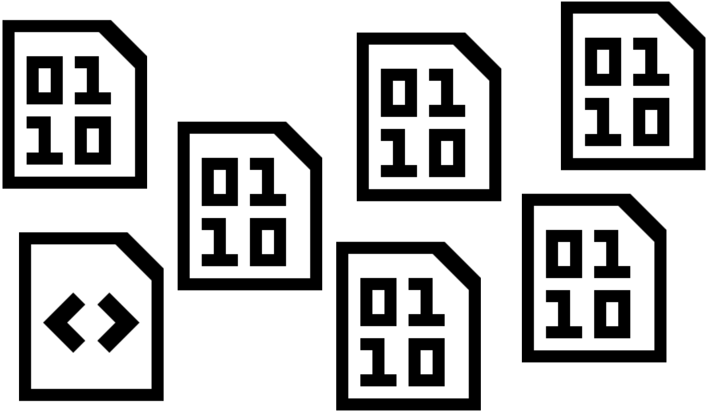
Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics

Erick Bauman[1], Zhiqiang Lin[1,2], Kevin Hamlen[1]

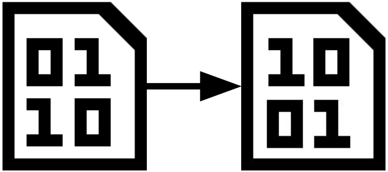
[1]University of Texas at Dallas
[2]The Ohio State University

NDSS 2018

Static Binary Rewriting



Static Binary Rewriting



Many Static Rewriters Have Been Developed Over the Past Decades

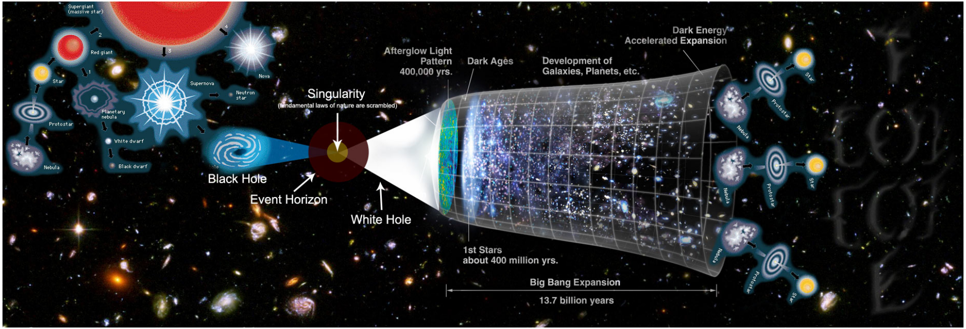
Systems	Year	RL	DS	HS	HP	HC	HD	IN	PR	OP	HA	CF	RU
ETCH [RVL+97]	1997	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗
SASI [ES99]	1999	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PLTO [SDAL01]	2001	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
VULCAN [SEV01]	2001	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
DIABLO [PCB+05]	2005	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
CFI [ABEL09]	2005	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗
XFI [EAV+06]	2006	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PITTSFIELD [MM06]	2006	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
BIRD [NLLC06]	2006	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	✗
NACL [YSD+09]	2009	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PEBIL [LTCS10]	2010	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
SECONDWRITE [OAK+11]	2011	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗
DYNINST [BM11]	2011	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗
STIR/REINS [WMHL12b, WMHL12a]	2012	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✓	✗
CCFIR [ZWC+13]	2013	✗	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗
BISTRO [DZX13]	2013	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
BINCFI [ZS13]	2013	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗
PSI [ZQHS14]	2014	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗
UROBOROS [WWW16]	2016	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
RAMBLR [WSB+17]	2017	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓

Many Static Rewriters Have Been Developed Over the Past Decades

Systems	Year	RL	DS	HS	HP	HC	HD	IN	PR	OP	HA	CF	RU
ETCH [RVL+97]	1997	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗
SASI [ES99]	1999	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PLTO [SDAL01]	2001	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
VULCAN [SEV01]	2001	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
DIABLO [PCB+05]	2005	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
CFI [ABEL09]	2005	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗
XFI [EAV+06]	2006	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PITTSFIELD [MM06]	2006	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
BIRD [NLLC06]	2006	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	✗
NACL [YSD+09]	2009	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PEBIL [LTCS10]	2010	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
SECONDWRITE [OAK+11]	2011	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗
DYNINST [BM11]	2011	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗
STIR/REINS [WMHL12b, WMHL12a]	2012	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✓	✗
CCFIR [ZWC+13]	2013	✗	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗
BISTRO [DZX13]	2013	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
BINCFI [ZS13]	2013	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗
PSI [ZQHS14]	2014	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗
UROBOROS [WWW16]	2016	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
RAMBLR [WSB+17]	2017	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓

