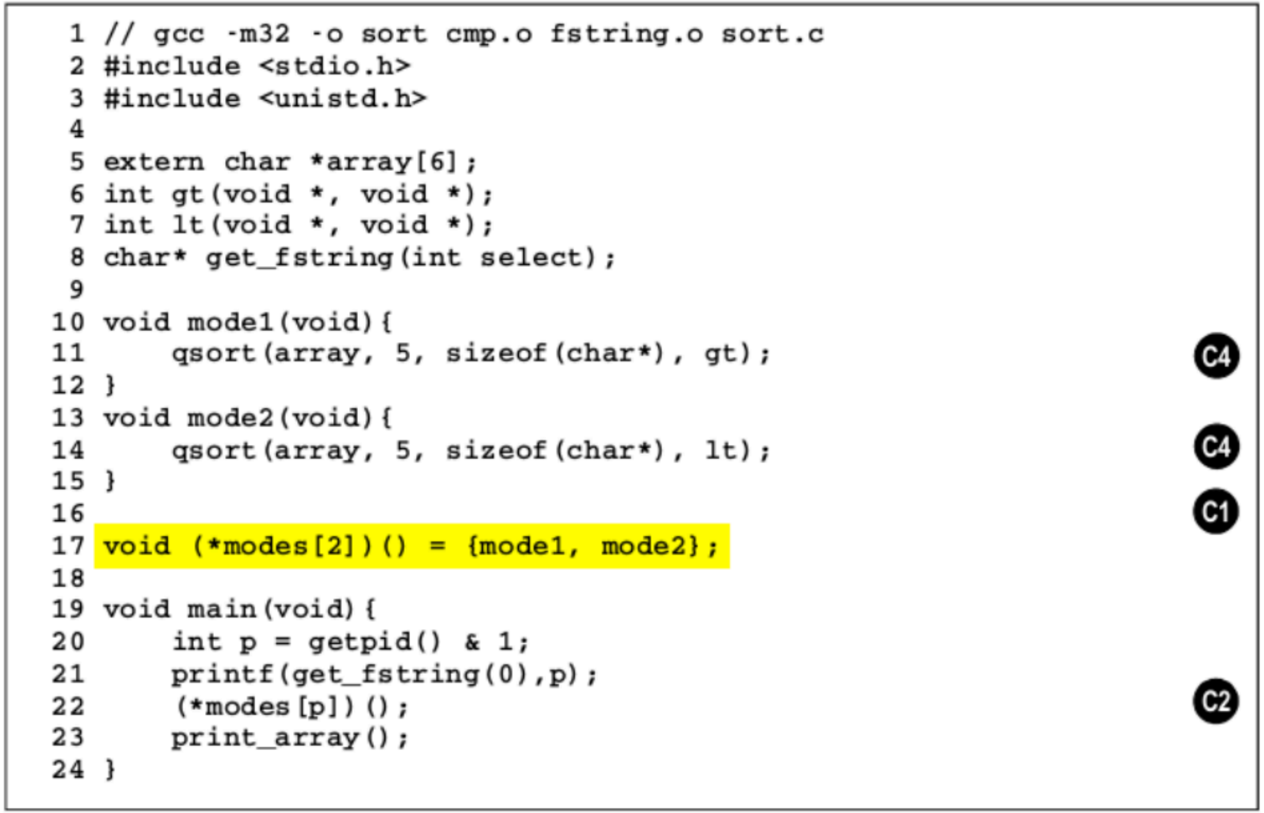
Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics

Erick Bauman, Zhiqiang Lin, Kevin W. Hamlen

Motivation

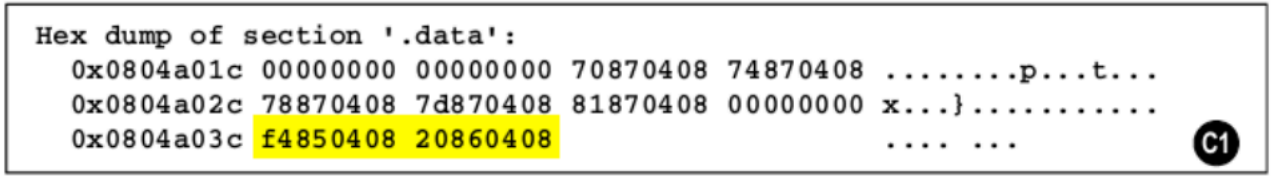
- Static binary rewriting is a core technology for many systems and security applications
- Many static rewriters have been developed over the past decades
- These tools rely on various assumptions and heuristics
- MULTIVERSE: the first heuristic-free static binary rewriter

