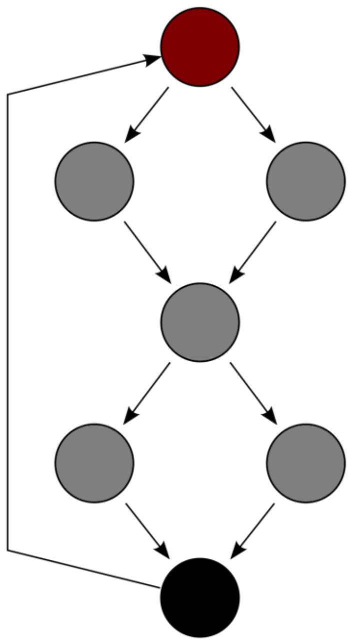
Efficient Protection of Path-Sensitive Control Security

Ren Ding, Chenxiong Qian, Chengyu Song*, Bill Harris, Taesoo Kim, Wenke Lee

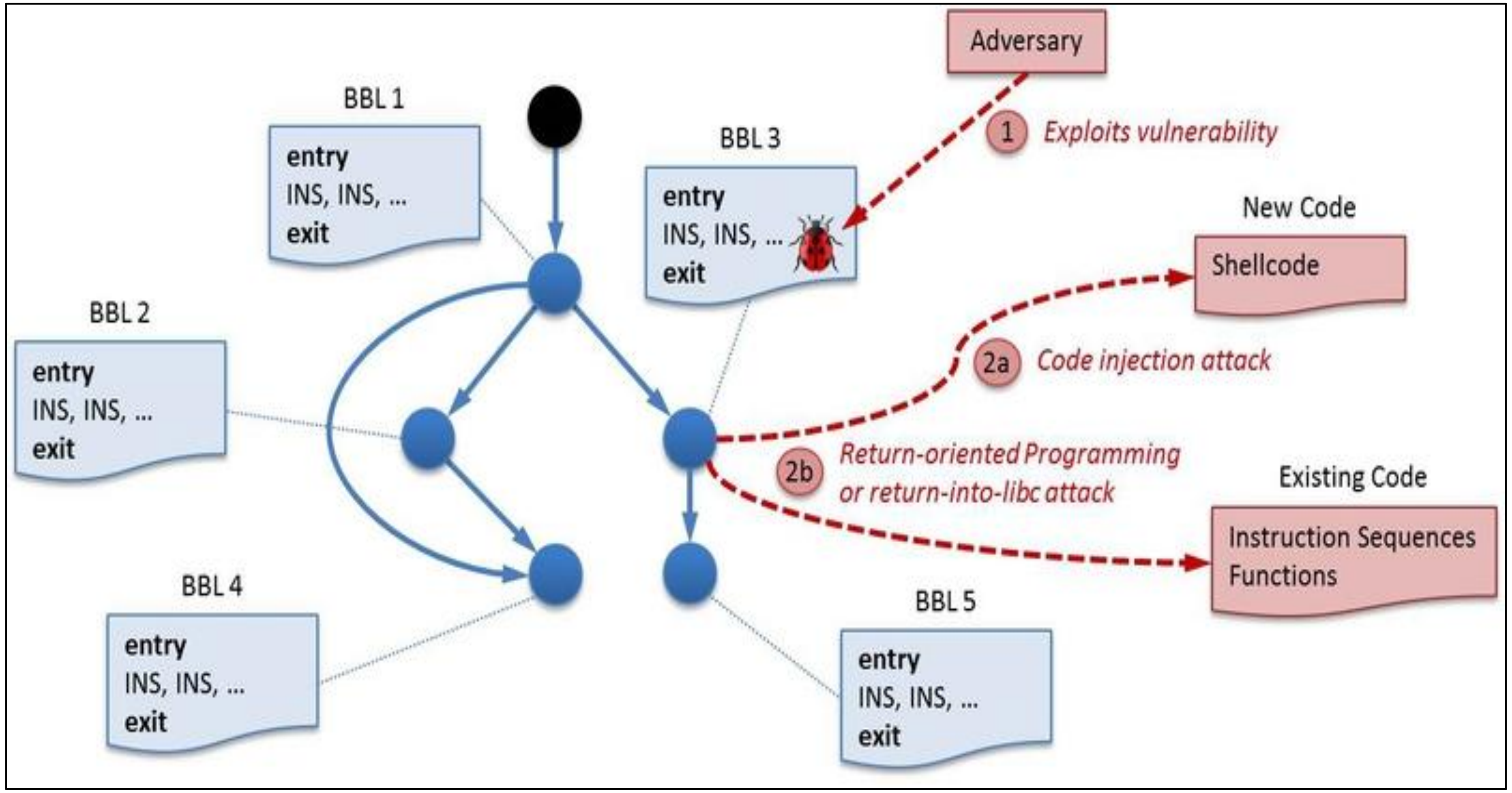
Georgia Tech, UC Riverside*

What is Control Flow?