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

```
// 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`

Working 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`

Working 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"};                  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`

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);                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`

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                   .... ...
```

C1

(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+11, ZS13, WMHL12b, WMHL12a]

C2: 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);                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`

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
...
```

C2

(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

```
;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`

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
...
```

C3

(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 <u>every</u> offset [KRVV04, WZHK14, LVP+15]
- All intended code will be within resulting superset

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);                    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

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)        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>
...
```

(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 ❷) to translate callback upon use

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"};     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`

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              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
```

(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

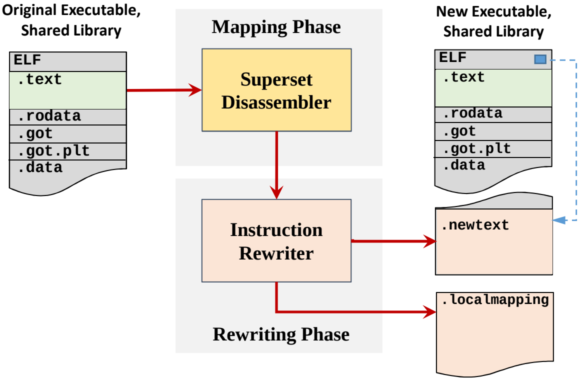
C5: Handling PIC

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

Rewriting all `call` instructions

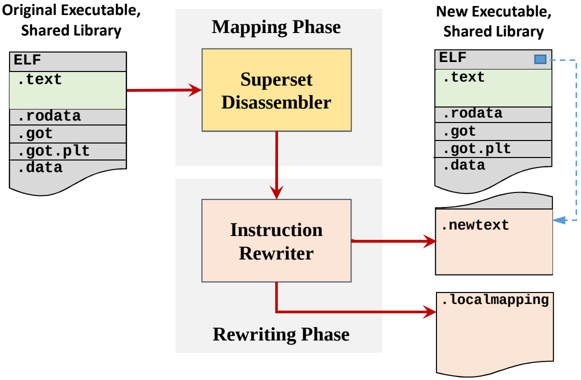
- For x86-32 instructions, only `call` reveals instruction pointer
- Rewrite `call` to `push`/`jmp` and push old return address [ZS13, CBG17]
- Offsets computed based on old address
- From Solution ❷, rewritten `ret` instructions translate return address with mapping

MULTIVERSE



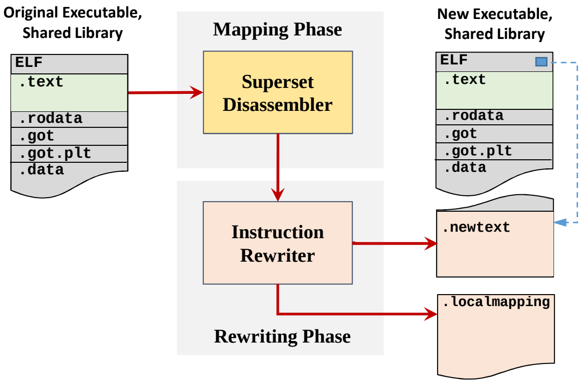
1. Mapping Phase

MULTIVERSE



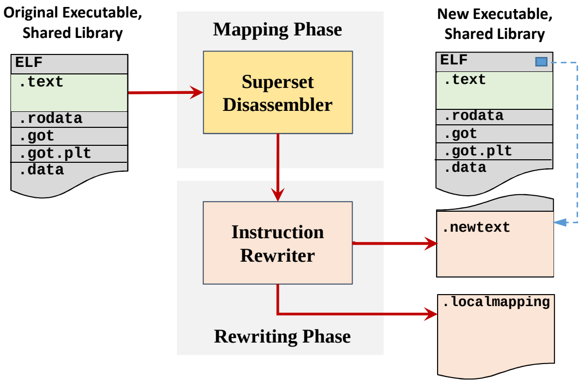
1. Mapping Phase
   - Disassemble starting from every byte

MULTIVERSE



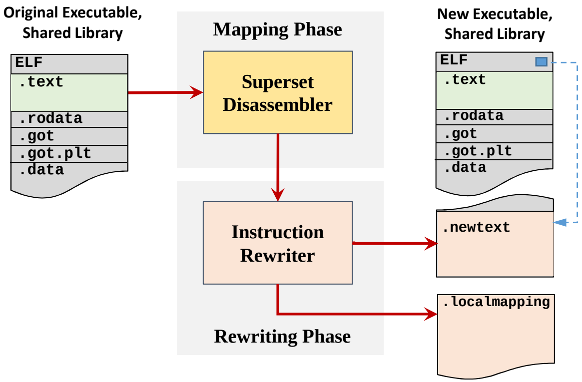
1. Mapping Phase
   - Disassemble starting from every byte
   - Determine lengths of rewritten instructions

MULTIVERSE



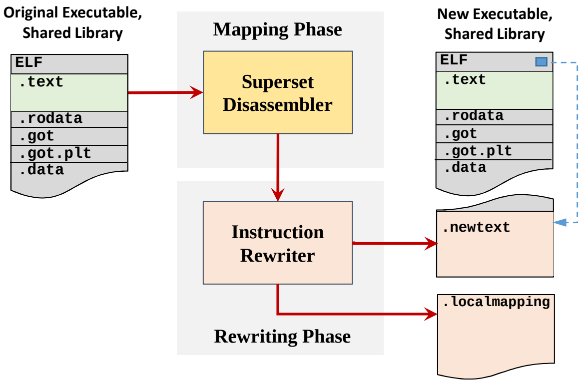
1. Mapping Phase
   - Disassemble starting from every byte
   - Determine lengths of rewritten instructions
   - Create mapping from original address to rewritten address

MULTIVERSE



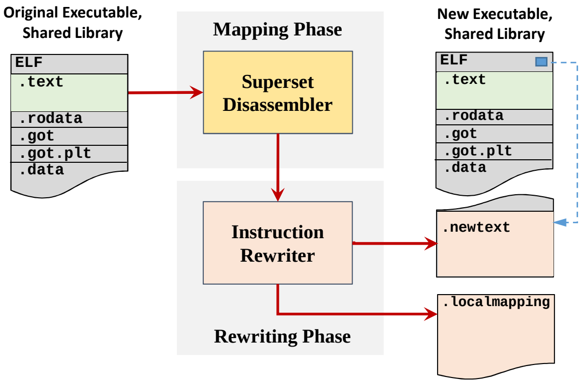
1. Mapping Phase
   - Disassemble starting from every byte
   - Determine lengths of rewritten instructions
   - Create mapping from original address to rewritten address
2. Rewriting Phase

MULTIVERSE



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



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*

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

The Algorithm

1. Start disassembly at first byte

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
for start_offset ← 0 to length(bytes) do
    offset ← start_offset;
    while legal(offset) and offset ∉ instructions and
      offset < length(bytes) do
        instruction ← disassemble(offset);
        instructions[start_offset][offset] ← instruction;
        offset ← offset + length(instruction);
    if offset ∈ instructions then
        instructions[start_offset][offset] ← "jmp
         offset";
```