Challenge 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();
}
```

Challenge 1:Recognizing and relocating static addresses



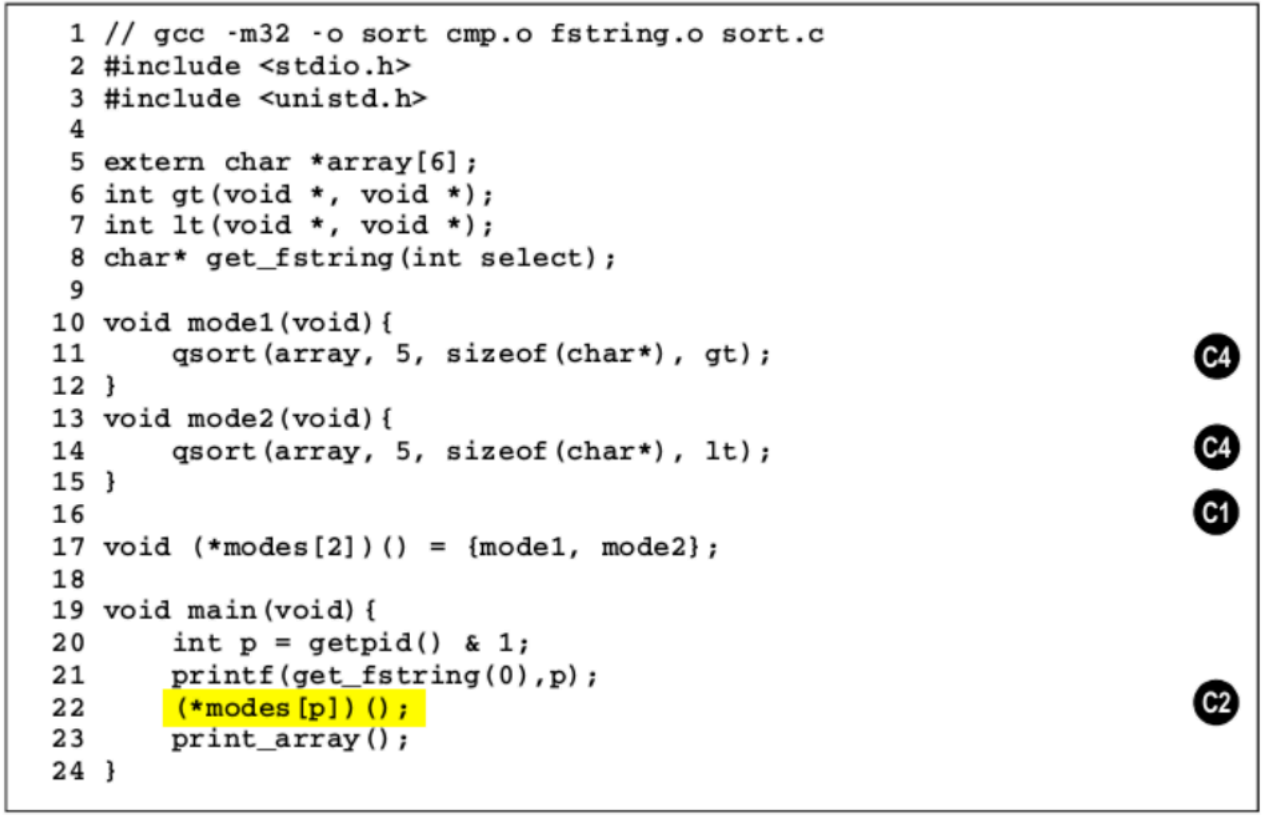
```
Hex dump of section '.data':
  0x0804a01c 00000000 00000000 70870408 74870408 ........p...t...
  0x0804a02c 78870408 7d870408 81870408 00000000 x...}...........
  0x0804a03c f4850408 20860408                   .... ...
```

(f) Hexdump of `.data` section

Keeping original data space intact

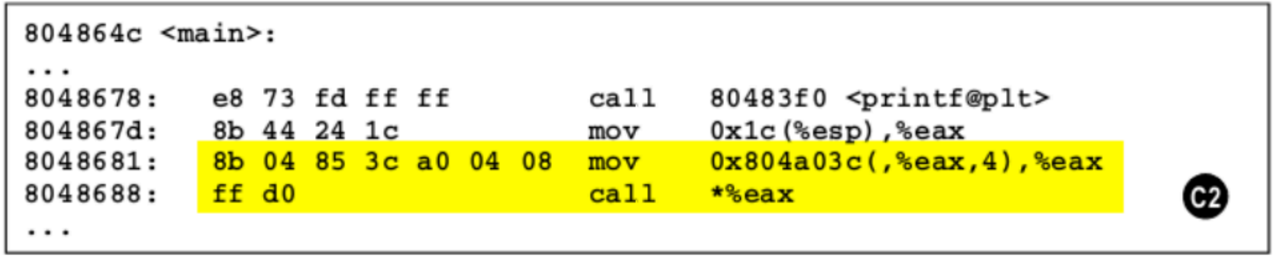
- No need to modify data addresses if data unchanged
- Keep read-only copy of code for inline data in original code section

Challenge 2: Handling dynamically computed memory 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();
}
```

