000 01000200 666f6f00 62617200 ........foo.bar.
  0x08048778 7175757a 0062617a 00666c75 7800     quuz.baz.flux.
```

(e) Hexdump of **ro.data** section

```
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

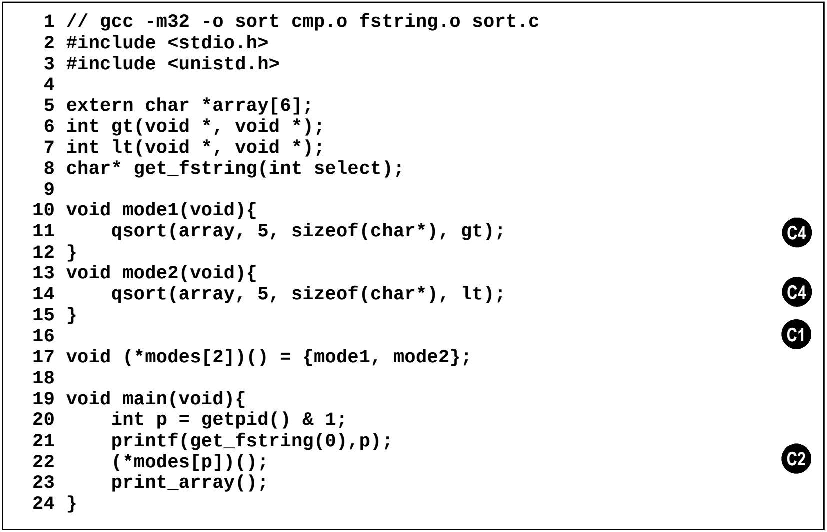
```
;nasm -f elf fstring.asm
BITS 32
GLOBAL get_fstring
SECTION .text
get_fstring:
    mov eax,[esp+4]
    cmp eax,0
    jz after
    mov eax,msg2
    ret
msg1:
    db 'mode: %d', 10, 0
msg2:
    db '%s', 10, 0
after:
    mov eax,msg1
    ret
```

(b) Source code of **fstring.asm**

```
8048510 <print_array>:
...
8048515:   53                      push   %ebx
8048516:   e8 b1 00 00 00          call   80485cc <__i686.get_pc_thunk.bx>
804851b:   81 c3 d9 1a 00 00       add    $0x1ad9,%ebx
8048521:   83 ec 1c                sub    $0x1c,%esp
8048524:   8b ab fc ff ff ff       mov    -0x4(%ebx),%ebp
...
80485a0 <gt>:
80485a0:   53                      push   %ebx
...
80485cc <__i686.get_pc_thunk.bx>:
80485cc:   8b 1c 24                mov    (%esp),%ebx
80485cf:   c3                      ret

80485d0 <get_fstring>:
80485d0:   8b 44 24 04             mov    0x4(%esp),%eax
80485d4:   83 f8 00                cmp    $0x0,%eax
80485d7:   74 14                   je     80485ed <after>
80485d9:   b8 e9 85 04 08          mov    $0x80485e9,%eax
80485de:   c3                      ret
80485df:   6d                      insl   (%dx),%es:(%edi)
80485e0:   6f                      outsl  %ds:(%esi),(%dx)
80485e1:   64 65 3a 20             fs cmp %fs:%gs:(%eax),%ah
...
80485f4 <mode1>:
...
80485fa:   c7 44 24 0c a0 85 04    movl   $0x80485a0,0xc(%esp)
8048601:   08
8048602:   c7 44 24 08 04 00 00    movl   $0x4,0x8(%esp)
8048609:   00
804860a:   c7 44 24 04 05 00 00    movl   $0x5,0x4(%esp)
8048611:   00
8048612:   c7 04 24 24 a0 04 08    movl   $0x804a024,(%esp)
8048619:   e8 12 fe ff ff          call   8048430 <qsort@plt>
...
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**

A Running Example



```
// gcc -m32 -o sort cmp.o fstring.o sort.c
#include <stdio.h>
#include <unistd.h>

extern char *array[6];
int gt(void *, void *);
int lt(void *, void *);
char* get_fstring(int select);

void mode1(void){
    qsort(array, 5, sizeof(char*), gt);
}
void mode2(void){
    qsort(array, 5, sizeof(char*), lt);
}

void (*modes[2])() = {mode1, mode2};

void main(void){
    int p = getpid() & 1;
    printf(get_fstring(0),p);
    (*modes[p])();
    print_array();
}
```

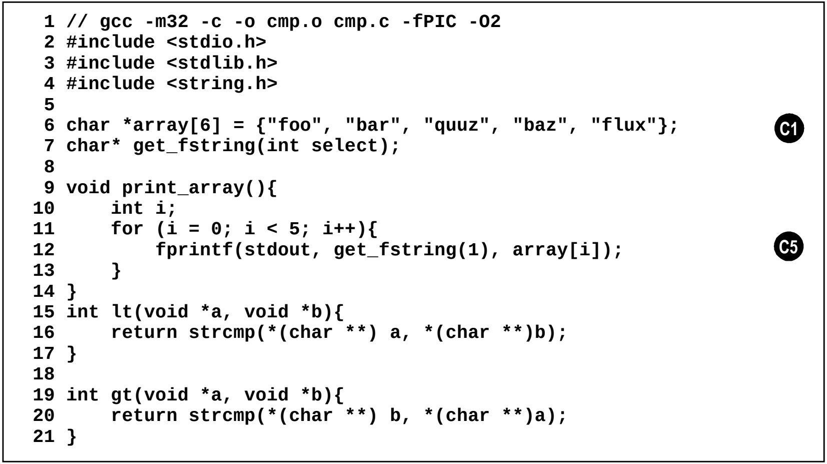
(a) Source code of **sort.c**

A Running Example