The Algorithm

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

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*

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

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

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 ← 0 to length(bytes) do
    offset ← start_offset;
    while legal(offset) and offset ∉ instructions and
      offset < length(bytes) do
        instruction ← disassemble(offset);
        instructions[start_offset][offset] ← instruction;
        offset ← offset + length(instruction);
    if offset ∈ instructions then
        instructions[start_offset][offset] ← "jmp
         offset";
```

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

Superset Disassembly

Algorithm 1: Superset Disassembly

---

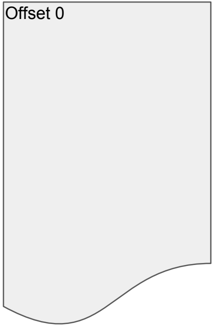
input : empty two-dimensional list *instructions*
input : string of raw bytes of text section *bytes*
output : all disassembled instructions are in *instructions*

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`";

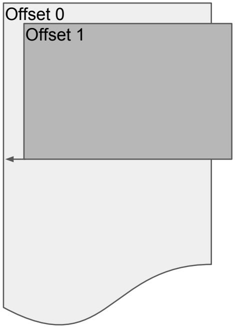
The Algorithm

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 ❷

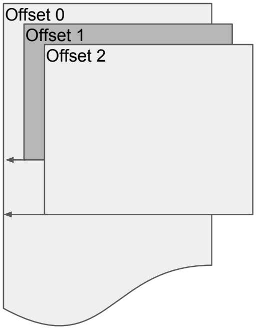
Superset Disassembly



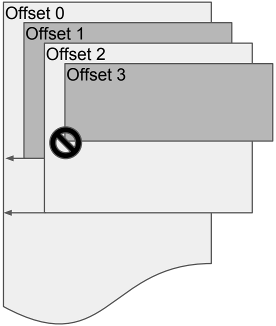
Superset Disassembly



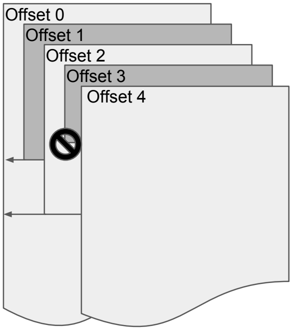
Superset Disassembly



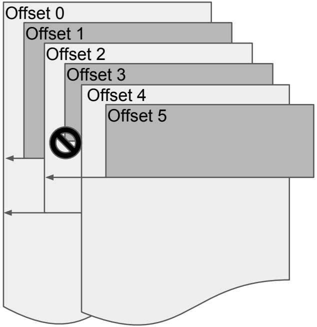
Superset Disassembly



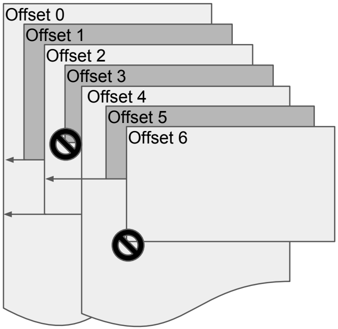
Superset Disassembly



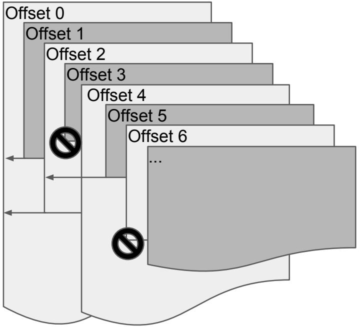
Superset Disassembly