Challenge 2: Handling dynamically computed memory addresses



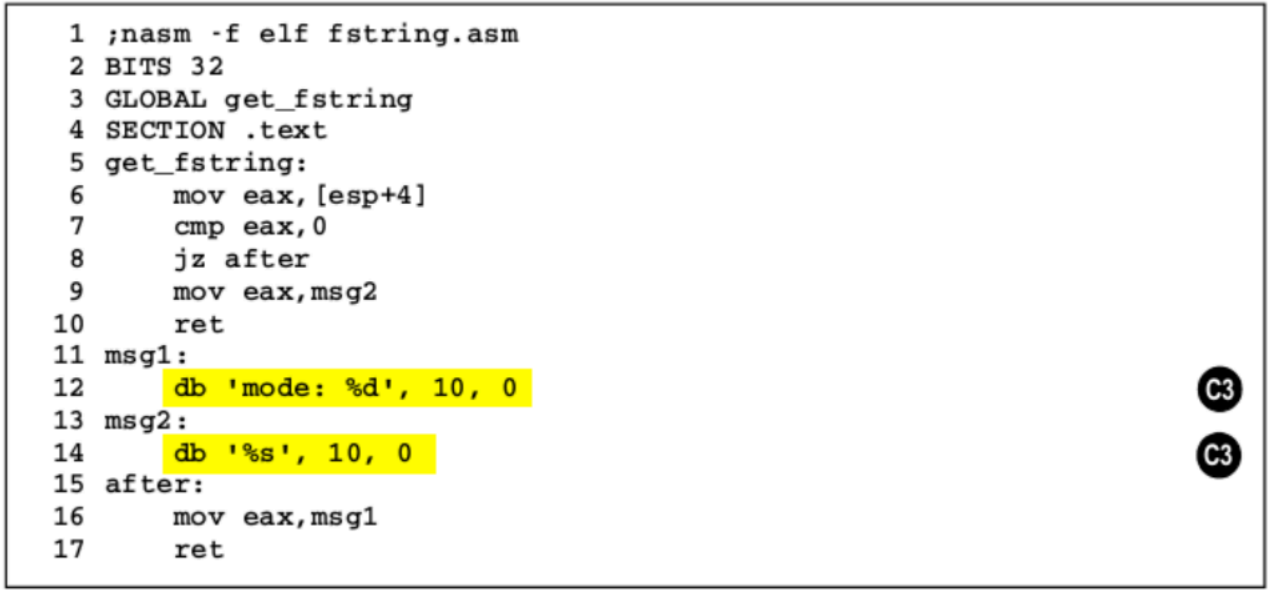
```
804864c <main>:
...
8048678:    e8 73 fd ff ff          call    80483f0 <printf@plt>
804867d:    8b 44 24 1c             mov     0x1c(%esp),%eax
8048681:    8b 04 85 3c a0 04 08    mov     0x804a03c(,%eax,4),%eax
8048688:    ff d0                   call    *%eax
...
```