```
;nasm -f elf fstring.asm
BITS 32
GLOBAL get_fstring
SECTION .text
get_fstring:
    mov eax,[esp+4]
    cmp eax,0
    jz after
    mov eax,msg2
    ret
msg1:
    db 'mode: %d', 10, 0                                  C3
msg2:
    db '%s', 10, 0                                        C3
after:
    mov eax,msg1
    ret
```

(b) Source code of **fstring.asm**

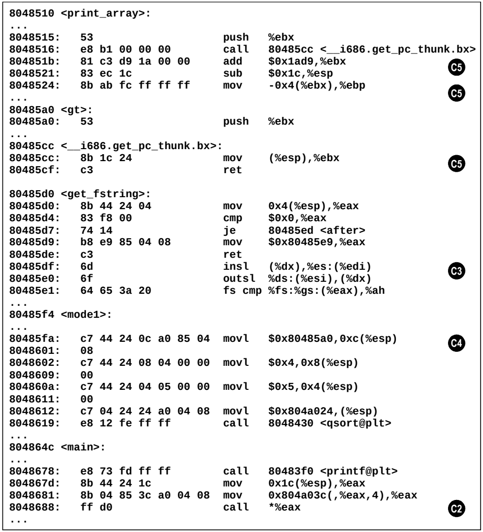
A Running Example



```
// gcc -m32 -c -o cmp.o cmp.c -fPIC -O2
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *array[6] = {"foo", "bar", "quuz", "baz", "flux"};
char* get_fstring(int select);

void print_array(){
    int i;
    for (i = 0; i < 5; i++){
        fprintf(stdout, get_fstring(1), array[i]);
    }
}
int lt(void *a, void *b){
    return strcmp(*(char **) a, *(char **)b);
}

int gt(void *a, void *b){
    return strcmp(*(char **) b, *(char **)a);
}
```

(c) Source code of **cmp.c**

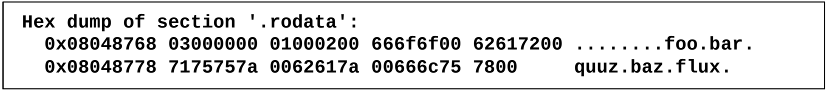
A Running Example



```
8048510 <print_array>:
...
8048515:   53                     push   %ebx
8048516:   e8 b1 00 00 00         call   80485cc <__i686.get_pc_thunk.bx>
804851b:   81 c3 d9 1a 00 00      add    $0x1ad9,%ebx                       C5
8048521:   83 ec 1c               sub    $0x1c,%esp
8048524:   8b ab fc ff ff ff      mov    -0x4(%ebx),%ebp                    C5
...
80485a0 <gt>:
80485a0:   53                     push   %ebx
...
80485cc <__i686.get_pc_thunk.bx>:
80485cc:   8b 1c 24               mov    (%esp),%ebx                        C5
80485cf:   c3                     ret

80485d0 <get_fstring>:
80485d0:   8b 44 24 04            mov    0x4(%esp),%eax
80485d4:   83 f8 00               cmp    $0x0,%eax
80485d7:   74 14                  je     80485ed <after>
80485d9:   b8 e9 85 04 08         mov    $0x80485e9,%eax
80485de:   c3                     ret
80485df:   6d                     insl   (%dx),%es:(%edi)                   C3
80485e0:   6f                     outsl  %ds:(%esi),(%dx)
80485e1:   64 65 3a 20            fs cmp %fs:%gs:(%eax),%ah
...
80485f4 <mode1>:
...
80485fa:   c7 44 24 0c a0 85 04   movl   $0x80485a0,0xc(%esp)               C4
8048601:   08
8048602:   c7 44 24 08 04 00 00   movl   $0x4,0x8(%esp)
8048609:   00
804860a:   c7 44 24 04 05 00 00   movl   $0x5,0x4(%esp)
8048611:   00
8048612:   c7 04 24 24 a0 04 08   movl   $0x804a024,(%esp)
8048619:   e8 12 fe ff ff         call   8048430 <qsort@plt>
...
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                              C2
...
```

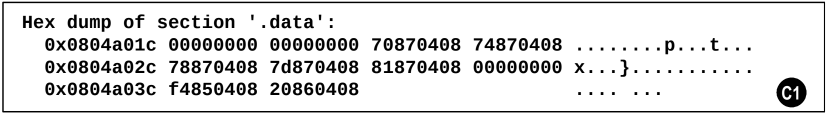
(d) Partial binary code of **sort**

A Running Example



```
Hex dump of section '.rodata':
  0x08048768 03000000 01000200 666f6f00 62617200 ........foo.bar.
  0x08048778 7175757a 0062617a 00666c75 7800     quuz.baz.flux.
```

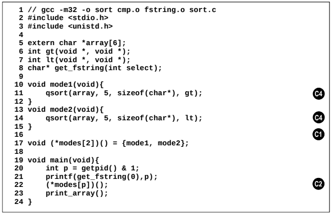
(e) Hexdump of **`ro.data`** section



```
Hex dump of section '.data':
  0x0804a01c 00000000 00000000 70870408 74870408 ........p...t...
  0x0804a02c 78870408 7d870408 81870408 00000000 x...}...........
  0x0804a03c f4850408 20860408                   .... ...
```

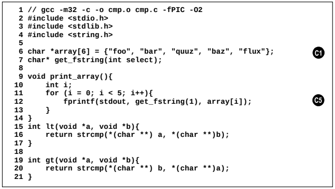
(f) Hexdump of **`.data`** section

A Running Example



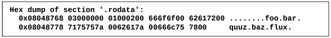
```
// gcc -m32 -o sort cmp.o fstring.o sort.c
#include <stdio.h>
#include <unistd.h>

extern char *array[6];
int gt(void *, void *);
int lt(void *, void *);
char* get_fstring(int select);

