Go With the Flow

Enforcing Program Behavior Through Syscall Sequences and Origins

Claudio Canella (🐦@cc0x1f)

August 11, 2022

Graz University of Technology

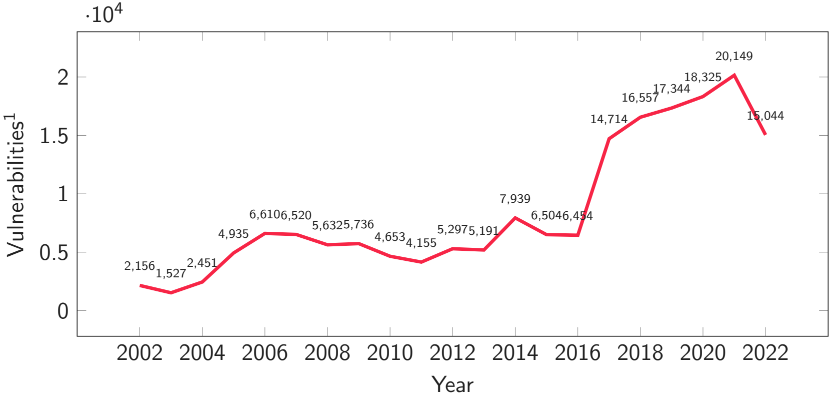
Who am I?

Claudio Canella

PhD Candidate @ Graz University of Technology

🐦 @cc0x1f

✉ claudio.canella@iaik.tugraz.at



[1] Source: `http://www.cvedetails.com/vulnerabilities-by-types.php`

What to do?

Eliminate bugs

What to do?

Eliminate bugs

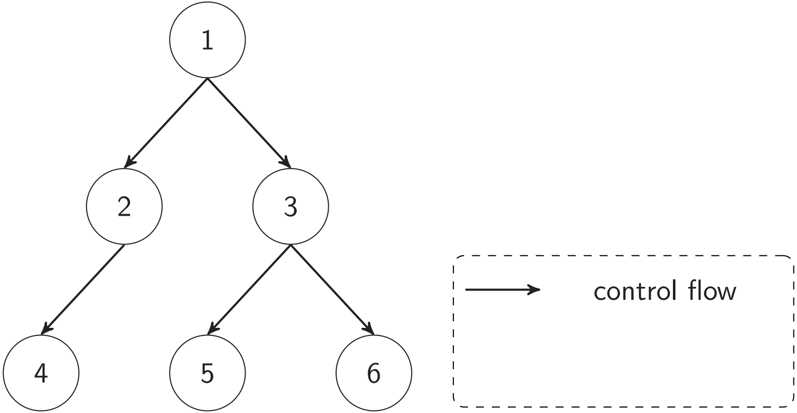
Limit Post-Exploitation Impact

What to do?

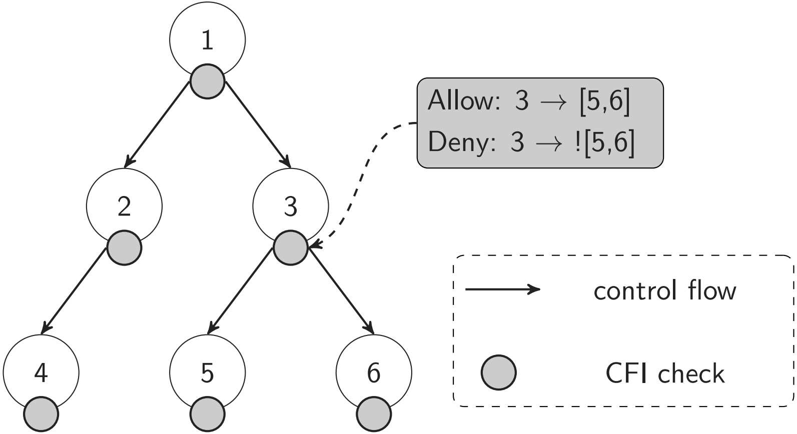
Eliminate bugs

Limit Post-Exploitation Impact

Control-Flow Integrity



Control-Flow Integrity



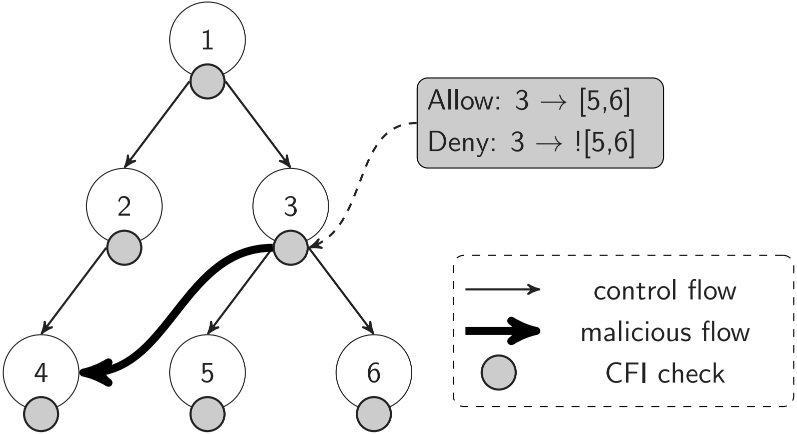
Allow: 3 → [5,6]

Deny: 3 → ![5,6]

→ control flow

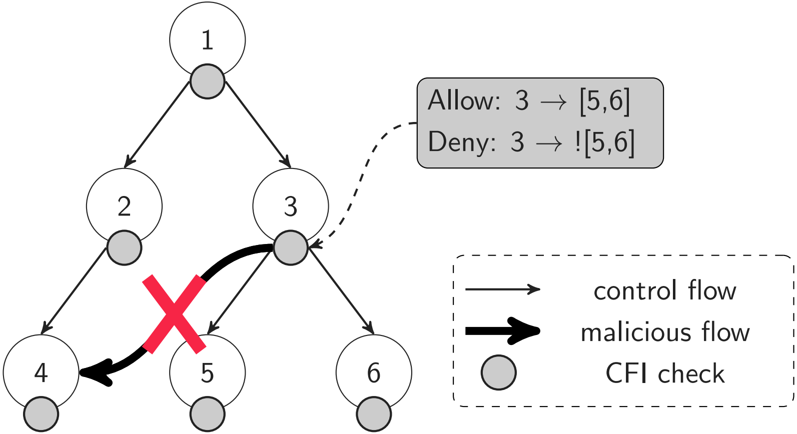
● CFI check

Control-Flow Integrity



Allow: 3 → [5,6]
Deny: 3 → ![5,6]

Control-Flow Integrity

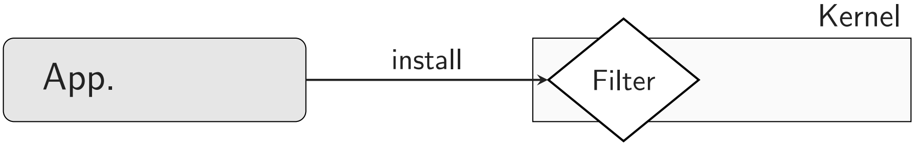


Allow: 3 → [5,6]
Deny: 3 → ![5,6]

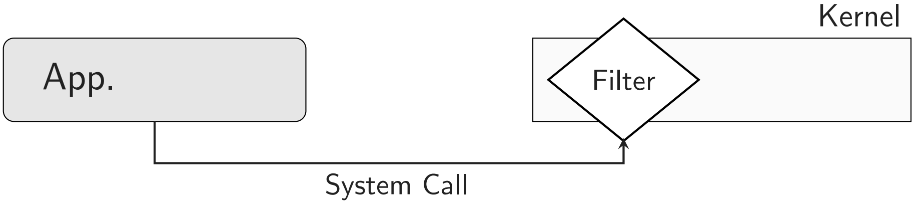
→ control flow
➔ malicious flow
● CFI check