(d) Partial binary code of `sort`

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

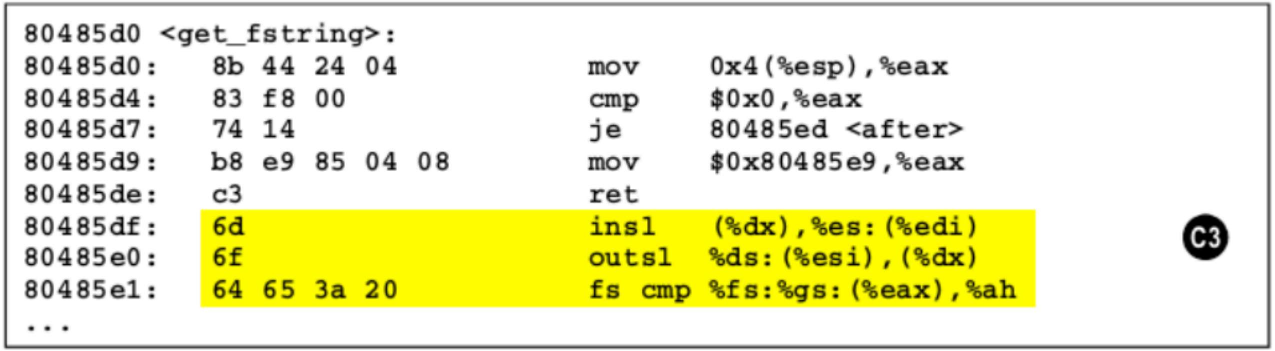
Challenge 3: Differentiating code and data



```
;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

Challenge 3: Differentiating code and data



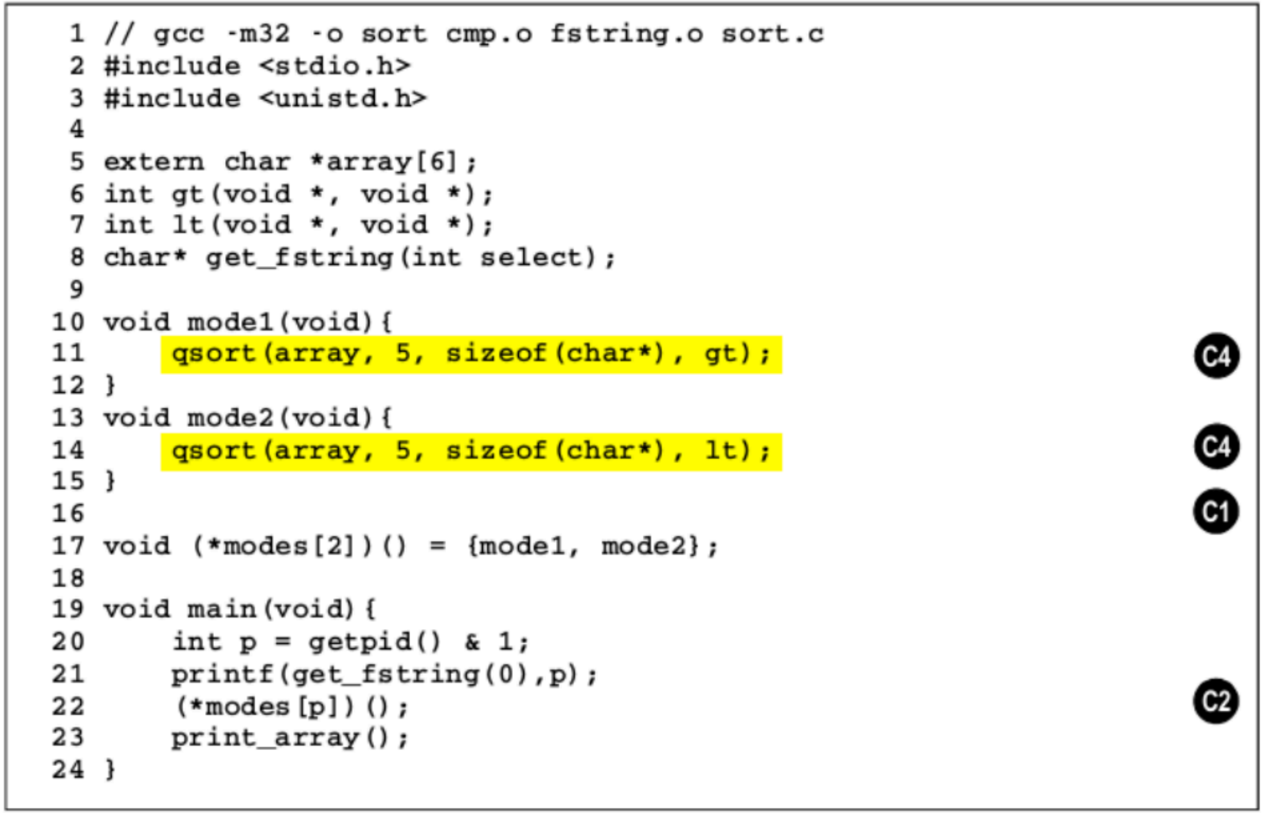
```
80485d0 <get_fstring>:
80485d0:    8b 44 24 04            mov     0x4(%esp),%eax
80485d4:    83 f8 00               cmp     $0x0,%eax
80485d7:    74 14                  je      80485ed <after>
80485d9:    b8 e9 85 04 08         mov     $0x80485e9,%eax
80485de:    c3                     ret
80485df:    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