void mode1(void){
    qsort(array, 5, sizeof(char*), gt);                C4
}
void mode2(void){
    qsort(array, 5, sizeof(char*), lt);                C4
}                                                      
                                                       C1
void (*modes[2])() = {mode1, mode2};

void main(void){
    int p = getpid() & 1;
    printf(get_fstring(0),p);
    (*modes[p])();                                     C2
    print_array();
}
```

(a) Source code of **sort.c**



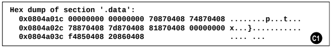
```
// gcc -m32 -c -o cmp.o cmp.c -fPIC -O2
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *array[6] = {"foo", "bar", "quuz", "baz", "flux"};      C1
char* get_fstring(int select);

void print_array(){
    int i;
    for (i = 0; i < 5; i++){
        fprintf(stdout, get_fstring(1), array[i]);           C5
    }
}
int lt(void *a, void *b){
    return strcmp(*(char **) a, *(char **)b);
}

int gt(void *a, void *b){
    return strcmp(*(char **) b, *(char **)a);
}
```

(c) Source code of **cmp.c**



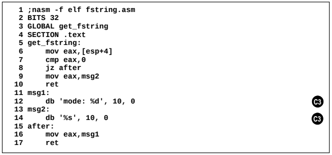
```
Hex dump of section '.rodata':
  0x08048768 03000000 01000200 666f6f00 62617200 ........foo.bar.
  0x08048778 7175757a 0062617a 00666c75 7800     quuz.baz.flux.
```

(e) Hexdump of **ro.data** section



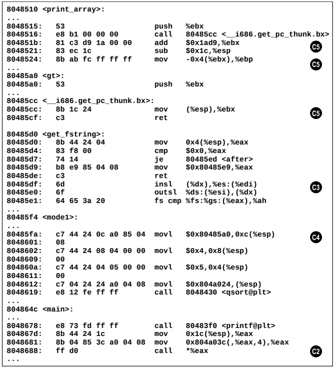
```
Hex dump of section '.data':
  0x0804a01c 00000000 00000000 70870408 74870408 ........p...t...
  0x0804a02c 78870408 7d870408 81870408 00000000 x...}...........
  0x0804a03c f4850408 20860408                   .... ...          C1
```

(f) Hexdump of **.data** section



```
;nasm -f elf fstring.asm
BITS 32
GLOBAL get_fstring
SECTION .text
get_fstring:
    mov eax,[esp+4]
    cmp eax,0
    jz after
    mov eax,msg2
    ret
msg1:
    db 'mode: %d', 10, 0                               C3
msg2:
    db '%s', 10, 0                                     C3
after:
    mov eax,msg1
    ret
```

(b) Source code of **fstring.asm**



```
8048510 <print_array>:
...
8048515:   53                     push   %ebx
8048516:   e8 b1 00 00 00         call   80485cc <__i686.get_pc_thunk.bx>
804851b:   81 c3 d9 1a 00 00      add    $0x1ad9,%ebx                  C5
8048521:   83 ec 1c               sub    $0x1c,%esp
8048524:   8b ab fc ff ff ff      mov    -0x4(%ebx),%ebp               C5
...
80485a0 <gt>:
80485a0:   53                     push   %ebx
...
80485cc <__i686.get_pc_thunk.bx>:
80485cc:   8b 1c 24               mov    (%esp),%ebx                   C5
80485cf:   c3                     ret

80485d0 <get_fstring>:
80485d0:   8b 44 24 04            mov    0x4(%esp),%eax
80485d4:   83 f8 00               cmp    $0x0,%eax
80485d7:   74 14                  je     80485ed <after>
80485d9:   b8 e9 85 04 08         mov    $0x80485e9,%eax
80485de:   c3                     ret
80485df:   6d                     insl   (%dx),%es:(%edi)              C3
80485e0:   6f                     outsl  %ds:(%esi),(%dx)
80485e1:   64 65 3a 20            fs cmp %fs:%gs:(%eax),%ah
...
80485f4 <mode1>:
...
80485fa:   c7 44 24 0c a0 85 04   movl   $0x80485a0,0xc(%esp)          C4
8048601:   08
8048602:   c7 44 24 08 04 00 00   movl   $0x4,0x8(%esp)
8048609:   00
804860a:   c7 44 24 04 05 00 00   movl   $0x5,0x4(%esp)
8048611:   00
8048612:   c7 04 24 24 a0 04 08   movl   $0x804a024,(%esp)
8048619:   e8 12 fe ff ff         call   8048430 <qsort@plt>
...
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                         C2
...
```

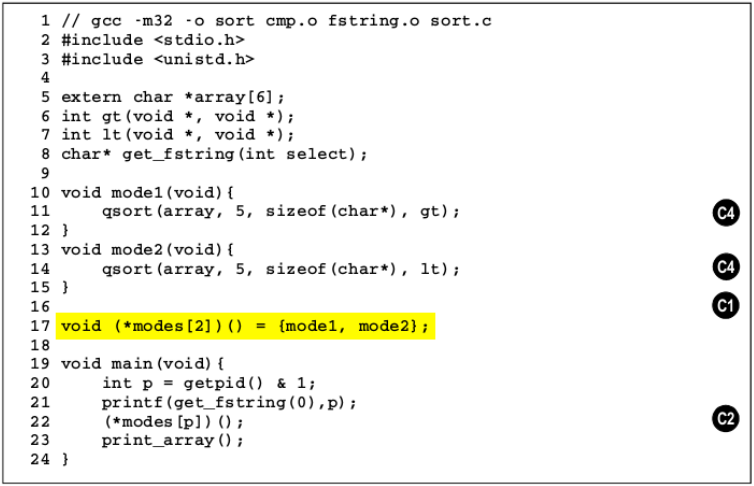
(d) Partial binary code of **sort**

A Running Example