Linux Seccomp

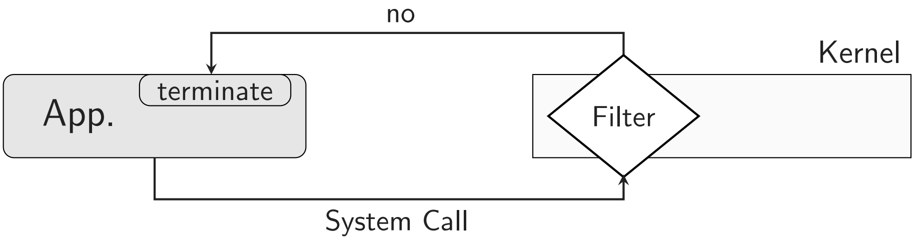
Linux Seccomp



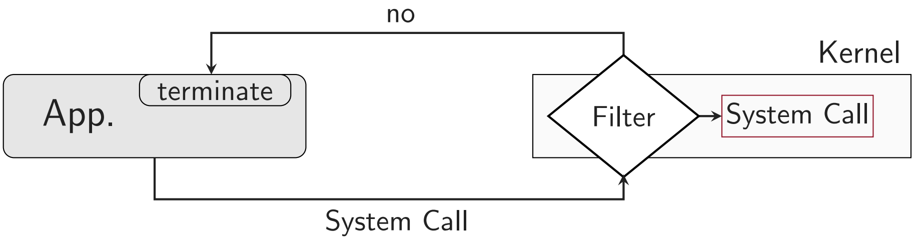
Linux Seccomp



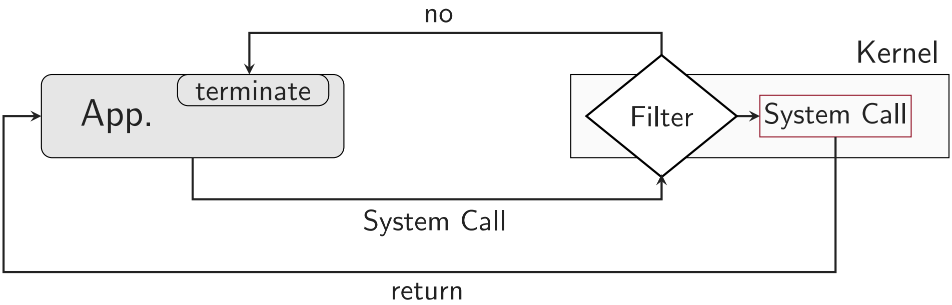
Linux Seccomp



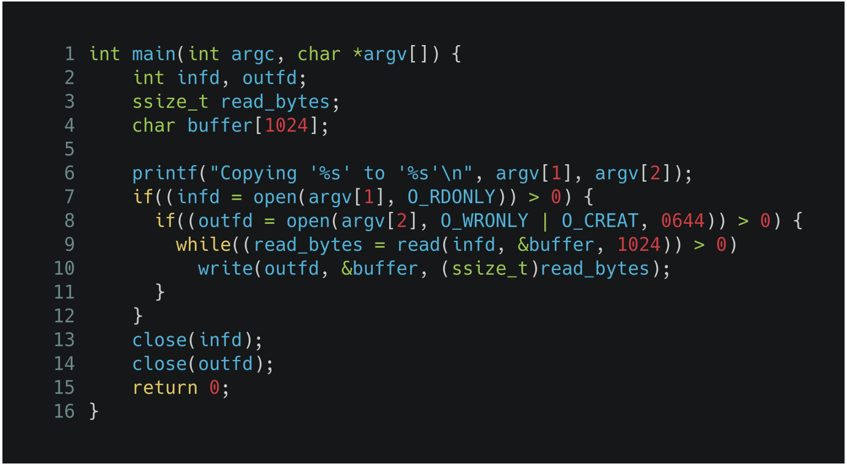
Linux Seccomp



Linux Seccomp



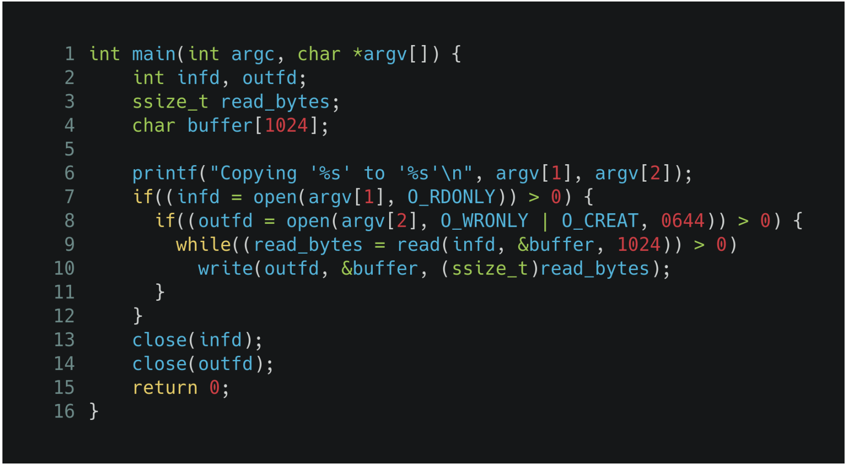
Sample



```c
int main(int argc, char *argv[]) {
    int infd, outfd;
    ssize_t read_bytes;
    char buffer[1024];

    printf("Copying '%s' to '%s'\n", argv[1], argv[2]);
    if((infd = open(argv[1], O_RDONLY)) > 0) {
      if((outfd = open(argv[2], O_WRONLY | O_CREAT, 0644)) > 0) {
        while((read_bytes = read(infd, &buffer, 1024)) > 0)
          write(outfd, &buffer, (ssize_t)read_bytes);
      }
    }
    close(infd);
    close(outfd);
    return 0;
}
```

Sample



```c
int main(int argc, char *argv[]) {
    int infd, outfd;
    ssize_t read_bytes;
    char buffer[1024];

    printf("Copying '%s' to '%s'\n", argv[1], argv[2]);
    if((infd = open(argv[1], O_RDONLY)) > 0) {
      if((outfd = open(argv[2], O_WRONLY | O_CREAT, 0644)) > 0) {
        while((read_bytes = read(infd, &buffer, 1024)) > 0)
          write(outfd, &buffer, (ssize_t)read_bytes);
      }
    }
    close(infd);
    close(outfd);
    return 0;
}
```

Syscalls: 0 1 2 3 16 19 20 60 72 202 231

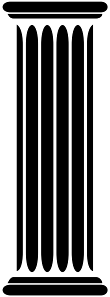
ENTER SANDBOX

https://github.com/chestnut-sandbox/Chestnut

Claudio Canella (@cc0x1f), Mario Werner (we.rner.at), Michael Schwarz (@misc0110)

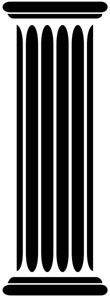
Syscall-Flow-Integrity Protection

Syscall-Flow-Integrity Protection

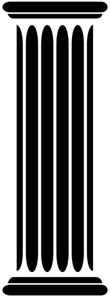


State Machine

Syscall-Flow-Integrity Protection

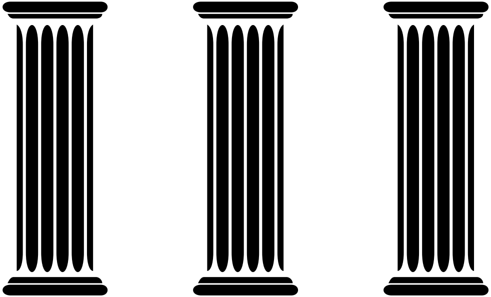


State Machine



Origins

Syscall-Flow-Integrity Protection

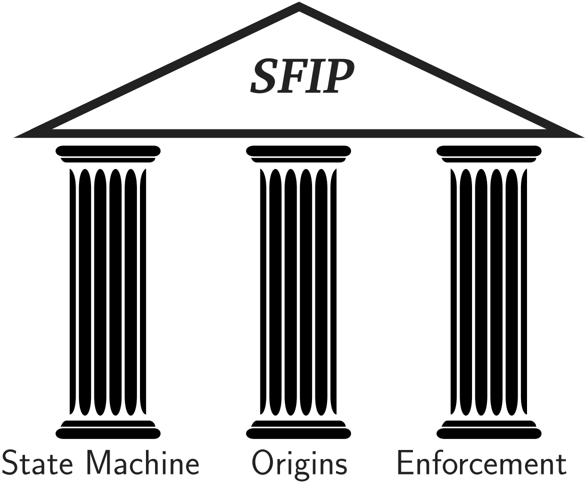


State Machine

Origins

Enforcement

Syscall-Flow-Integrity Protection



SysFlow

Compiler: Extraction

SysFlow

Compiler: Extraction

Library: Setup

SysFlow

Compiler: Extraction

Library: Setup

Kernel: Enforcement

Syscall and CFG Extraction

Source Code

```
L01: void foo(int test) {
L02:   scanf(...);
L03:   if(test)
L04:     printf(...)
L05:   else
L06:     syscall(read, ...);
L07:   int ret = bar(...);
L08:   if(!ret)
L09:       exit(0);
L10:   return ret;
L11: }
```

Syscall and CFG Extraction

Source Code

```
L01: void foo(int test) {
L02:   scanf(...);
L03:   if(test)
L04:     printf(...)
L05:   else
L06:     syscall(read, ...);
L07:   int ret = bar(...);
L08:   if(!ret)
L09:       exit(0);
L10:   return ret;
L11: }
```

extract →

Extracted Function Info

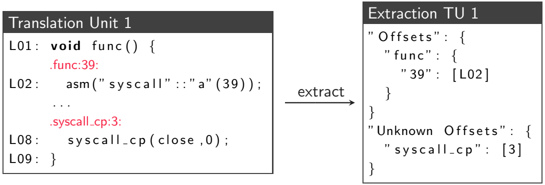
```
{
  "Transitions": {
    "L03": [L04,L06],
    "L04": [L07],
    "L06": [L07]
    "L08": [L09,L10]
  }
  "Call Targets": {
    "L02": ["scanf"],
    "L04": ["printf"],
    "L07": ["bar"],
    "L09": ["exit"],
  }
  "Syscalls": {
    "L06" : [read]
  }
}
```

Syscall Offset Extraction

Translation Unit 1

```
void func() {
.func:39:
  asm("syscall"::"a"(39));
  ...
.syscall_cp:3:
  syscall_cp(close,0);
}
```

Syscall Offset Extraction



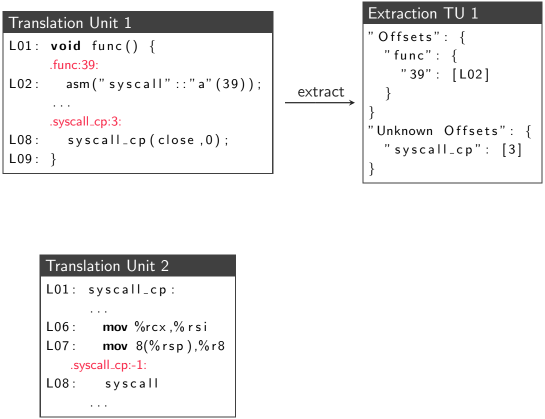
Translation Unit 1

```
L01: void func() {
     .func:39:
L02:   asm("syscall"::"a"(39));
     ...
     .syscall_cp:3:
L08:   syscall_cp(close,0);
L09: }
```

Extraction TU 1

```
"Offsets": {
  "func": {
    "39": [L02]
  }
}
"Unknown Offsets": {
  "syscall_cp": [3]
}
```