Brute force disassembling of all possible code

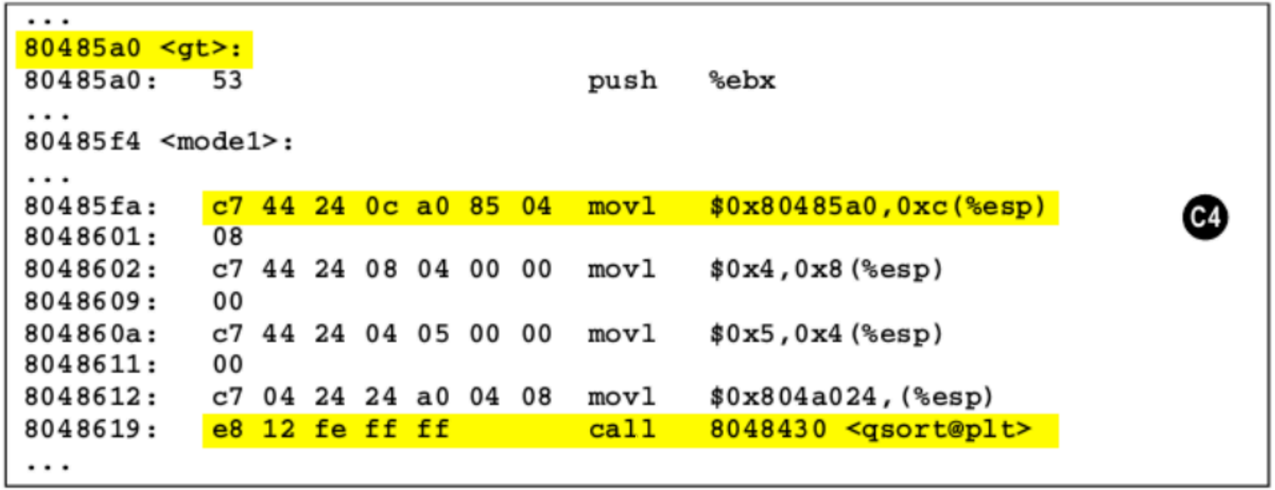
- Disassemble every offset
- All intended code will be within resulting superset

Challenge 4: Handling function pointer arguments



```
// 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();
}
```

Challenge 4: Handling function pointer arguments



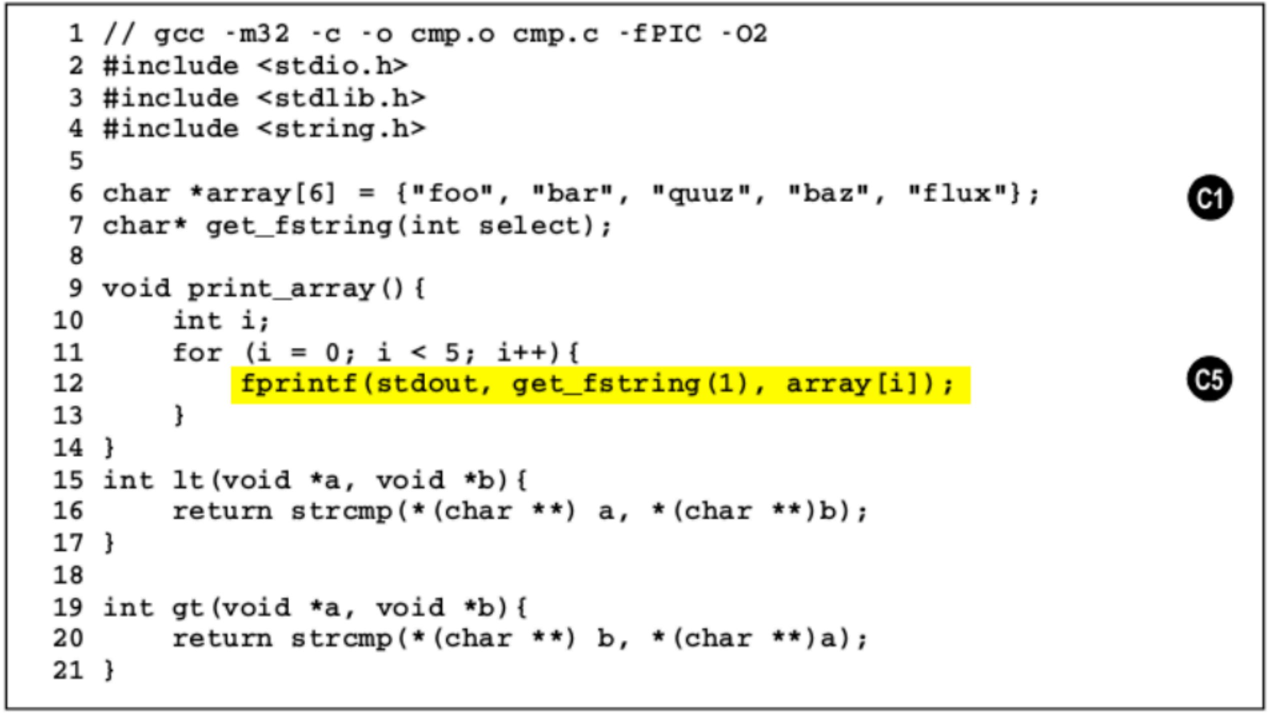
```
...
80485a0 <gt>:
80485a0:    53                      push   %ebx
...
80485f4 <mode1>:
...
80485fa:    c7 44 24 0c a0 85 04    movl   $0x80485a0,0xc(%esp)
8048601:    08
8048602:    c7 44 24 08 04 00 00    movl   $0x4,0x8(%esp)
8048609:    00
804860a:    c7 44 24 04 05 00 00    movl   $0x5,0x4(%esp)
8048611:    00
8048612:    c7 04 24 24 a0 04 08    movl   $0x804a024,(%esp)
8048619:    e8 12 fe ff ff          call   8048430 <qsort@plt>
...
```