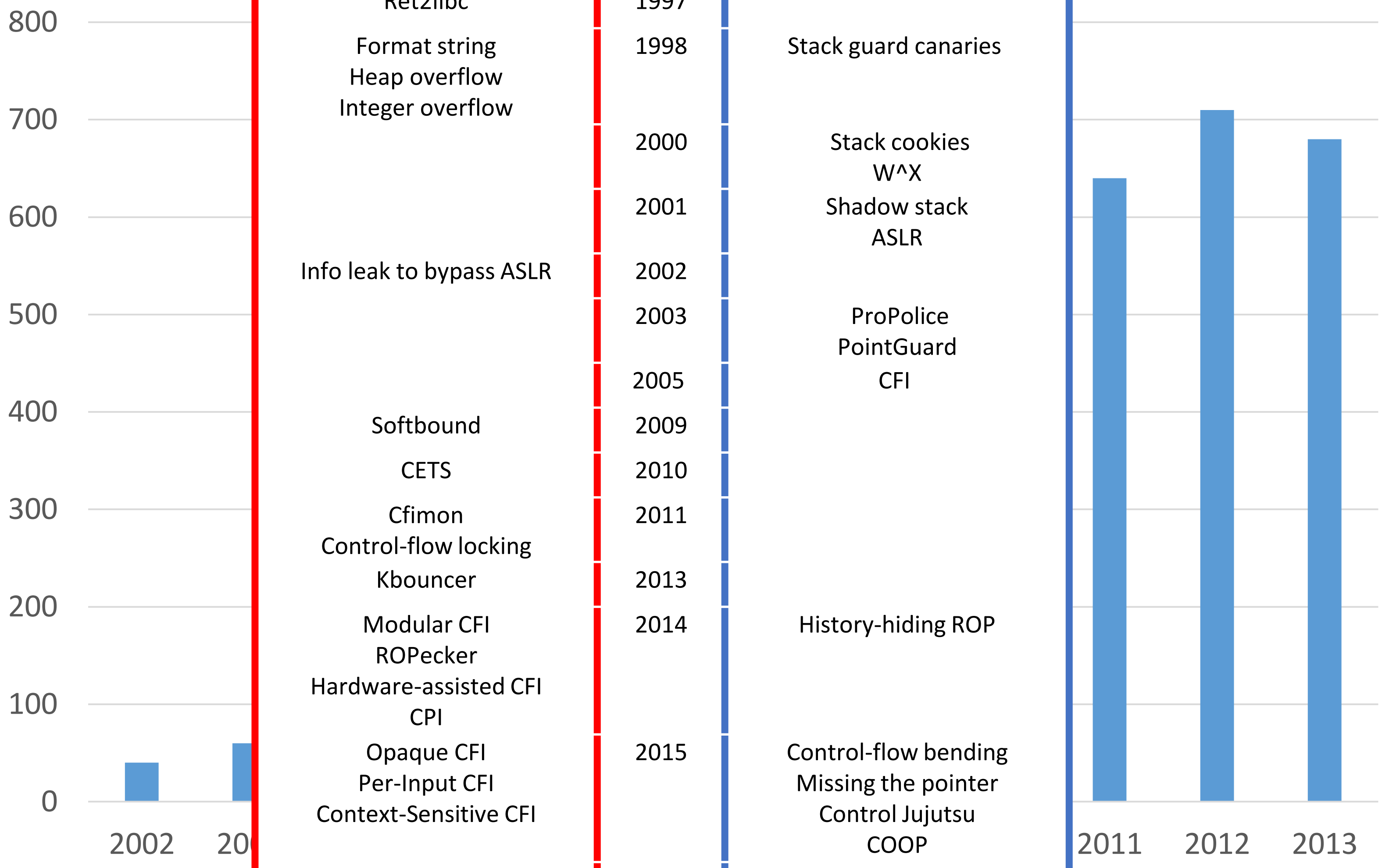
- The order of instruction execution
- Only limited sets of valid transitions



What is Control Hijacking?



Control Flow Attacks Still Exist...