Syscall Offset Extraction



Translation Unit 1

```
L01: void func() {
       .func:39:
L02:     asm("syscall"::"a"(39));
       ...
       .syscall_cp:3:
L08:     syscall_cp(close,0);
L09: }
```

extract

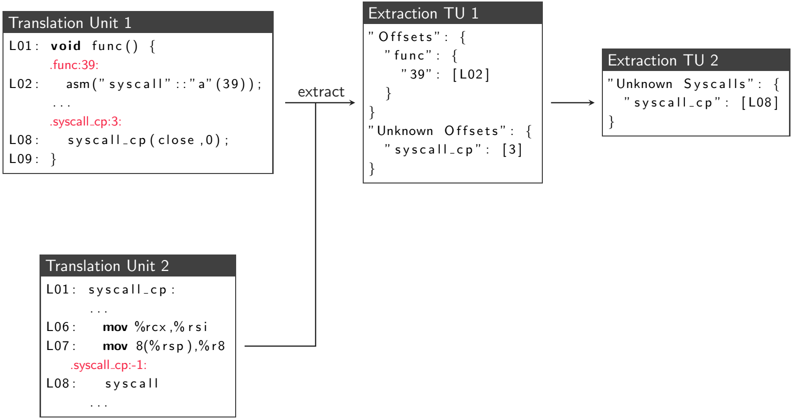
Extraction TU 1

```
"Offsets": {
  "func": {
    "39": [L02]
  }
}
"Unknown Offsets": {
  "syscall_cp": [3]
}
```

Translation Unit 2

```
L01: syscall_cp:
       ...
L06:     mov %rcx,%rsi
L07:     mov 8(%rsp),%r8
       .syscall_cp:-1:
L08:     syscall
       ...
```

Syscall Offset Extraction



Translation Unit 1

```
L01: void func() {
     .func:39:
L02:    asm("syscall"::"a"(39));
        ...
     .syscall_cp:3:
L08:    syscall_cp(close,0);
L09: }
```

Translation Unit 2

```
L01: syscall_cp:
        ...
L06:    mov %rcx,%rsi
L07:    mov 8(%rsp),%r8
      .syscall_cp:-1:
L08:    syscall
        ...
```

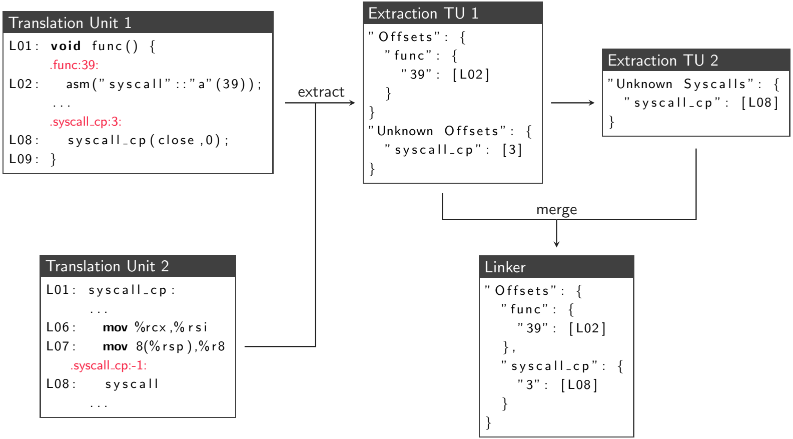
Extraction TU 1

```
"Offsets": {
  "func": {
    "39": [L02]
  }
}
"Unknown Offsets": {
  "syscall_cp": [3]
}
```

Extraction TU 2

```
"Unknown Syscalls": {
  "syscall_cp": [L08]
}
```

Syscall Offset Extraction



Translation Unit 1

```
L01: void func() {
     .func:39:
L02:    asm("syscall"::"a"(39));
     ...
     .syscall_cp:3:
L08:    syscall_cp(close,0);
L09: }
```

Translation Unit 2

```
L01: syscall_cp:
       ...
L06:     mov %rcx,%rsi
L07:     mov 8(%rsp),%r8
       .syscall_cp:-1:
L08:     syscall
       ...
```

Extraction TU 1

```
"Offsets": {
  "func": {
    "39": [L02]
  }
}
"Unknown Offsets": {
  "syscall_cp": [3]
}
```

Extraction TU 2

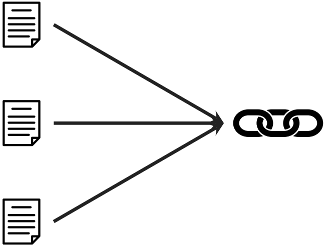
```
"Unknown Syscalls": {
  "syscall_cp": [L08]
}
```

Linker

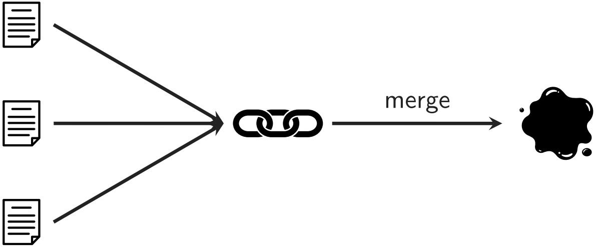
```
"Offsets": {
  "func": {
    "39": [L02]
  },
  "syscall_cp": {
    "3": [L08]
  }
}
```

State Machine Generation

State Machine Generation



State Machine Generation



State Machine Generation

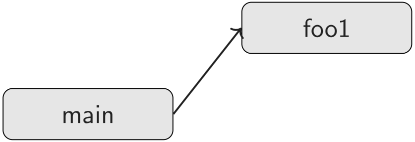
Info main

```
Call Targets: {
  "L56": [foo1],
  "L59": [foo2]
}
```

Last Syscalls

State Machine

State Machine Generation



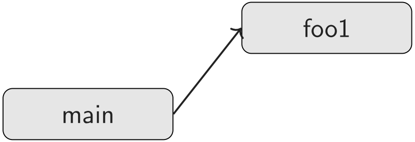
Info foo1

```
Call Targets: {
  "L03": [bar1]
}
Syscalls: {
  "L02": [open]
}
```

Last Syscalls

State Machine

State Machine Generation



Info foo1

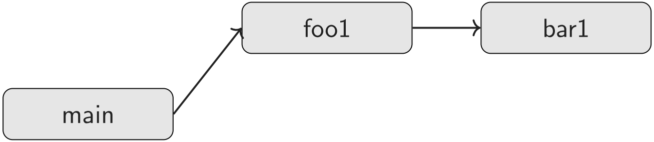
```
Call Targets: {
  "L03": [bar1]
}
Syscalls: {
}
```

Last Syscalls

open

State Machine

State Machine Generation



Info bar1

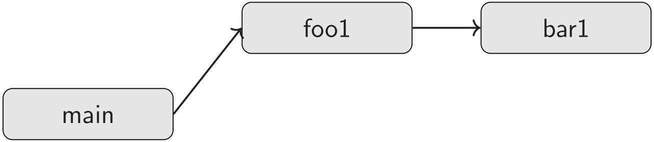
```
Syscalls: {
  "L12": [read]
}
```

Last Syscalls

open

State Machine

State Machine Generation



Info bar1

```
Syscalls: {
  "L12": [read]
}
```

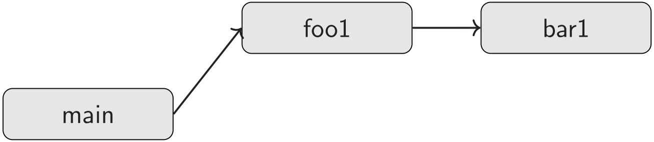
Last Syscalls

open

State Machine

open: [read]

State Machine Generation



Info bar1

Syscalls: {
}

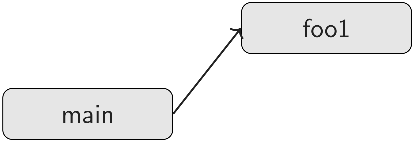
Last Syscalls

read

State Machine

open: [read]

State Machine Generation



Info foo1

```
Call Targets: {
}
Syscalls: {
}
```

Last Syscalls

read

State Machine

open: [read]

State Machine Generation

main

Info main

```
Call Targets: {
  "L59": [foo2]
}
```

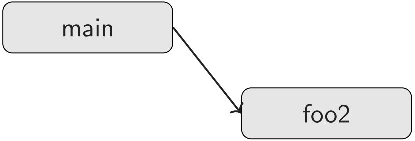
Last Syscalls

read

State Machine

open: [read]

State Machine Generation



Info foo2

```
Call Targets: {
  "L179": [bar2]
}
Syscalls: {
  "L178": [open]
}
```

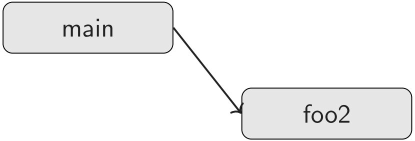
Last Syscalls

read

State Machine

open: [read]

State Machine Generation



Info foo2

```
Call Targets: {
  "L179": [bar2]
}
Syscalls: {
  "L178": [open]
}
```

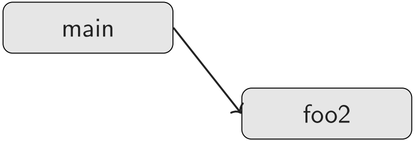
Last Syscalls

read

State Machine

open: [read]
read: [open]

State Machine Generation



Info foo2

```
Call Targets: {
  "L179": [bar2]
}
Syscalls: {
}
```

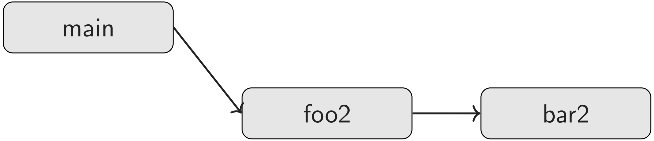
Last Syscalls

open

State Machine

open: [read]
read: [open]