(d) Partial binary code of sort

Rewriting all user level code including libraries

- Hard to automatically identify all function pointer arguments
- Instead, rewrite everything
- Use mapping(from solution 2) to translate callback upon use

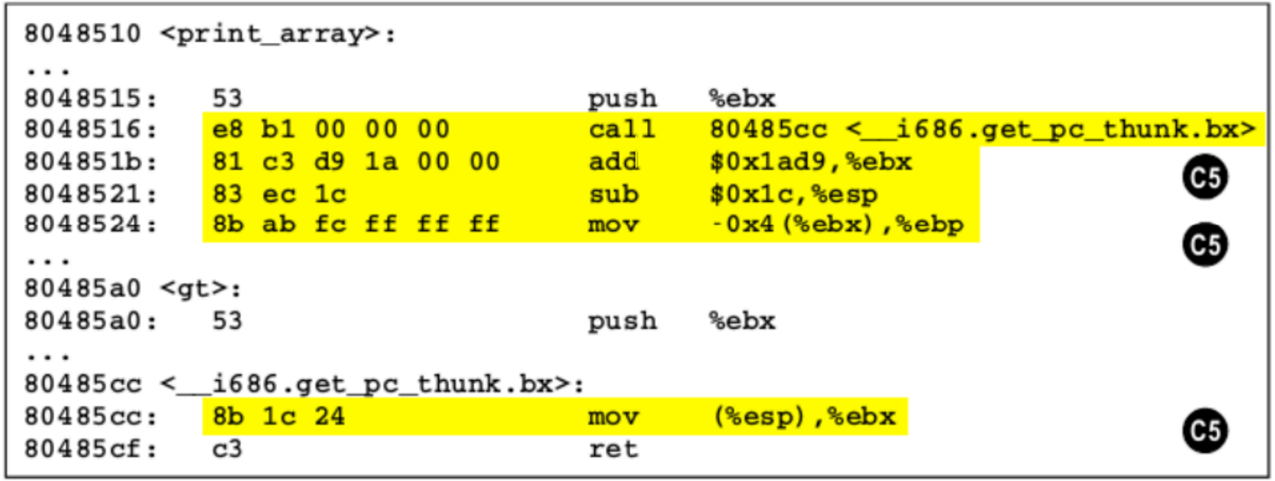
Challenge 5: Handling PIC



```
// 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

Challenge 5: Handling PIC



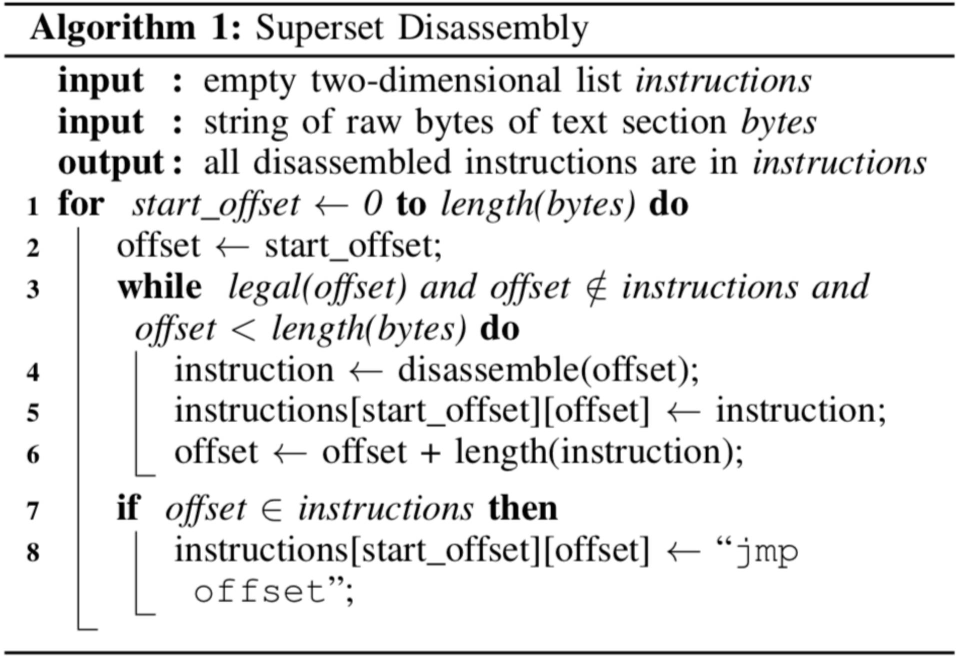
```
8048510 <print_array>:
...
8048515:   53                      push   %ebx
8048516:   e8 b1 00 00 00          call   80485cc <__i686.get_pc_thunk.bx>
804851b:   81 c3 d9 1a 00 00       add    $0x1ad9,%ebx
8048521:   83 ec 1c                sub    $0x1c,%esp
8048524:   8b ab fc ff ff ff       mov 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