```
zlin@zlin-desktop:~/rewriting-example$ ls -l
total 108
-rw-rw-r-- 1 zlin zlin   436 Apr 29  2016 cmp.c
-rw-rw-r-- 1 zlin zlin  1992 Apr 29  2016 cmp.o
-rw-rw-r-- 1 zlin zlin   219 Apr 29  2016 fstring.asm
-rw-rw-r-- 1 zlin zlin   576 Apr 29  2016 fstring.o
-rwxrwxr-x 1 zlin zlin  7682 Apr 29  2016 sort
-rw-rw-r-- 1 zlin zlin 17172 Apr 29  2016 sort.asm
-rw-rw-r-- 1 zlin zlin   443 Apr 29  2016 sort.c
-rw-rw-r-- 1 zlin zlin  5624 Apr 29  2016 ssort
-rw-rw-r-- 1 zlin zlin 15042 Apr 29  2016 ssort.asm
```

The above code can be downloaded at http://http://web.cse.ohio-state.edu/~lin.3021/file/rewriting-example.zip

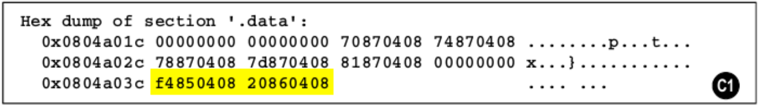
Challenge (C)1: Recognizing and relocating static addresses



```
// gcc -m32 -o sort cmp.o fstring.o sort.c
#include <stdio.h>
#include <unistd.h>

extern char *array[6];
int gt(void *, void *);
int lt(void *, void *);
char* get_fstring(int select);

void mode1(void){
    qsort(array, 5, sizeof(char*), gt);
}
void mode2(void){
    qsort(array, 5, sizeof(char*), lt);
}

void (*modes[2])() = {mode1, mode2};

void main(void){
    int p = getpid() & 1;
    printf(get_fstring(0),p);
    (*modes[p])();
    print_array();
}
```

(a) Source code of `sort.c`

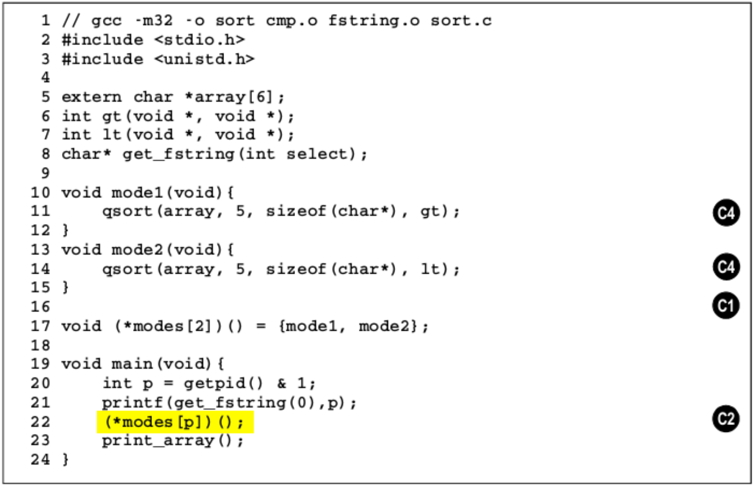
Challenge (C)1: Recognizing and relocating static addresses



```
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

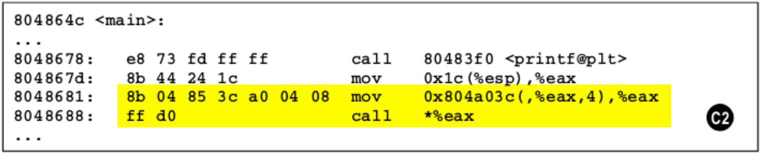
C2: Handling dynamically computed memory addresses



```c
// gcc -m32 -o sort cmp.o fstring.o sort.c
#include <stdio.h>
#include <unistd.h>

extern char *array[6];
int gt(void *, void *);
int lt(void *, void *);
char* get_fstring(int select);

void mode1(void){
    qsort(array, 5, sizeof(char*), gt);
}
void mode2(void){
    qsort(array, 5, sizeof(char*), lt);
}

void (*modes[2])() = {mode1, mode2};

void main(void){
    int p = getpid() & 1;
    printf(get_fstring(0),p);
    (*modes[p])();
    print_array();
}
```

(a) Source code of sort.c

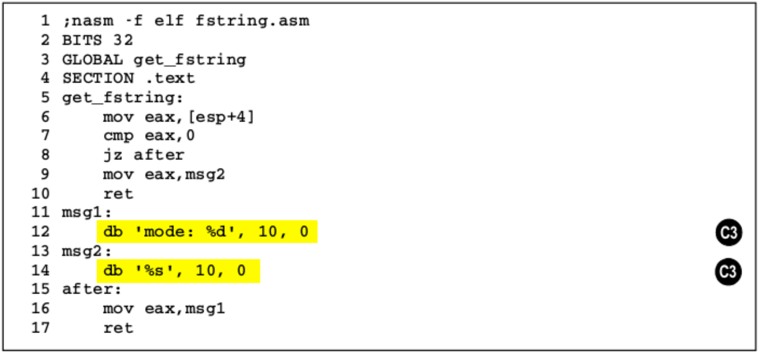
C2: Handling dynamically computed memory addresses



```
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`

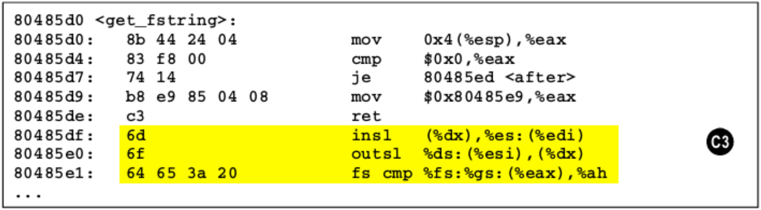
C3: Differentiating code and data