State Machine Generation



Info bar2

```
Syscalls: {
  "L162": [stat]
}
```

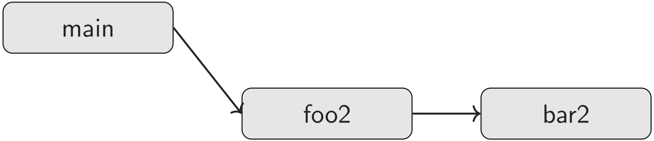
Last Syscalls

open

State Machine

open: [read]
read: [open]

State Machine Generation



Info bar2

```
Syscalls: {
}
```

Last Syscalls

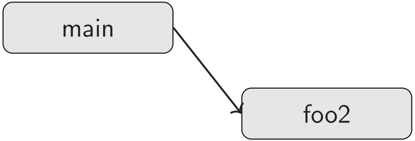
stat

State Machine

open: [read,stat]

read: [open]

State Machine Generation



Info foo2

```
Call Targets: {
}
Syscalls: {
}
```

Last Syscalls

stat

State Machine

open: [read,stat]
read: [open]

State Machine Generation

Info main

```
Call Targets: {
}
```

Last Syscalls

stat

State Machine

open: [read,stat]

read: [open]

Enforcement

Library

- extracts information

Enforcement

Library

- extracts information
- makes offset adjustment

Enforcement

Library

- extracts information
- makes offset adjustment

Kernel

- performs transition check

Enforcement

Library

- extracts information
- makes offset adjustment

Kernel

- performs transition check
- performs independent origin check

Evaluation

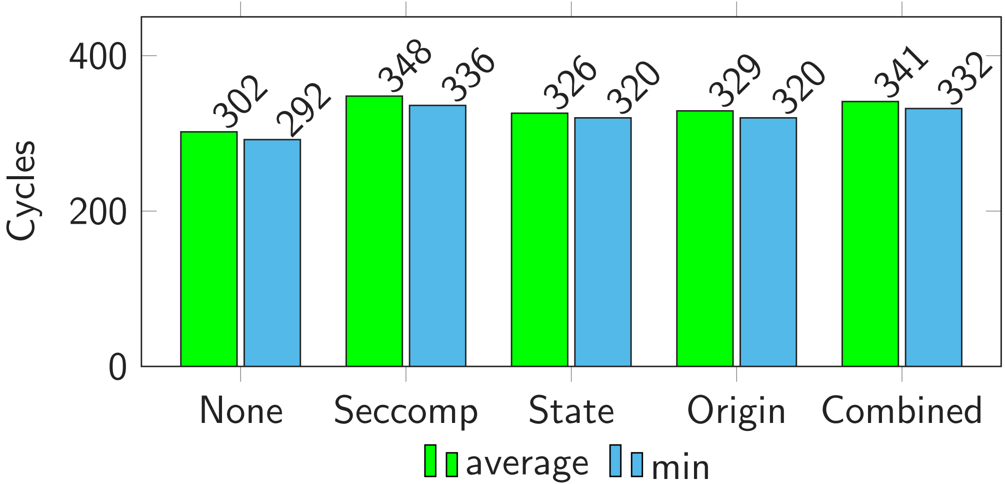
Performance

Evaluation

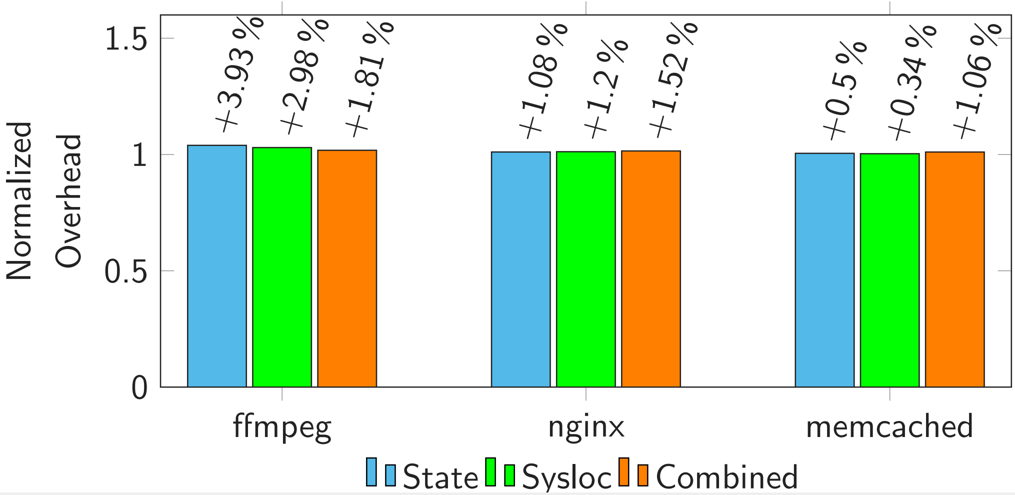
Performance

Security

Microbenchmark



Macrobenchmark



State Machine Analysis

Application	Average Transitions	#States
busybox	15.99	23.52
coreutils	16.66	26.64
pwgen	13.56	18
muraster	18.89	29
nginx	74.05	107
ffmpeg	49.07	55
memcached	43.16	86
mutool	32.26	53

State Machine Analysis

Application	Average Transitions	#States
busybox	15.99	23.52
coreutils	16.66	26.64
pwgen	13.56	18
muraster	18.89	29
nginx	74.05	107
ffmpeg	49.07	55
memcached	43.16	86
mutool	32.26	53

Origin Analysis

Application	Total #Offsets	Avg #Offsets
busybox	102.64	3.75
coreutils	116.71	4.42
pwgen	84	4.42
muraster	193	4.6
nginx	318	3.0
ffmpeg	279	4.98
memcached	317	3.69
mutool	278	4.15

Origin Analysis

Application	Total #Offsets	Avg #Offsets
busybox	102.64	3.75
coreutils	116.71	4.42
pwgen	84	4.42
muraster	193	4.6
nginx	318	3.0
ffmpeg	279	4.98
memcached	317	3.69
mutool	278	4.15

Return-oriented programming

- Use exisiting code to exploit a program

Return-oriented programming

- Use exisiting code to exploit a program
- Jumps to parts of functions (so called gadgets)

Return-oriented programming

- Use exisiting code to exploit a program
- Jumps to parts of functions (so called gadgets)
- These *gadgets* are assembler instructions followed by a `ret`
  - `pop RDI; retq`
  - `syscall; retq`
  - `add RSP, 8; retq`

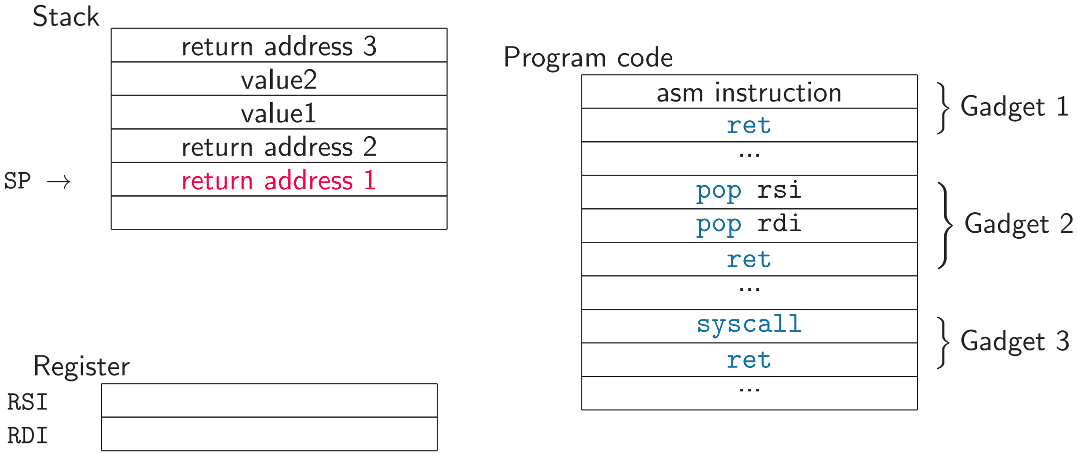
Return-oriented programming

- Use exisiting code to exploit a program
- Jumps to parts of functions (so called gadgets)
- These *gadgets* are assembler instructions followed by a `ret`
    - `pop RDI; retq`
    - `syscall; retq`
    - `add RSP, 8; retq`
- Gadgets are chained together for an exploit

Return-oriented programming

- Use exisiting code to exploit a program
- Jumps to parts of functions (so called gadgets)
- These *gadgets* are assembler instructions followed by a `ret`
  - `pop RDI; retq`
  - `syscall; retq`
  - `add RSP, 8; retq`
- Gadgets are chained together for an exploit
- Overwrite the stack with gadget addresses and parameters

Return-oriented programming



Stack

- return address 3
- value2
- value1
- return address 2
- return address 1

SP →

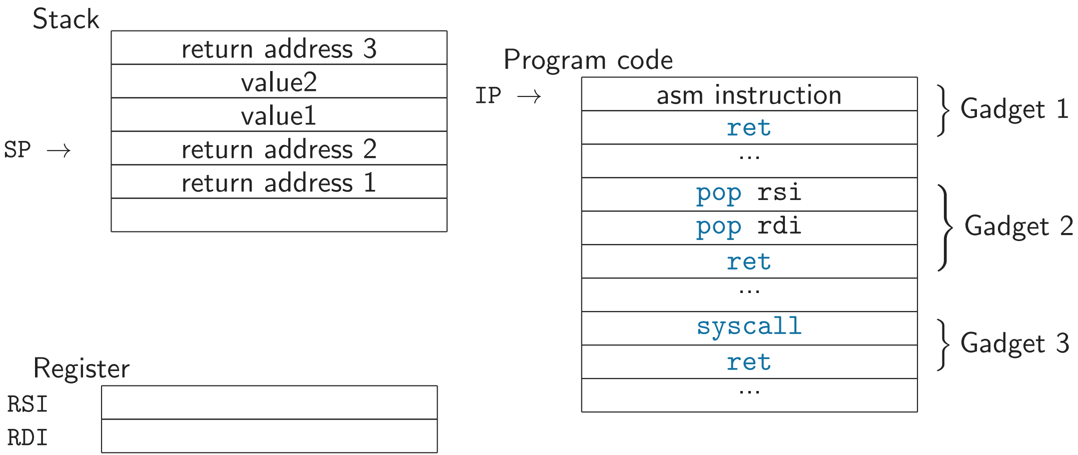
Register

- RSI
- RDI

Program code

```
asm instruction   } Gadget 1
ret
...
pop rsi           } Gadget 2
pop rdi
ret
...
syscall           } Gadget 3
ret
...
```

Return-oriented programming



Stack

return address 3

value2

value1

SP → return address 2

return address 1

Program code

IP → asm instruction

ret

...

pop rsi

pop rdi

ret

...

syscall

ret

...