Rewriting all call instruction

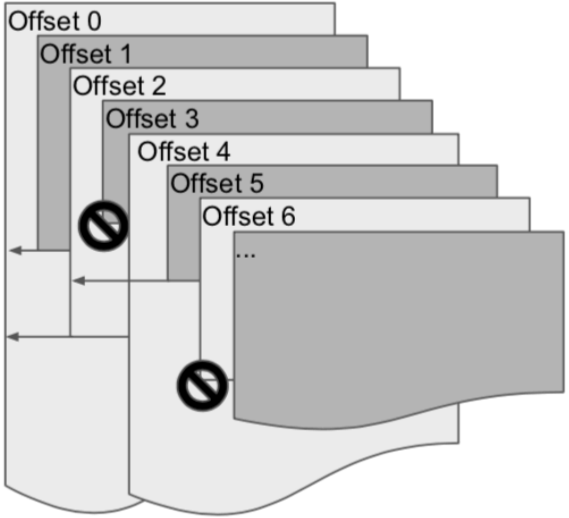
- For x86-32 instructions, only call reveals instruction pointer
- Rewrite call to push/jmp and push old return address
- Offsets computed based on old address
- From solution 2, rewritten ret instructions translate return address with mapping

Superset Disassembler



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 $instructions$
for $start\_offset \leftarrow 0$ to $length(bytes)$ do
    offset ← start_offset;
    while $legal(offset)$ and $offset \notin instructions$ and
     $offset < length(bytes)$ do
        instruction ← disassemble(offset);
        instructions[start_offset][offset] ← instruction;
        offset ← offset + length(instruction);
    if $offset \in instructions$ then
        instructions[start_offset][offset] ← "jmp