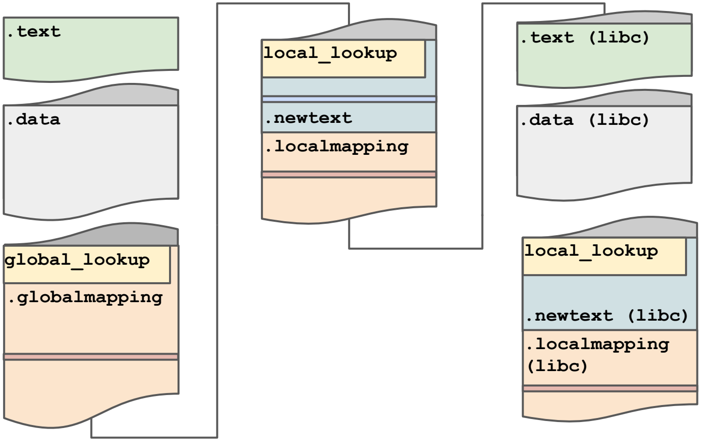
Superset Disassembly



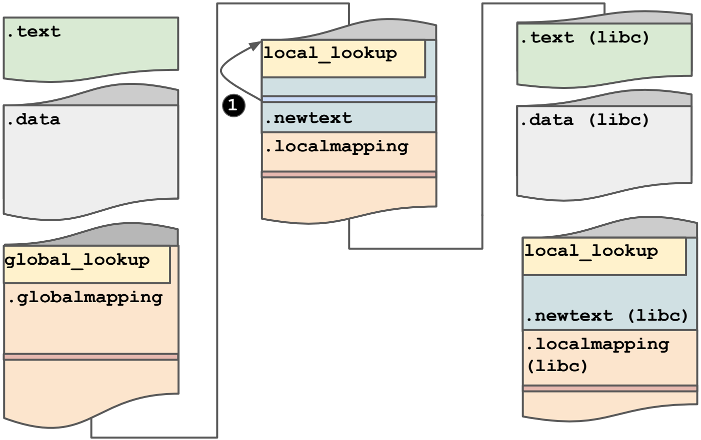
Superset Disassembly



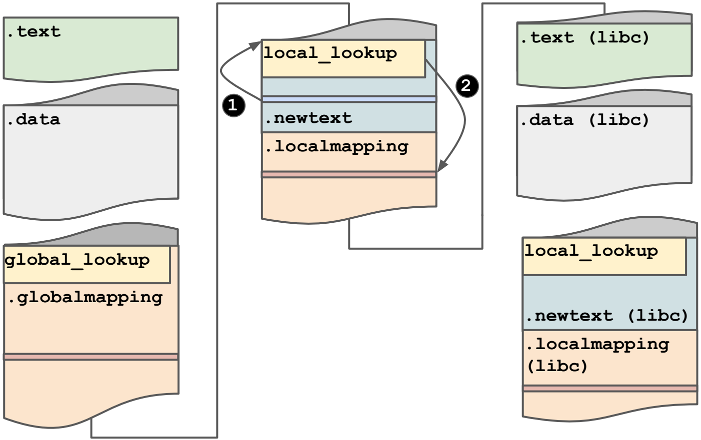
Mapping Lookups



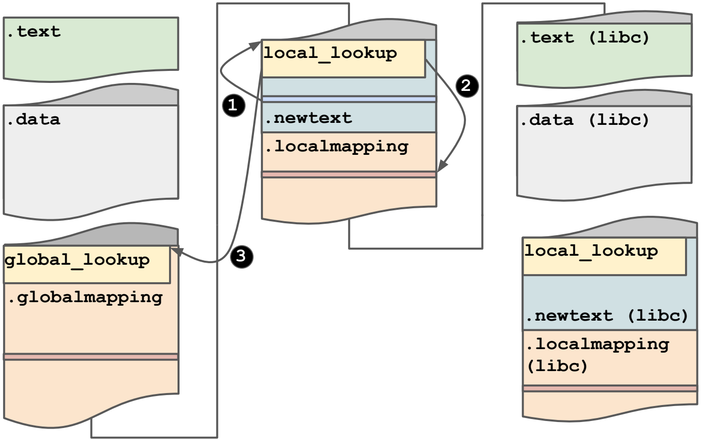
Mapping Lookups



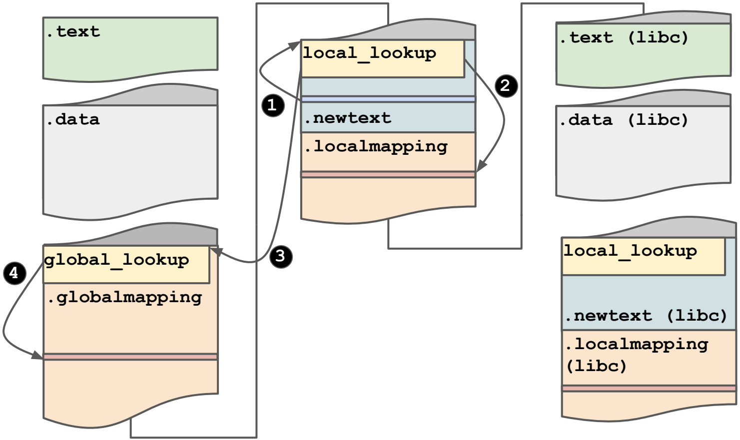
Mapping Lookups



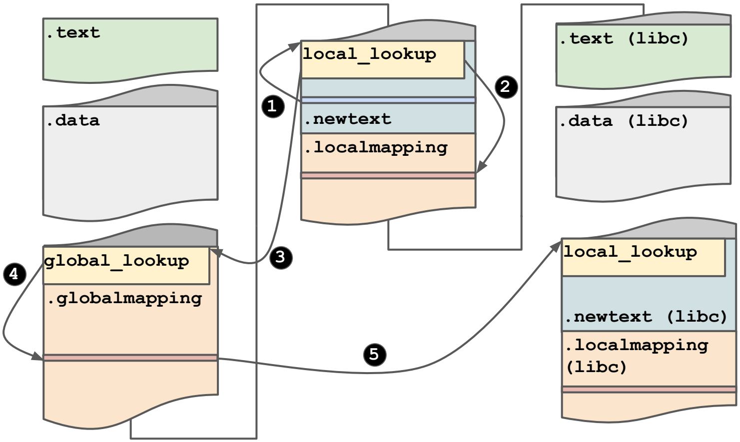
Mapping Lookups



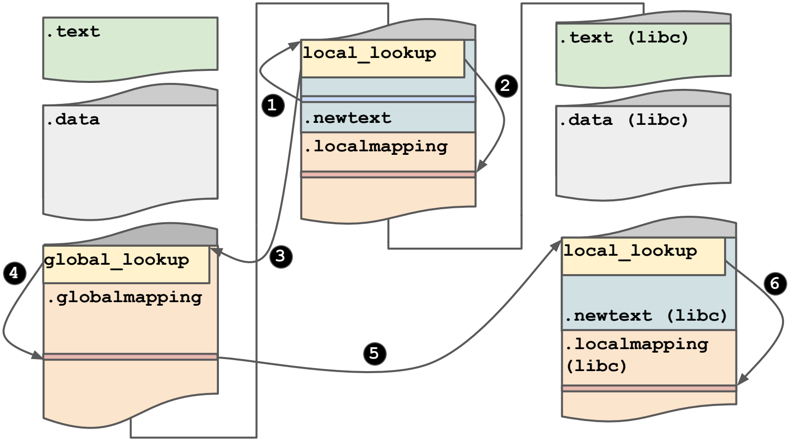
Mapping Lookups



Mapping Lookups



Mapping Lookups



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

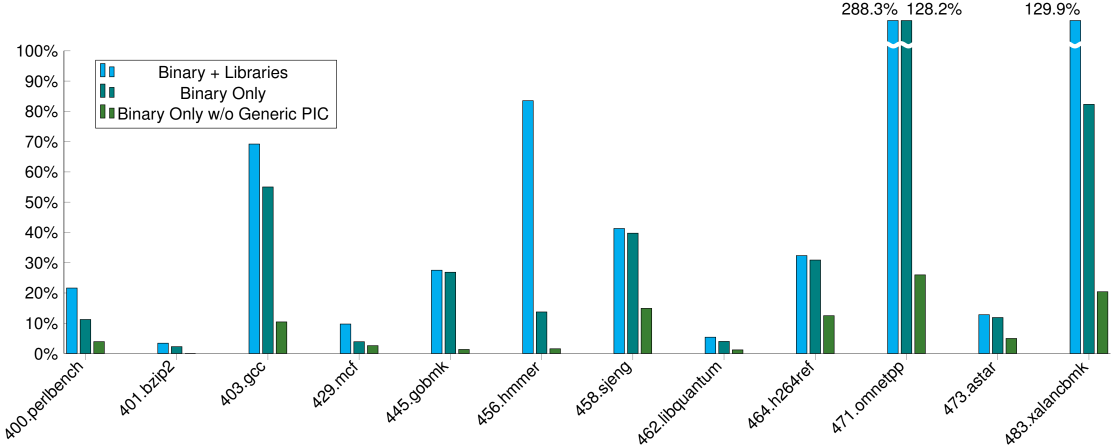
Optimization 1: Only Rewrite Main Binary

- 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

MULTIVERSE Overhead



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

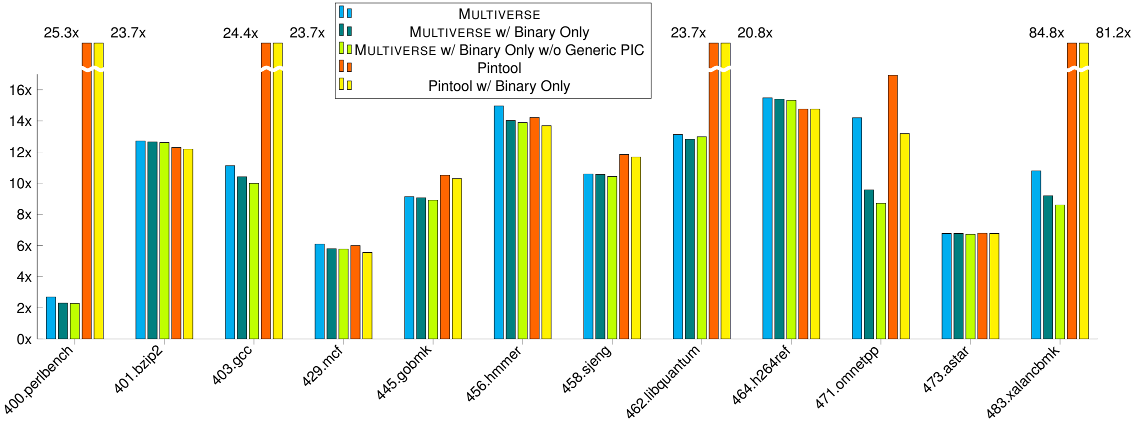
- Ultimate purpose of a rewriter is to insert instrumentation code
- Created straightforward instrumentation API
- For evaluation created instruction counting instrumentation in MULTIVERSE

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



Security Applications Evaluation

Shadow Stack

- 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

Security Applications Evaluation

Shadow Stack

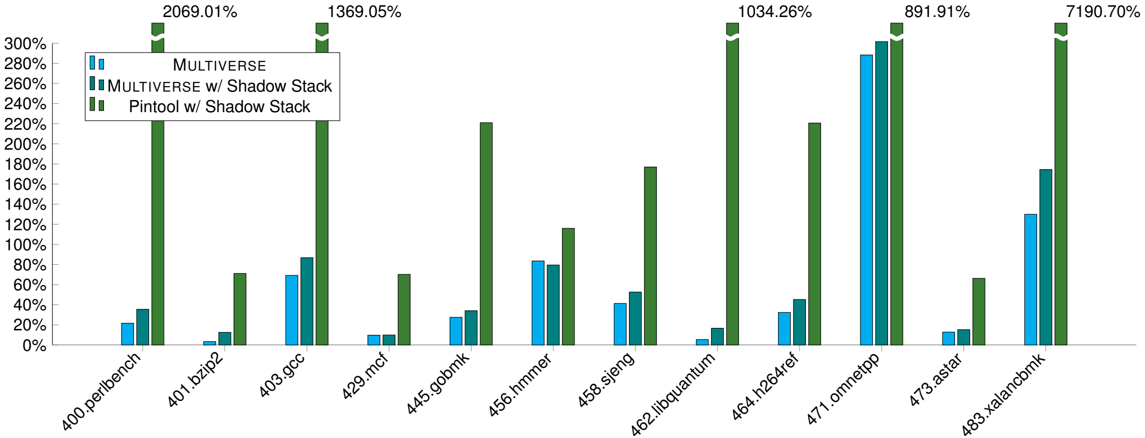
- An appealing application of rewriters is binary hardening
- Shadow stacks implement a form of backward-edge CFI
- Implemented a simple shadow stack in MULTIVERSE

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

Shadow Stack Overhead



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

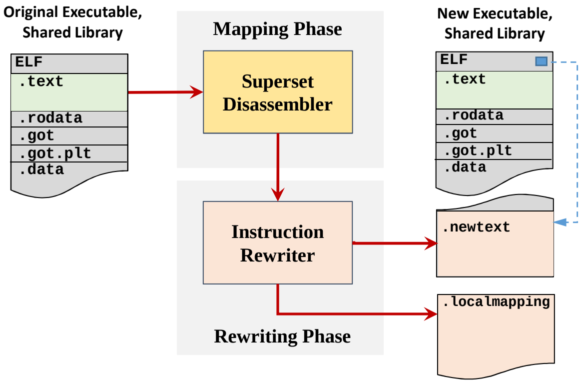
Optimization