Gadget 1

Gadget 2

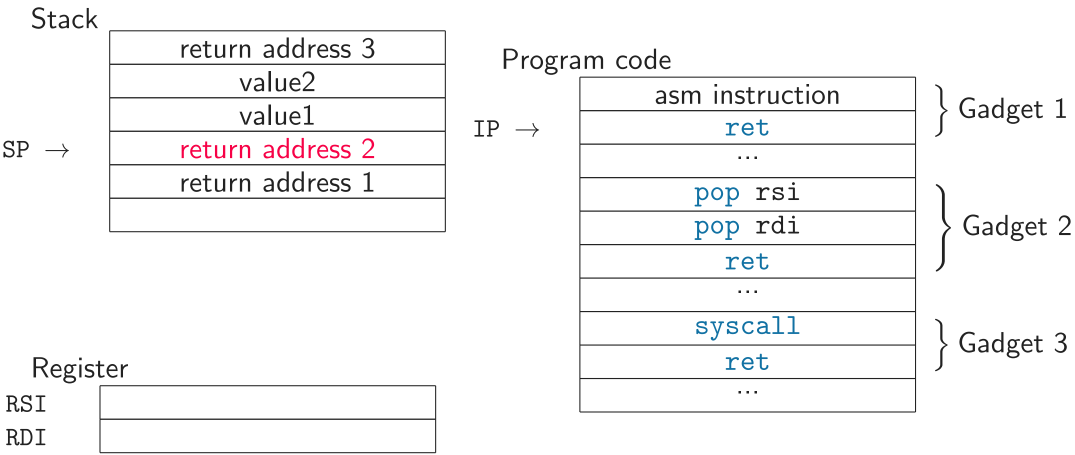
Gadget 3

Register

RSI

RDI

Return-oriented programming



Stack

SP →

return address 3

value2

value1

return address 2

return address 1

Program code

IP →

asm instruction

ret

...

pop rsi

pop rdi

ret

...

syscall

ret

...

Gadget 1

Gadget 2

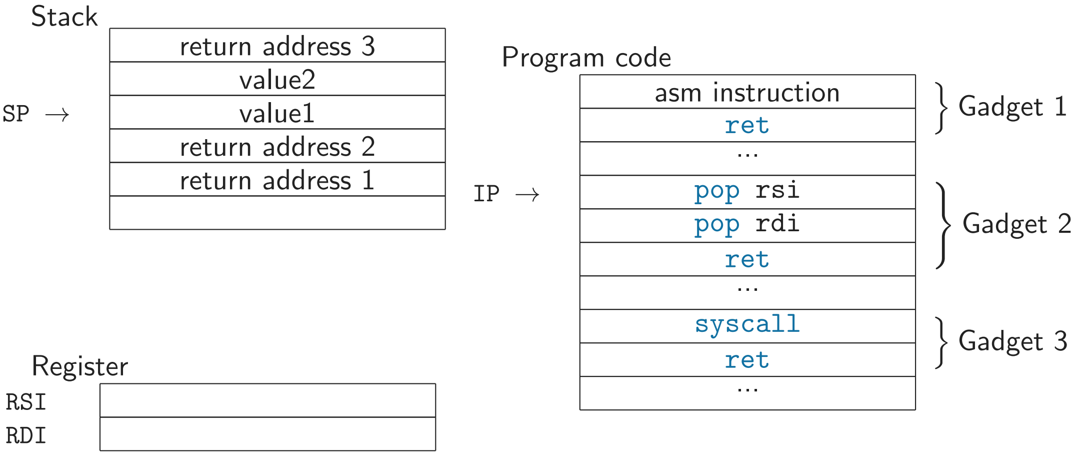
Gadget 3

Register

RSI

RDI

Return-oriented programming



Stack

SP →

return address 3
value2
value1
return address 2
return address 1

Register

RSI
RDI

Program code

IP →

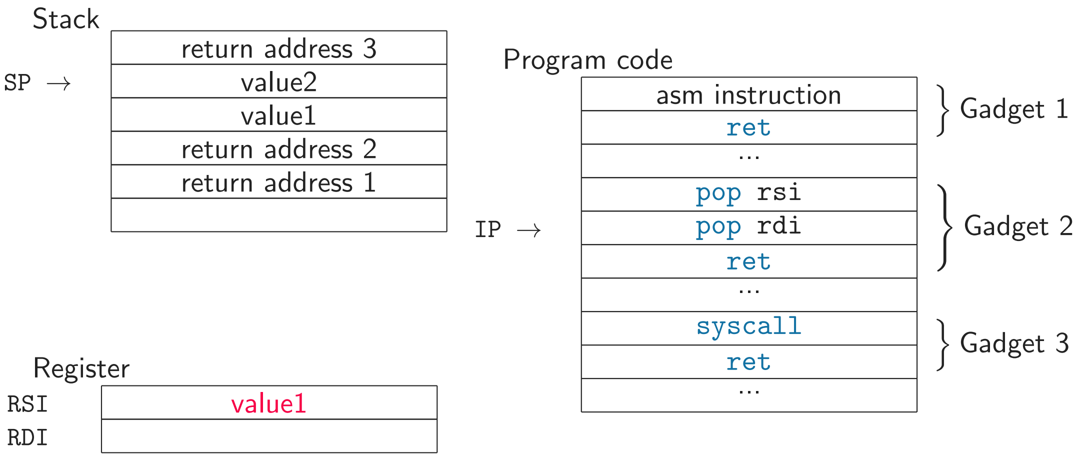
asm instruction
ret
...
pop rsi
pop rdi
ret
...
syscall
ret
...

Gadget 1

Gadget 2

Gadget 3

Return-oriented programming



Stack

SP →

return address 3
value2
value1
return address 2
return address 1

Program code

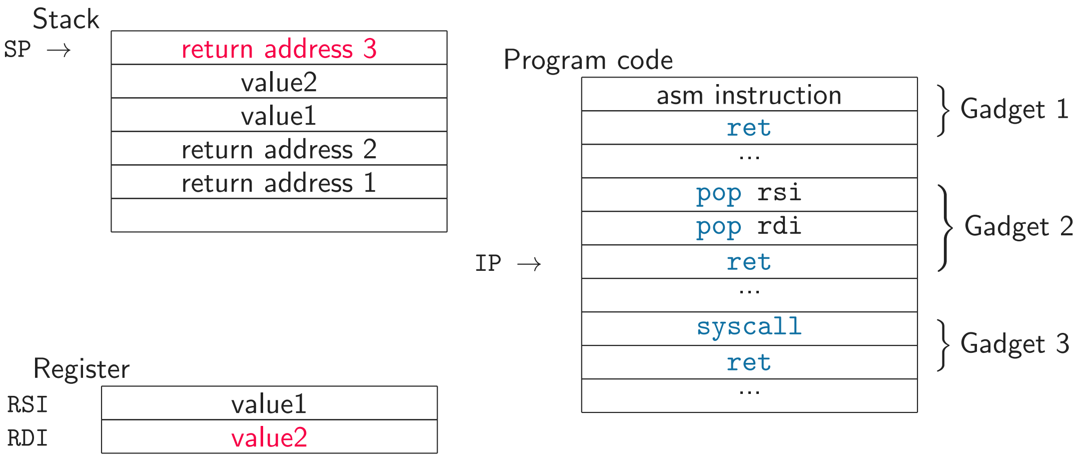
IP →

```
asm instruction   } Gadget 1
ret
...
pop rsi           } Gadget 2
pop rdi
ret
...
syscall           } Gadget 3
ret
...
```

Register

RSI	value1
RDI	

Return-oriented programming



Stack

SP →

return address 3

value2

value1

return address 2

return address 1

Register

RSI value1

RDI value2

Program code

asm instruction

ret

...

pop rsi

pop rdi

ret

...

syscall

ret

...