```
;nasm -f elf fstring.asm
BITS 32
GLOBAL get_fstring
SECTION .text
get_fstring:
    mov eax,[esp+4]
    cmp eax,0
    jz after
    mov eax,msg2
    ret
msg1:
    db 'mode: %d', 10, 0
msg2:
    db '%s', 10, 0
after:
    mov eax,msg1
    ret
```

(b) Source code of `fstring.asm`

C3: Differentiating code and data



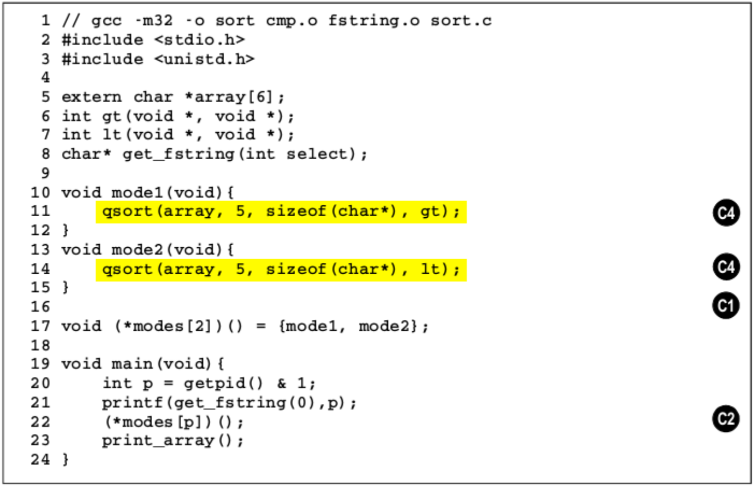
```
80485d0 <get_fstring>:
80485d0:   8b 44 24 04          mov    0x4(%esp),%eax
80485d4:   83 f8 00             cmp    $0x0,%eax
80485d7:   74 14                je     80485ed <after>
80485d9:   b8 e9 85 04 08       mov    $0x80485e9,%eax
80485de:   c3                   ret
80485df:   6d                   insl   (%dx),%es:(%edi)
80485e0:   6f                   outsl  %ds:(%esi),(%dx)
80485e1:   64 65 3a 20          fs cmp %fs:%gs:(%eax),%ah
...
```

(d) Partial binary code of sort

More Real World Examples

```
$ ls *.so*
libavcodec.so.57  libcrypto.so.1.0.0  libffi.so.6
libfreeblpriv3.so
$ ls *.asm
avodec.asm  crypto.asm  ffi.asm  free.asm
$ cat crypto.asm |grep -C3 bad
   f3ae5:       00 00                   add    %al,(%rax)
   f3ae7:       00 ff                   add    %bh,%bh
   f3ae9:       ff                      (bad)
   f3aea:       ff                      (bad)
   f3aeb:       ff                      (bad)
   f3aec:       fb                      sti
   f3aed:       ff                      (bad)
$ cat *.asm |grep bad|wc
   5113   27640  255015
```

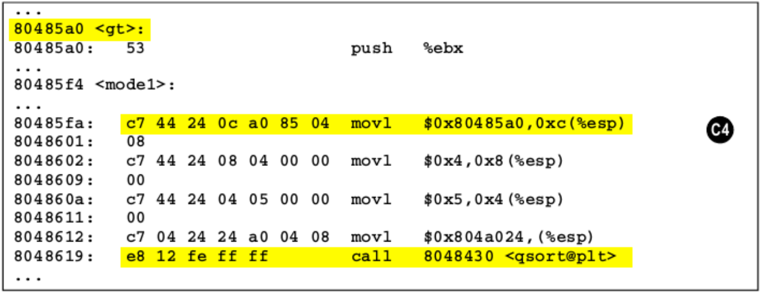
C4: Handling function pointer arguments (e.g., callbacks)



```
// gcc -m32 -o sort cmp.o fstring.o sort.c
#include <stdio.h>
#include <unistd.h>

extern char *array[6];
int gt(void *, void *);
int lt(void *, void *);
char* get_fstring(int select);

void mode1(void){
    qsort(array, 5, sizeof(char*), gt);
}
void mode2(void){
    qsort(array, 5, sizeof(char*), lt);
}

void (*modes[2])() = {mode1, mode2};

void main(void){
    int p = getpid() & 1;
    printf(get_fstring(0),p);
    (*modes[p])();
    print_array();
}
```

(a) Source code of sort.c

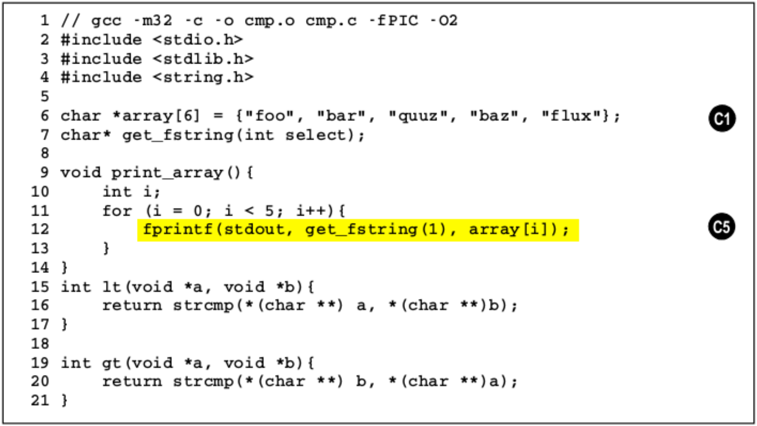
C4: Handling function pointer arguments (e.g., callbacks)