ATTACK	Year	DEFENSE
Stack smashing	1996	
Ret2libc	1997	
Format string Heap overflow Integer overflow	1998	Stack guard canaries
	2000	Stack cookies W^X
	2001	Shadow stack ASLR
Info leak to bypass ASLR	2002	
	2003	ProPolice PointGuard
	2005	CFI
Softbound	2009	
CETS	2010	
Cfimon Control-flow locking	2011	
Kbouncer	2013	
Modular CFI ROPecker Hardware-assisted CFI CPI	2014	History-hiding ROP
Opaque CFI Per-Input CFI Context-Sensitive CFI	2015	Control-flow bending Missing the pointer Control Jujutsu COOP
Griffin FlowGuard	2017	

Control Flow Integrity (CFI)

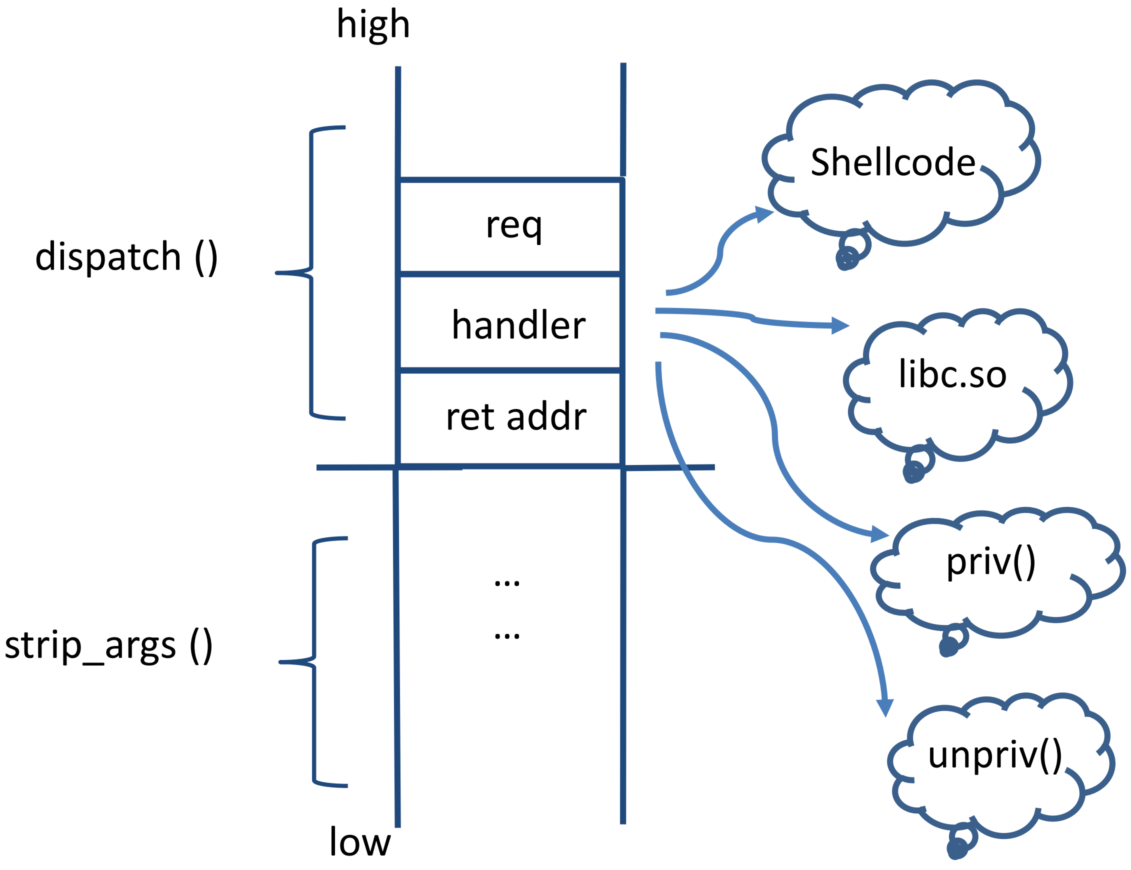
- Lightweight
- Runtime Enforcement
- Pre-computed valid sets: points-to analysis
- Limitations: over-approximation for soundness!