         offset";
```



: An illustration of our disassembly strategy.

Mapping

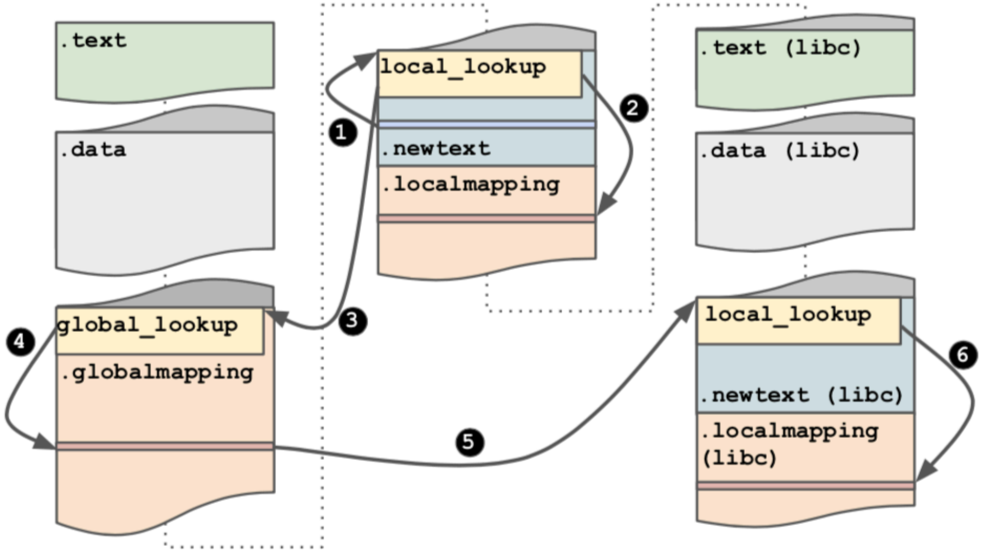
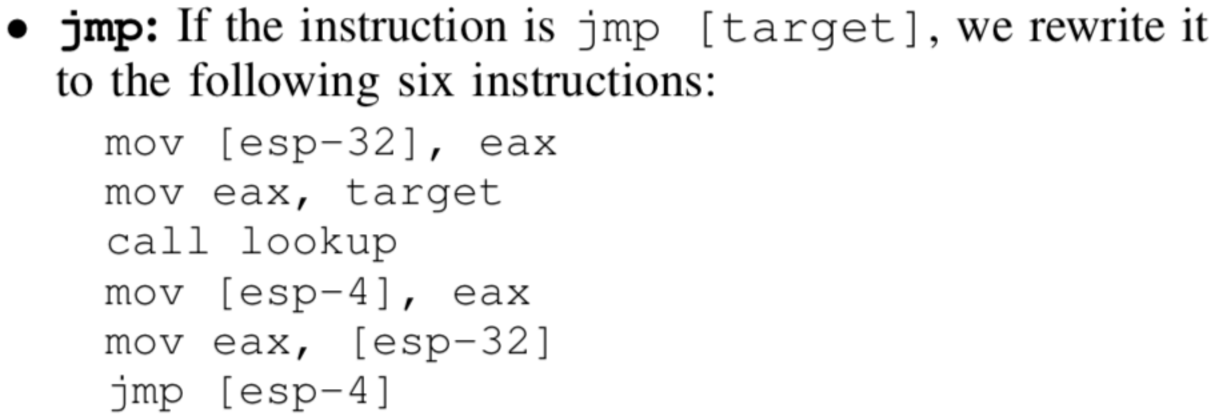


Fig. 4: A mapping lookup example for a rewritten binary dynamically linked with our rewritten `libc`.

Rewriting

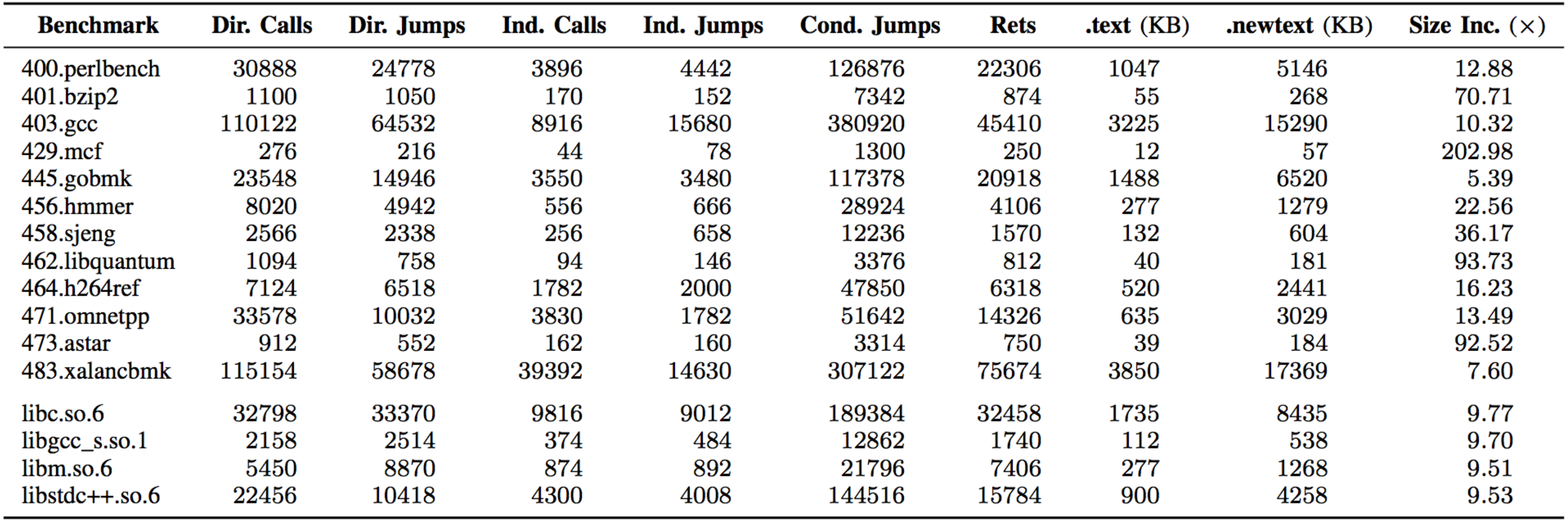
- jcc/jmp/call
- Indirect call



- **jmp:** If the instruction is `jmp [target]`, we rewrite it to the following six instructions:

```
mov [esp-32], eax
mov eax, target
call lookup
mov [esp-4], eax
mov eax, [esp-32]
jmp [esp-4]
```

Evaluation



Benchmark	Dir. Calls	Dir. Jumps	Ind. Calls	Ind. Jumps	Cond. Jumps	Rets	.text (KB)	.newtext (KB)	Size Inc. (×)
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

TABLE I: Statistics of MULTIVERSE rewritten binaries and libraries