```
...
80485a0 <gt>:
80485a0:    53                      push   %ebx
...
80485f4 <model>:
...
80485fa:    c7 44 24 0c a0 85 04    movl   $0x80485a0,0xc(%esp)
8048601:    08
8048602:    c7 44 24 08 04 00 00    movl   $0x4,0x8(%esp)
8048609:    00
804860a:    c7 44 24 04 05 00 00    movl   $0x5,0x4(%esp)
8048611:    00
8048612:    c7 04 24 24 a0 04 08    movl   $0x804a024,(%esp)
8048619:    e8 12 fe ff ff          call   8048430 <qsort@plt>
...
```

(d) Partial binary code of sort

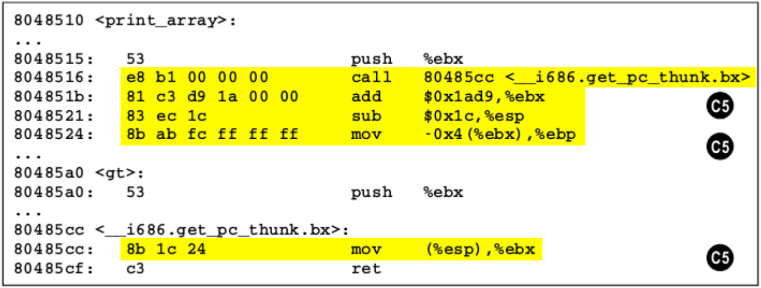
C5: Handling PIC



```
// gcc -m32 -c -o cmp.o cmp.c -fPIC -O2
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *array[6] = {"foo", "bar", "quuz", "baz", "flux"};
char* get_fstring(int select);

void print_array(){
    int i;
    for (i = 0; i < 5; i++){
        fprintf(stdout, get_fstring(1), array[i]);
    }
}
int lt(void *a, void *b){
    return strcmp(*(char **) a, *(char **)b);
}

int gt(void *a, void *b){
    return strcmp(*(char **) b, *(char **)a);
}
```

(c) Source code of **cmp.c**

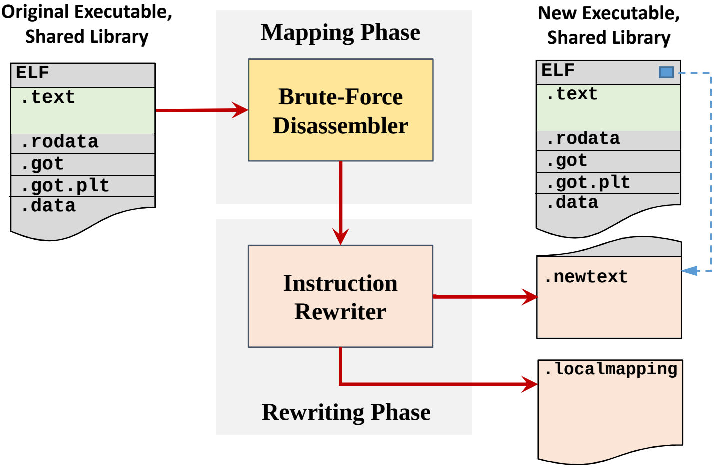
C5: Handling PIC