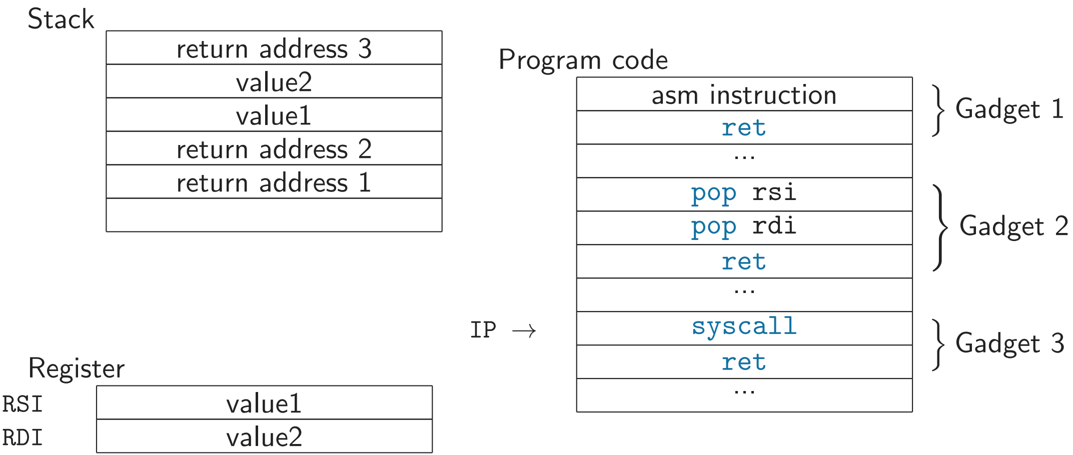
IP →

Gadget 1

Gadget 2

Gadget 3

Return-oriented programming



Stack

return address 3
value2
value1
return address 2
return address 1

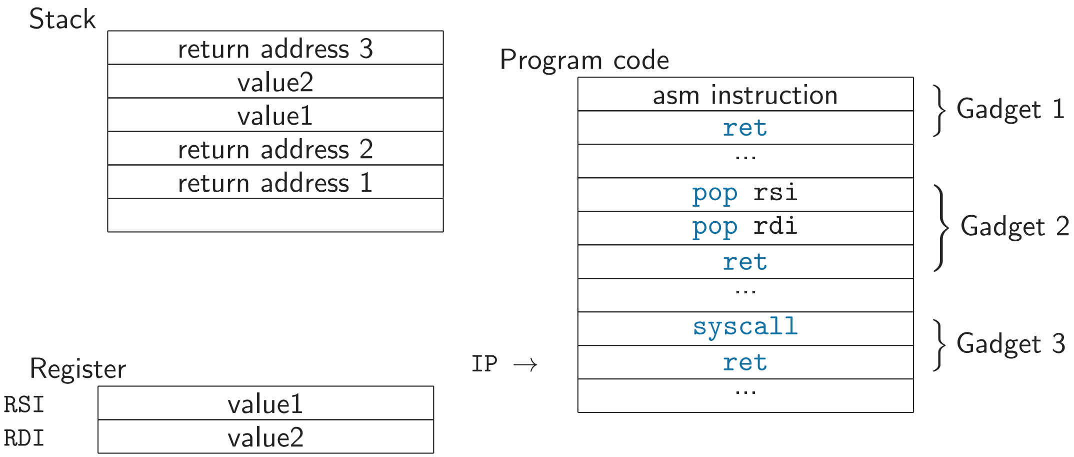
Register

RSI	value1
RDI	value2

Program code

asm instruction	Gadget 1
ret	
...	
pop rsi	Gadget 2
pop rdi	
ret	
...	
syscall	Gadget 3
ret	
...	

Return-oriented programming



Stack

return address 3

value2

value1

return address 2

return address 1

Program code

asm instruction

ret

...

pop rsi

pop rdi

ret

...

syscall

ret

...

Gadget 1

Gadget 2

Gadget 3

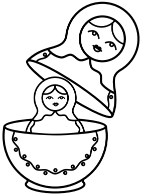
IP →

Register

RSI value1

RDI value2

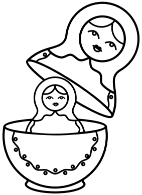
Return-oriented programming



Gadgets are often unintended

- Consider the byte sequence `05 5a 5e 5f c3`

Return-oriented programming

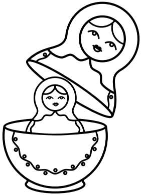


Gadgets are often unintended

- Consider the byte sequence `05 5a 5e 5f c3`
- It disassembles to

```
add eax, 0xc35f5e5a
```

Return-oriented programming



Gadgets are often unintended

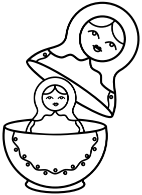
- Consider the byte sequence `05 5a 5e 5f c3`
- It disassembles to

```
add eax, 0xc35f5e5a
```

- However, if we skip the first byte, it disassembles to

```
pop rdx
pop rsi
pop rdi
ret
```

Return-oriented programming



Gadgets are often unintended

- Consider the byte sequence `05 5a 5e 5f c3`
- It disassembles to

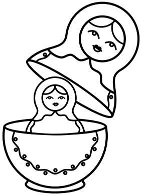
```
add eax, 0xc35f5e5a
```

- However, if we skip the first byte, it disassembles to

```
pop rdx
pop rsi
pop rdi
ret
```

- This property is due to non-aligned, variable-width opcodes

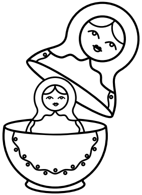
Return-oriented programming



Syscall instruction has byte sequence `0f 05`

→ easy to find unaligned syscall instructions

Return-oriented programming

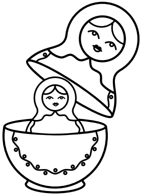


Syscall instruction has byte sequence `0f 05`

→ easy to find unaligned syscall instructions

SFIP restricts ROP chains via

Return-oriented programming



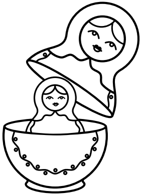
Syscall instruction has byte sequence `0f 05`

→ easy to find unaligned syscall instructions

SFIP restricts ROP chains via

- syscall origins → unaligned instructions not possible

Return-oriented programming



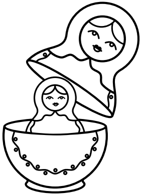
Syscall instruction has byte sequence `0f 05`

→ easy to find unaligned syscall instructions

SFIP restricts ROP chains via

- syscall origins → unaligned instructions not possible
- syscall transitions → not every sequence is possible

Return-oriented programming



Syscall instruction has byte sequence `0f 05`

→ easy to find unaligned syscall instructions

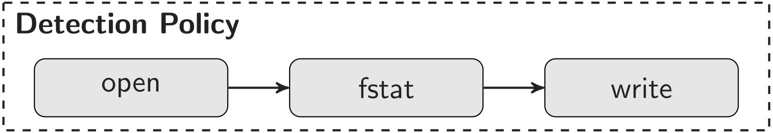
SFIP restricts ROP chains via

- syscall origins → unaligned instructions not possible
- syscall transitions → not every sequence is possible