- 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



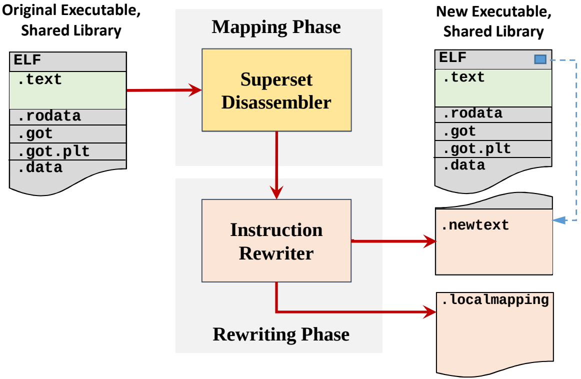
MULTIVERSE

- 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



{erick.bauman,hamlen}@utdallas.edu
zlin@cse.ohio-state.edu

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 (TISSEC) **13** (2009), no. 1.
- Andrew R. Bernat and Barton P. Miller, Anywhere, any-time binary instrumentation, Proc. 10th ACM SIGPLAN-SIGSOFT Work. Program Analysis for Software Tools (PASTE), 2011, pp. 9–16.
- Xi Chen, Herbert Bos, and Cristiano Giuffrida, CodeArmor: Virtualizing the code space to counter disclosure attacks, Proc. 2nd IEEE Sym. Security and Privacy (EuroS&P), 2017, pp. 514–529.
- Brian Cox and Jeffrey Robert. Forshaw, The quantum universe: everything that can happen does happen, Penguin, 2012.
- Zhui Deng, Xiangyu Zhang, and Dongyan Xu, Bistro: Binary component extraction and embedding for software security applications, Proc. 18th European Sym. Research in Computer Security (ESORICS), 2013, pp. 200–218.
- Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula, XFI: Software guards for system address spaces, Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI), 2006, pp. 75–88.
- Úlfar Erlingsson and Fred B. Schneider, SASI enforcement of security policies: A retrospective, Proc. New Security Paradigms Work. (NSPW), 1999, pp. 87–95.

References II

- Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna, Static disassembly of obfuscated binaries, Proc. 13th USENIX Security Sym., 2004.
- Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely, PEBIL: Efficient static binary instrumentation for Linux, Proc. IEEE Int. Sym. Performance Analysis Systems and Software (ISPASS), 2010, pp. 175–183.
- Evangelos Ladakis, Giorgos Vasiliadis, Michalis Polychronakis, Sotiris Ioannidis, and Georgios Portokalidis, GPU-Disasm: A GPU-based x86 disassembler, Int. Information Security Conf., 2015, pp. 472–489.
- Stephen McCamant and Greg Morrisett, Evaluating SFI for a CISC architecture, Proc. 15th USENIX Security Sym., 2006.
- Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh, BIRD: Binary interpretation using runtime disassembly, Proc. 4th IEEE/ACM Int. Sym. Code Generation and Optimization (CGO), 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. 26th IFIP TC Int. Information Security Conf. (SEC), 2011, pp. 154–172.

References III

- 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, Proc. 5th IEEE Int. Sym. Signal Processing and Information Technology (ISSPIT), 2005, pp. 7–12.
- Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi, Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation, Proc. 37th IEEE/ACM Int. Sym. Microarchitecture (MICRO), 2004, pp. 81–92.
- 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, Proc. USENIX Windows NT Work., 1997, pp. 1–7.
- Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre, Plto: A link-time optimizer for the Intel IA-32 architecture, Proc. Work. Binary Translation (WBT), 2001.
- Amitabh Srivastava, Andrew Edwards, and Hoi Vo, Vulcan: Binary transformation in a distributed environment, Tech. Report MSR-TR-2001-50, Microsoft Research, 2001.