```
8048510 <print_array>:
...
8048515:    53                    push   %ebx
8048516:    e8 b1 00 00 00        call   80485cc <__i686.get_pc_thunk.bx>
804851b:    81 c3 d9 1a 00 00     add    $0x1ad9,%ebx
8048521:    83 ec 1c              sub    $0x1c,%esp
8048524:    8b ab fc ff ff ff     mov    -0x4(%ebx),%ebp
...
80485a0 <gt>:
80485a0:    53                    push   %ebx
...
80485cc <__i686.get_pc_thunk.bx>:
80485cc:    8b 1c 24              mov    (%esp),%ebx
80485cf:    c3                    ret
```

(d) Partial binary code of sort

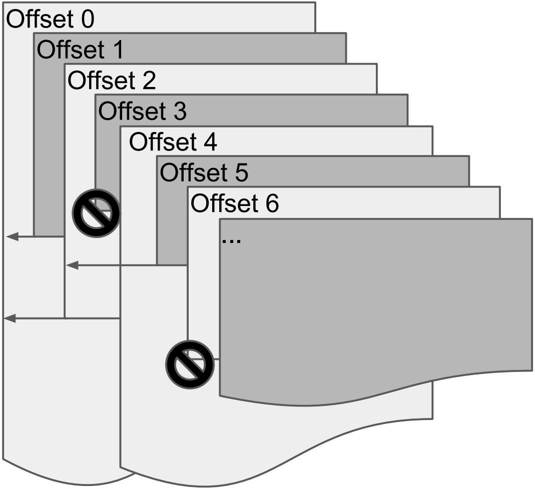
Overview of MULTIVERSE



Superset Disassembler

“When in doubt, use brute force.” – Ken Thompson

Superset Disassembler



Implementations (with Python)

- Disassembler engine: the python bindings for CAPSTONE [cap]
- Parse the ELF data structures: `pyelftools` [pye]
- Reassemble the instructions: `pwntools` [pwn]
- Additional 3,000 lines of our own python code to implement our algorithm and maintain our data structures
- ...

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

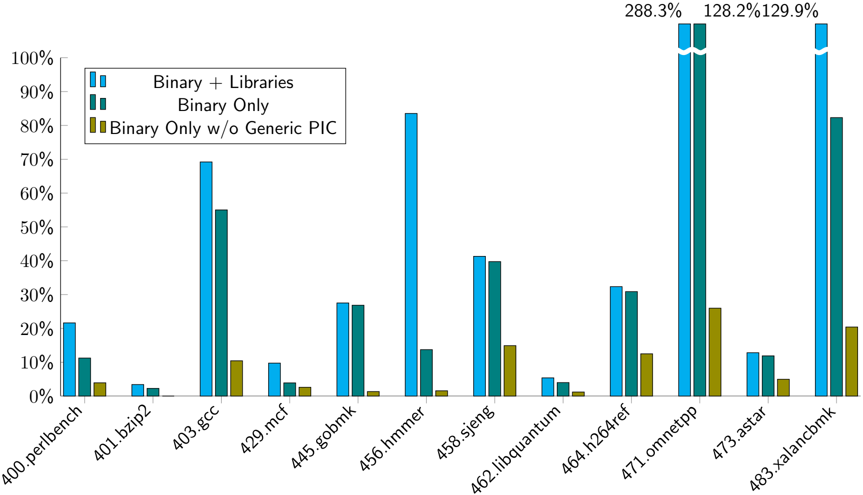
Optimization 2: No Generic PIC

- Assume only PIC is via `get_pc_thunk`
- True for many binaries
- Significant performance increase for compatible binaries

Statistics of our rewritten binaries and libraries

Benchmark	Dir. Calls	Dir. Jumps	Ind. Calls	Ind. Jumps	Cond. Jumps	Rets	.text (KB)	.newtext (KB)	Size Inc. (X)
400.perlbench	30888	24778	3896	4442	126876	22306	1047	5146	12.88
401.bzip2	1100	1050	170	152	7342	874	55	268	70.71
403.gcc	110122	64532	8916	15680	380920	45410	3225	15290	10.32
429.mcf	276	216	44	78	1300	250	12	57	202.98
445.gobmk	23548	14946	3550	3480	117378	20918	1488	6520	5.39
456.hmmer	8020	4942	556	666	28924	4106	277	1279	22.56
458.sjeng	2566	2338	256	658	12236	1570	132	604	36.17
462.libquantum	1094	758	94	146	3376	812	40	181	93.73
464.h264ref	7124	6518	1782	2000	47850	6318	520	2441	16.23
471.omnetpp	33578	10032	3830	1782	51642	14326	635	3029	13.49
473.astar	912	552	162	160	3314	750	39	184	92.52
483.xalancbmk	115154	58678	39392	14630	307122	75674	3850	17369	7.60
libc.so.6	32798	33370	9816	9012	189384	32458	1735	8435	9.77
libgcc_s.so.1	2158	2514	374	484	12862	1740	112	538	9.70
libm.so.6	5450	8870	874	892	21796	7406	277	1268	9.51
libstdc++.so.6	22456	10418	4300	4008	144516	15784	900	4258	9.53

Multiverse Overhead: No Instrumentaiton



Instrumentation Evaluation

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

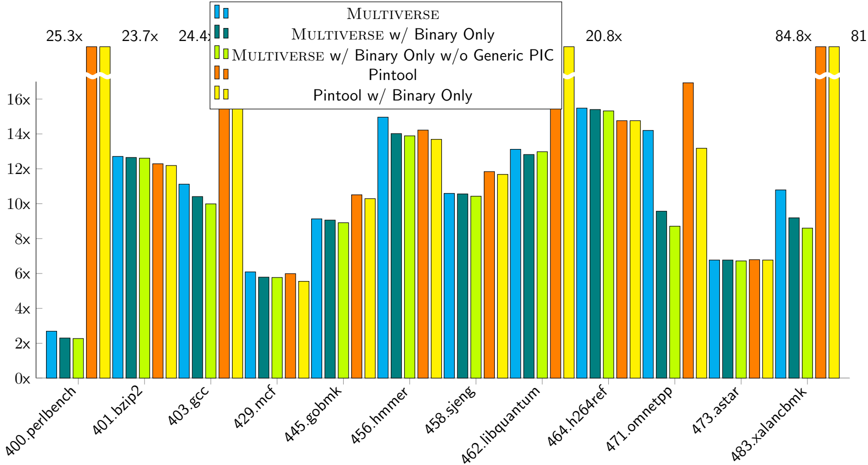
- Ultimate purpose of a rewriter is to insert instrumentation code
- Created straightforward instrumentation API
- For evaluation created instruction counting instrumentation in MULTIVERSE

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



Related Work

Systems	Year	w/o Relocation	w/o (Debugging) Symbols	w/o Heuristics for Static Address	w/o Heuristics for PIC	w/o Heuristics for Callbacks	w/o Heuristics for Disassemble	Instrumentation	Profiling	Optimization	Binary Code Hardening	Control Flow Integrity	Binary Code Reuse
Etch [RVL+97]	1997	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗
SASI [ES99]	1999	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
Plto [SDAL01]	2001	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
Vulcan [SEV01]	2001	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
Diablo [VPCDB+05]	2005	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
CFI [ABEL09]	2005	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗
XFI [EVA+06]	2006	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PittSFIeld [MM06]	2006	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
Bird [NLLC06]	2006	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	✗
NaCl [YSD+09]	2009	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
Pebil [LTCS10]	2010	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
SecondWrite [OAK+11]	2011	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗
DynInst [BM11]	2011	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗
Stir/Reins [WMHL12a, WMHL12b]	2012	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✓	✗
Ccfir [ZWC+13]	2013	✗	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗
Bistro [DZX13]	2013	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
BinCFI [ZS13]	2013	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗
Psi [ZQHS14]	2014	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗
Uroboros [WWW16]	2016	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
Ramblr [WSB+17]	2017	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
Multiverse [BLH18]	2018	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

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

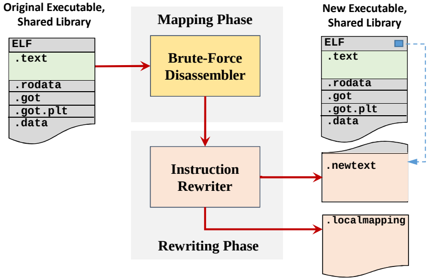
Optimization

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

Instrumentation API

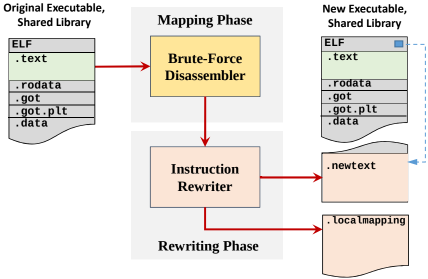
- For paper, used simple instruction-level API
- Currently working on more robust API

Conclusion



- Multiverse: **Statically rewriting** x86 binaries w/o heuristics
- Works for x86/64 binaries
- Useful for many security applications (e.g., hardening)

Thank You



Q&A

zlin@cse.ohio-state.edu

MULTIVERSE Source Code

github.com/utds3lab/multiverse

References I

- Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti, *Control-flow integrity principles, implementations, and applications*, ACM Trans. Information and System Security **13** (2009), no. 1.
- Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua, *A compiler-level intermediate representation based binary analysis and rewriting system*, Proceedings of the 8th ACM European Conference on Computer Systems, ACM, 2013, pp. 295–308.