Conclusion

SFIP imposes significant constraints on control-flow-hijacking attacks

Mimicry Attacks

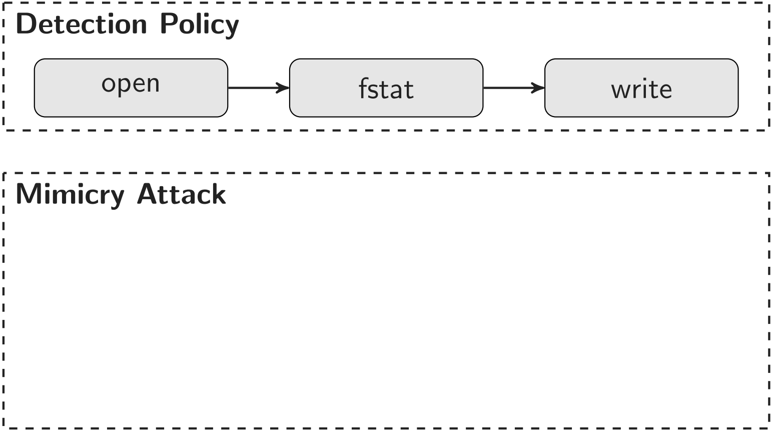

Detection Policy

open

fstat

write

Mimicry Attacks

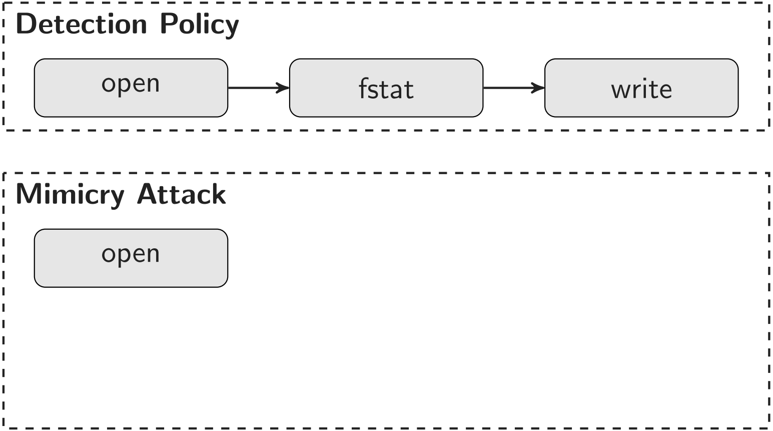


Detection Policy

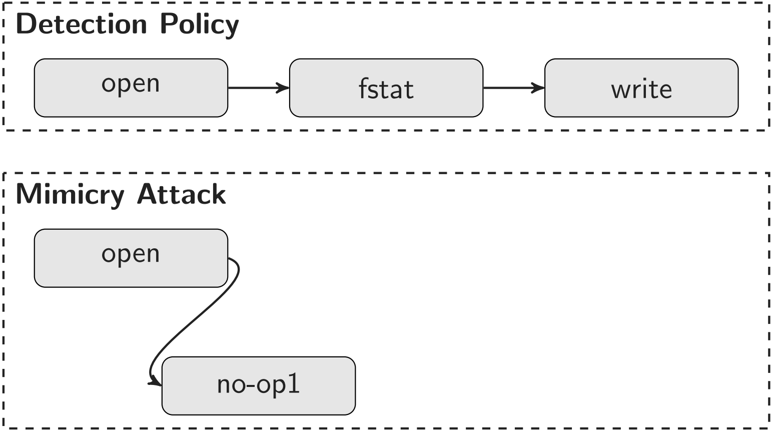
open → fstat → write

Mimicry Attack

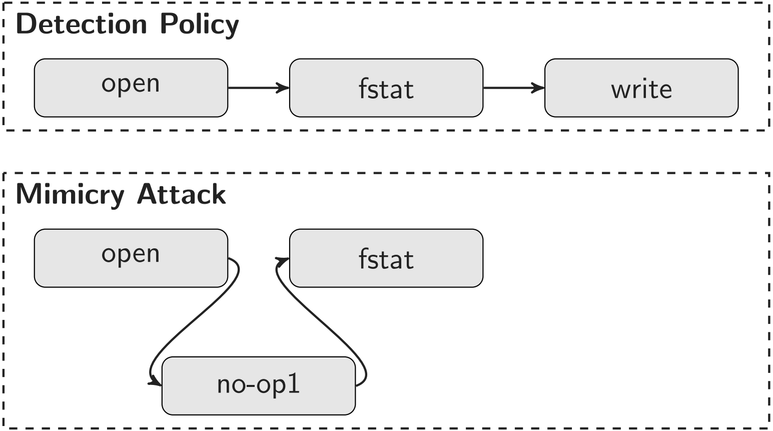
Mimicry Attacks



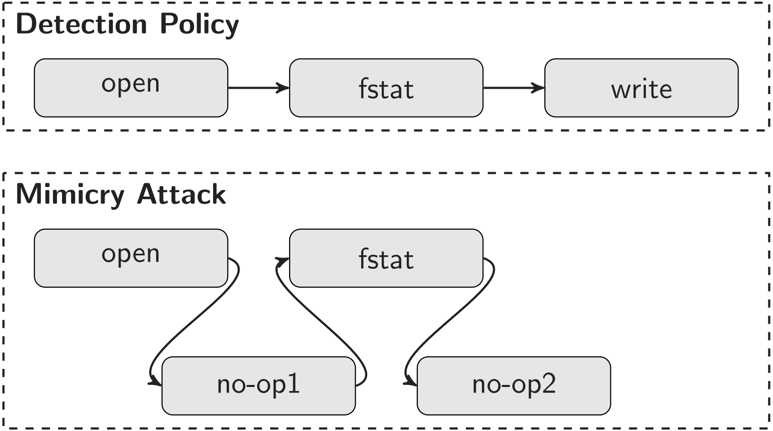
Mimicry Attacks



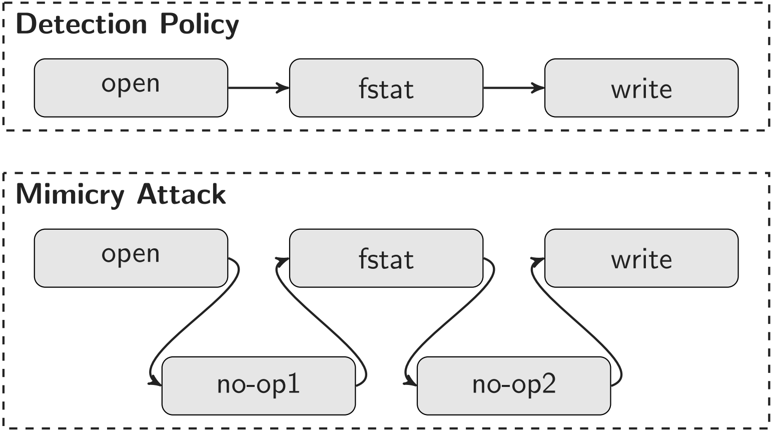
Mimicry Attacks



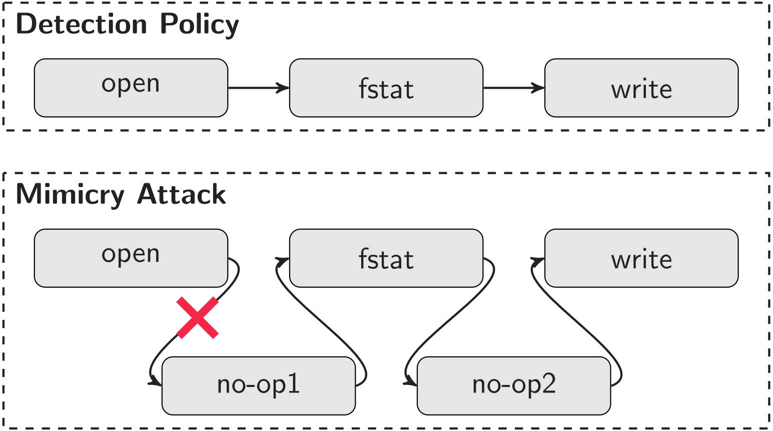
Mimicry Attacks



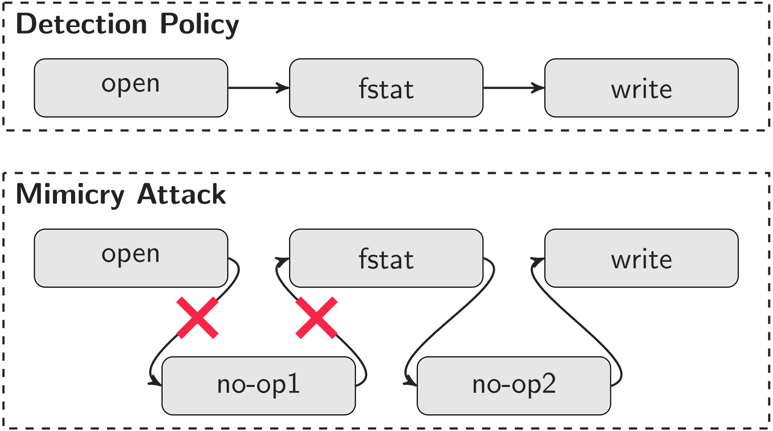
Mimicry Attacks



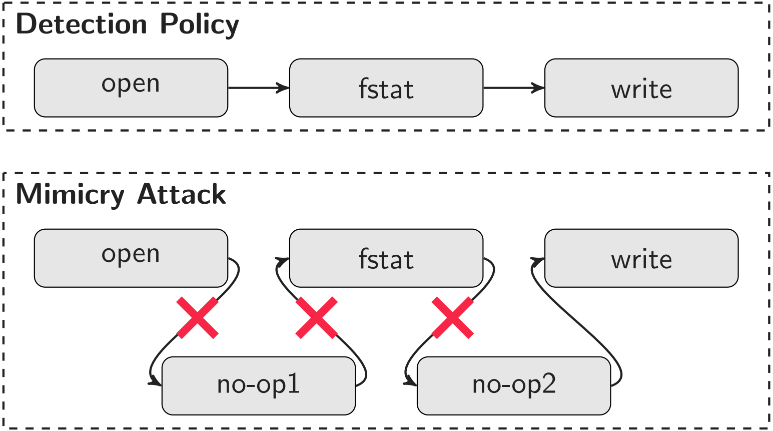
Mimicry Attacks



Mimicry Attacks



Mimicry Attacks



Detection Policy

open → fstat → write

Mimicry Attack

open

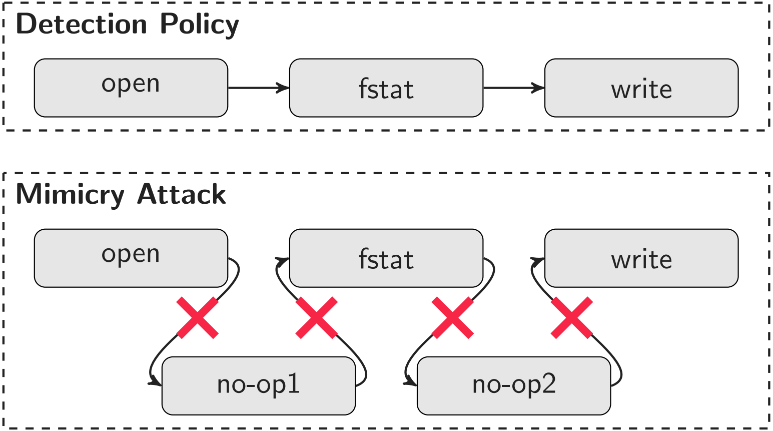
fstat

write

no-op1

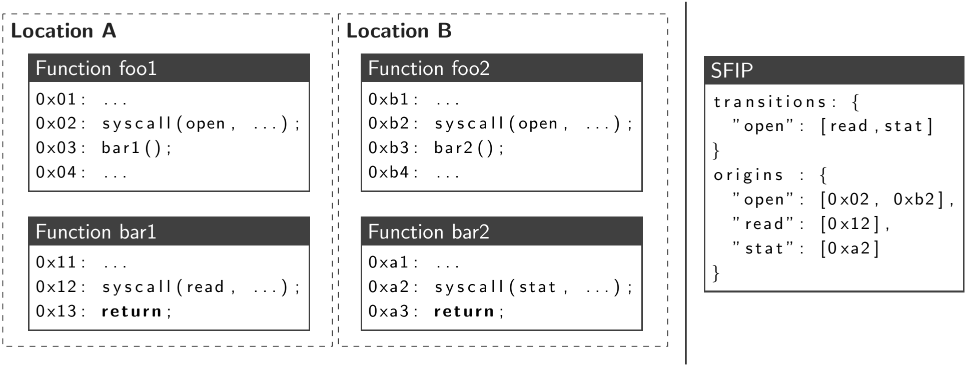
no-op2

Mimicry Attacks



In the near future…

Coarse- to Fine-Grained SFIP



Location A

Function foo1

```
0x01:  ...
0x02:  syscall(open,  ...);
0x03:  bar1();
0x04:  ...
```

Function bar1

```
0x11:  ...
0x12:  syscall(read,  ...);
0x13:  return;
```

Location B

Function foo2

```
0xb1:  ...
0xb2:  syscall(open,  ...);
0xb3:  bar2();
0xb4:  ...
```