- Richard Wartell, Vishwath Mohan, Kevin Hamlen, and Zhiqiang Lin, Securing untrusted code via compiler-agnostic binary rewriting, Proc. 28th Annual Computer Security Applications Conf. (ACSAC), 2012, pp. 299–308.

References IV

- Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin, Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code, Proc. 19th ACM Conf. Computer and Communications Security (CCS), 2012, pp. 157–168.
- Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna, Ramblr: Making reassembly great again, Proc. 24th Annual Network & Distributed System Security Sym. (NDSS), 2017.
- Shuai Wang, Pei Wang, and Dinghao Wu, Reassembleable disassembling, Proc. 24th USENIX Security Sym., 2015, pp. 627–642.
- ______, UROBOROS: Instrumenting stripped binaries with static reassembling, Proc. IEEE 23rd Int. Conf. Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 236–247.
- Richard Wartell, Yan Zhou, Kevin W Hamlen, and Murat Kantarcioglu, Shingled graph disassembly: Finding the undecideable path, Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD), 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. 30th IEEE Sym. Security & Privacy (S&P), 2009, pp. 79–93.

References V

- Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar, A platform for secure static binary instrumentation, Proc. 10th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments (VEE), 2014, pp. 129–140.
- Mingwei Zhang and R. Sekar, Control flow integrity for COTS binaries, Proc. 22nd USENIX Security Sym., 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, Proc. 34th IEEE Sym. Security & Privacy (S&P), 2013, pp. 559–573.