- Erick Bauman, Zhiqiang Lin, and Kevin Hamlen, *Superset disassembly: Statically rewriting x86 binaries without heuristics*, Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18) (San Diego, CA), February 2018.
- Andrew R Bernat and Barton P Miller, *Anywhere, any-time binary instrumentation*, Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, ACM, 2011, pp. 9–16.
- *Capstone: The ultimate disassembler*, http://www.capstone-engine.org/.
- Xi Chen, Herbert Bos, and Cristiano Giuffrida, *CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks*, EuroS&P, April 2017.
- Brian Cox and Jeffrey Robert. Forshaw, *The quantum universe: everything that can happen does happen*, Penguin, 2012.
- Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song, *Binary code extraction and interface identification for security applications*, NDSS, Feb. 2010.
- Zhui Deng, Xiangyu Zhang, and Dongyan Xu, *Bistro: Binary component extraction and embedding for software security applications*, Computer Security–ESORICS 2013, Springer, 2013, pp. 200–218.

References II

- Úlfar Erlingsson and Fred B. Schneider, *SASI enforcement of security policies: A retrospective*, Proc. New Security Paradigms Workshop, 1999.
- Úlfar Erlingsson, Silicon Valley, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula, *Xfi: software guards for system address spaces*, Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06) (Seattle, WA), USENIX Association, 2006, pp. 6–6.
- Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna, *Static disassembly of obfuscated binaries*, USENIX security Symposium, vol. 13, 2004, pp. 18–18.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, *Pin: Building customized program analysis tools with dynamic instrumentation*, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05) (Chicago, IL, USA), 2005, pp. 190–200.
- Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely, *Pebil: Efficient static binary instrumentation for linux*, Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, IEEE, 2010, pp. 175–183.
- Evangelos Ladakis, Giorgos Vasiliadis, Michalis Polychronakis, Sotiris Ioannidis, and Georgios Portokalidis, *Gpu-disasm: A gpu-based x86 disassembler*, International Information Security Conference, Springer, 2015, pp. 472–489.
- Stephen McCamant and Greg Morrisett, *Evaluating SFI for a CISC architecture*, Proc. USENIX Security Sym., 2006.

References III

- Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh, *Bird: Binary interpretation using runtime disassembly*, Proceedings of the International Symposium on Code Generation and Optimization (Washington, DC, USA), CGO '06, IEEE Computer Society, 2006, pp. 358–370.
- Pádraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D. Keromytis, *Retrofitting security in COTS software with binary rewriting*, Proc. Int. Information Security Conf., 2011, pp. 154–172.
- *Pwntools*, `https://github.com/Gallopsled/pwntools`.
- *Pyelftools*, `https://github.com/eliben/pyelftools`.
- Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen, *Instrumentation and optimization of win32/intel executables using etch*, Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997 (Berkeley, CA, USA), NT'97, USENIX Association, 1997, pp. 1–1.
- Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre, *Plto: A link-time optimizer for the intel ia-32 architecture*, Proc. 2001 Workshop on Binary Translation (WBT-2001), 2001.
- Amitabh Srivastava, Andrew Edwards, and Hoi Vo, *Vulcan: Binary transformation in a distributed environment*, Tech. report, technical report msr-tr-2001-50, microsoft research, 2001.
- Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere, *Diablo: a reliable, retargetable and extensible link-time rewriting framework*, Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on, IEEE, 2005, pp. 7–12.

References IV

- Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham, *Efficient software-based fault isolation*, Proc. ACM Sym. Operating Systems Principles, 1993, pp. 203–216.
- Richard Wartell, Vishwath Mohan, Kevin Hamlen, and Zhiqiang Lin, *Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code*, Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12) (Raleigh, NC), October 2012.
- ______, *Securing untrusted code via compiler-agnostic binary rewriting*, Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12) (Orlando, FL), December 2012.
- Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna, *Ramblr: Making reassembly great again*.
- Shuai Wang, Pei Wang, and Dinghao Wu, *Uroboros: Instrumenting stripped binaries with static reassembling*, Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, vol. 1, IEEE, 2016, pp. 236–247.
- Richard Wartell, Yan Zhou, Kevin W Hamlen, and Murat Kantarcioglu, *Shingled graph disassembly: Finding the undecideable path*, Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2014, pp. 273–285.
- Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar, *Native Client: A sandbox for portable, untrusted x86 native code*, Proc. IEEE Sym. Security and Privacy, 2009, pp. 79–93.

References V

- Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar, *A platform for secure static binary instrumentation*, Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (New York, NY, USA), VEE '14, ACM, 2014, pp. 129–140.
- Mingwei Zhang and R. Sekar, *Control flow integrity for cots binaries*, Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13) (Washington, D.C.), USENIX, 2013, pp. 337–352.
- Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dong Song, and Wei Zou, *Practical control flow integrity and randomization for binary executables*, Security and Privacy (SP), 2013 IEEE Symposium on, IEEE, 2013, pp. 559–573.