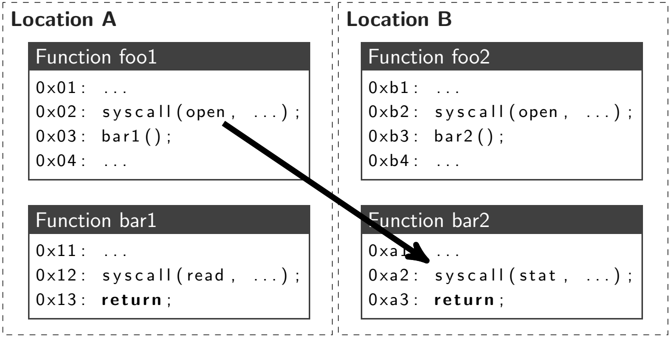
Function bar2

```
0xa1:  ...
0xa2:  syscall(stat,  ...);
0xa3:  return;
```

SFIP

```
transitions: {
  "open": [read,stat]
}
origins : {
  "open": [0x02, 0xb2],
  "read": [0x12],
  "stat": [0xa2]
}
```

Coarse- to Fine-Grained SFIP



Location A

Function foo1

```
0x01:  ...
0x02:  syscall(open, ...);
0x03:  bar1();
0x04:  ...
```

Function bar1

```
0x11:  ...
0x12:  syscall(read, ...);
0x13:  return;
```

Location B

Function foo2

```
0xb1:  ...
0xb2:  syscall(open, ...);
0xb3:  bar2();
0xb4:  ...
```

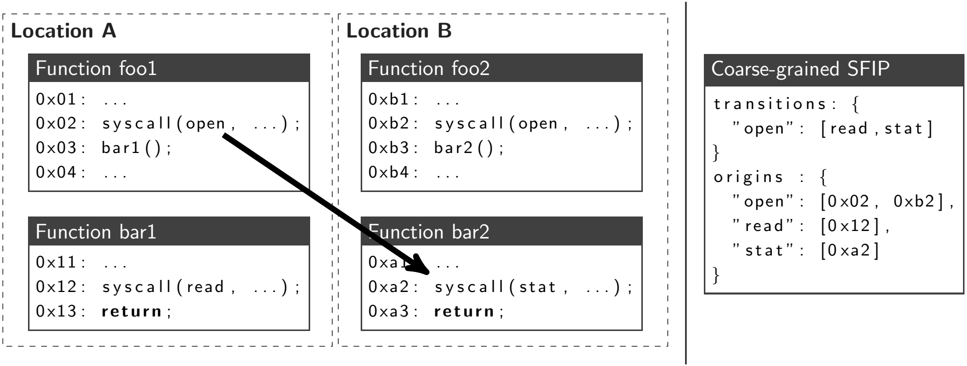
Function bar2

```
0xa1:  ...
0xa2:  syscall(stat, ...);
0xa3:  return;
```

SFIP

```
transitions: {
  "open": [read,stat]
}
origins : {
  "open": [0x02, 0xb2],
  "read": [0x12],
  "stat": [0xa2]
}
```

Coarse- to Fine-Grained SFIP



Location A

Function foo1

```
0x01: ...
0x02: syscall(open, ...);
0x03: bar1();
0x04: ...
```

Function bar1

```
0x11: ...
0x12: syscall(read, ...);
0x13: return;
```

Location B

Function foo2

```
0xb1: ...
0xb2: syscall(open, ...);
0xb3: bar2();
0xb4: ...
```

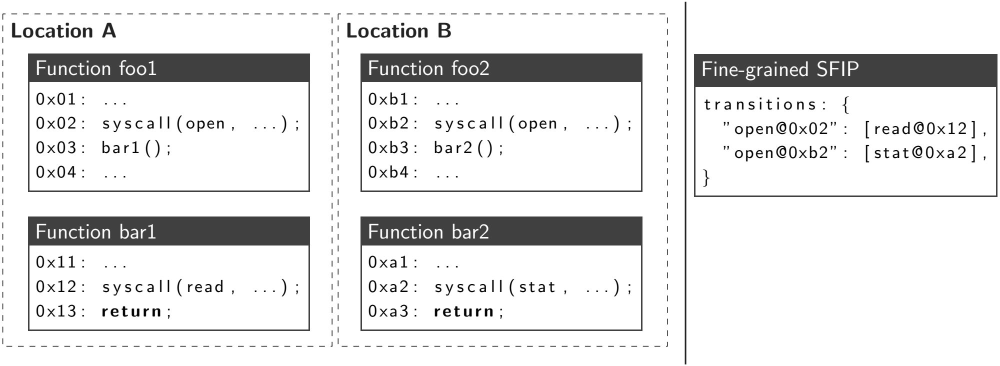
Function bar2

```
0xa1: ...
0xa2: syscall(stat, ...);
0xa3: return;
```

Coarse-grained SFIP

```
transitions: {
  "open": [read, stat]
}
origins : {
  "open": [0x02, 0xb2],
  "read": [0x12],
  "stat": [0xa2]
}
```

Coarse- to Fine-Grained SFIP



Location A

Function foo1

```
0x01:  ...
0x02:  syscall(open, ...);
0x03:  bar1();
0x04:  ...
```

Function bar1

```
0x11:  ...
0x12:  syscall(read, ...);
0x13:  return;
```

Location B

Function foo2

```
0xb1:  ...
0xb2:  syscall(open, ...);
0xb3:  bar2();
0xb4:  ...
```

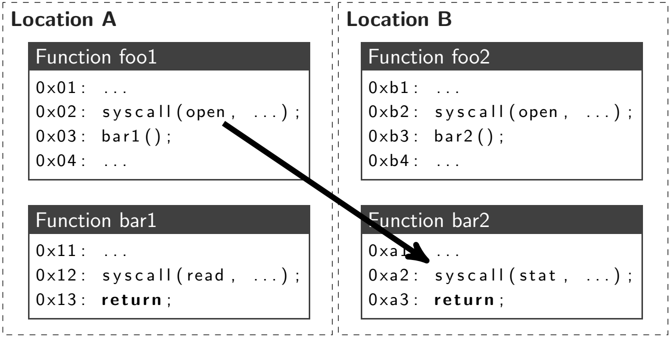
Function bar2

```
0xa1:  ...
0xa2:  syscall(stat, ...);
0xa3:  return;
```

Fine-grained SFIP

```
transitions: {
  "open@0x02": [read@0x12],
  "open@0xb2": [stat@0xa2],
}
```

Coarse- to Fine-Grained SFIP



Location A

Function foo1

```
0x01:  ...
0x02:  syscall(open, ...);
0x03:  bar1();
0x04:  ...
```

Function bar1

```
0x11:  ...
0x12:  syscall(read, ...);
0x13:  return;
```

Location B

Function foo2

```
0xb1:  ...
0xb2:  syscall(open, ...);
0xb3:  bar2();
0xb4:  ...
```

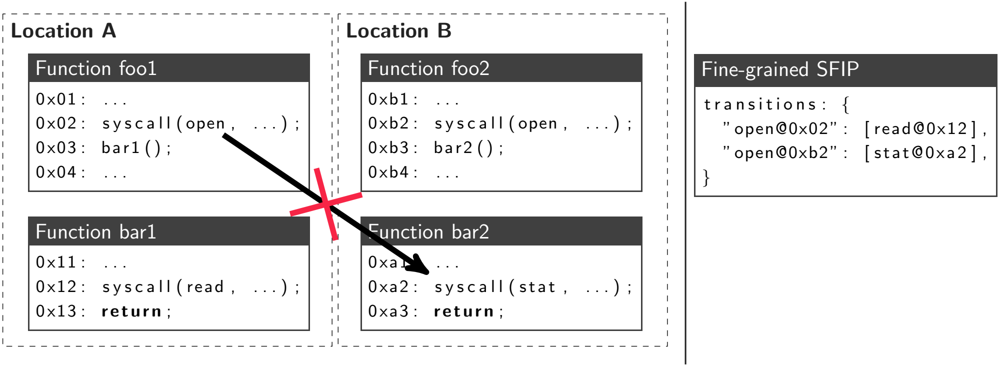
Function bar2

```
0xa1:  ...
0xa2:  syscall(stat, ...);
0xa3:  return;
```

Fine-grained SFIP

```
transitions: {
  "open@0x02": [read@0x12],
  "open@0xb2": [stat@0xa2],
}
```

Coarse- to Fine-Grained SFIP



Location A

Function foo1

```
0x01:  ...
0x02:  syscall(open, ...);
0x03:  bar1();
0x04:  ...
```

Function bar1

```
0x11:  ...
0x12:  syscall(read, ...);
0x13:  return;
```

Location B

Function foo2

```
0xb1:  ...
0xb2:  syscall(open, ...);
0xb3:  bar2();
0xb4:  ...
```

Function bar2

```
0xa1:  ...
0xa2:  syscall(stat, ...);
0xa3:  return;
```

Fine-grained SFIP

```
transitions: {
  "open@0x02": [read@0x12],
  "open@0xb2": [stat@0xa2],
}
```

Proof-of-Concept

You can find our proof-of-concept implementation of SysFlow on:

- `https://github.com/SFIP/SFIP`

More Details

More details in the paper

- More implementation details
- More extensive security discussion
- . . .

[Can+22]

Claudio Canella, Sebastian Dorn, Daniel Gruss, Michael Schwarz.
SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems.

Recap

SFIP provides

- integrity to user-kernel transitions

Recap

SFIP provides

- integrity to user-kernel transitions
- security via syscall transition and origin checks

Recap

SFIP provides

- integrity to user-kernel transitions
- security via syscall transition and origin checks

and

- is fully automatized

Recap

SFIP provides

- integrity to user-kernel transitions
- security via syscall transition and origin checks

and

- is fully automatized
- has minimal runtime overhead

Go With the Flow

Enforcing Program Behavior Through Syscall Sequences and Origins

Claudio Canella (🐦@cc0x1f)

August 11, 2022

Graz University of Technology

References

[Can+22] C. Canella, S. Dorn, D. Gruss, and M. Schwarz. SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems. In: arXiv:2202.13716 (2022).