Motivating Example

- Parse request
- Assign “handler” fptr
  - If request from admin:
    - handler() = priv
  - else:
    - handler() = unpriv
- Strip request args
- Handle request

```
void dispatch() {
 void (*handler)(struct request *) = 0;
 struct request req;

 while(1) {
   parse_request(&req);

   if (req.auth_user == ADMIN) {
     handler = priv;
   } else {
     handler = unpriv;
     // NOTE: buffer overflow
     strip_args(req.args);
   }

   handler(&req);
 }
}
```

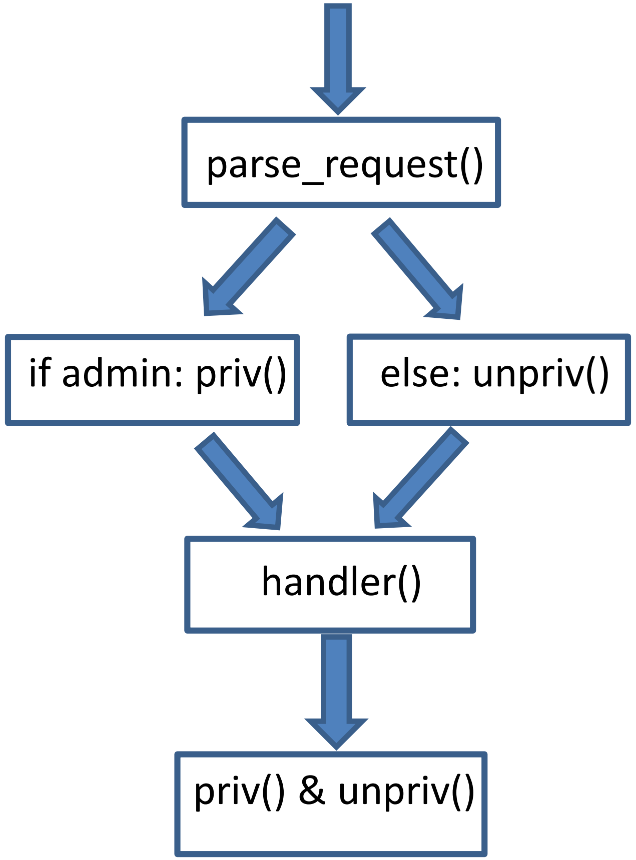
Motivating Example




```
void dispatch() {
 void (*handler)(struct request *) = 0;
 struct request req;

 while(1) {
   parse_request(&req);

   if (req.auth_user == ADMIN) {
     handler = priv;
   } else {
     handler = unpriv;
     // NOTE. buffer overflow
     strip_args(req.args);
   }

   handler(&req);
 }
}
```

Limitation of Traditional CFI

- Computes valid transfer sets at each location (lack dynamic info)



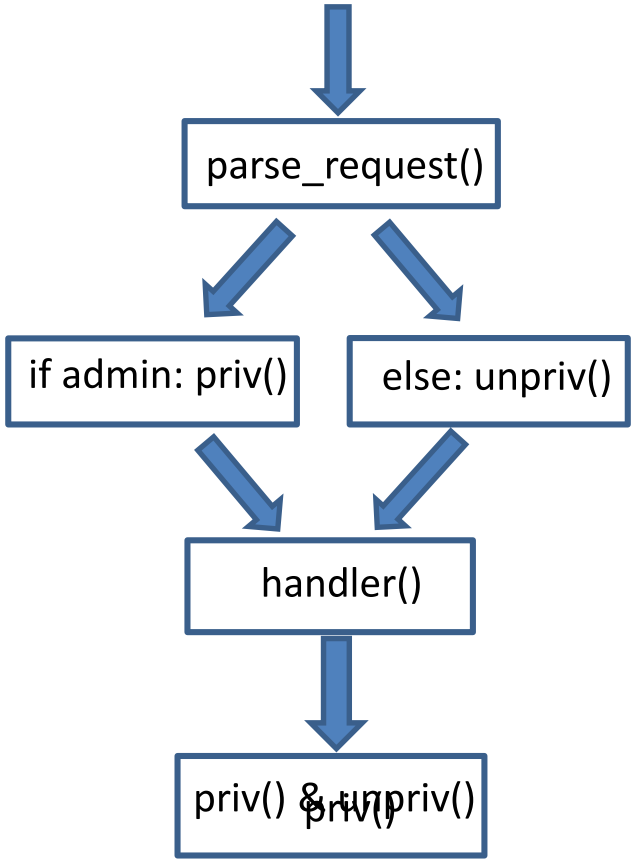

```
void dispatch() {
 void (*handler)(struct request *) = 0;
 struct request req;

 while(1) {
   parse_request(&req);

   if (req.auth_user == ADMIN) {
     handler = priv;
   } else {
     handler = unpriv;
     // NOTE. buffer overflow
     strip_args(req.args);
   }

   handler(&req);
 }
}
```

Per-Input CFI:
Most Precise Known CFI

- Relies on static analysis for soundness
- Instrumentation required
- Enable valid target based on execution history for addresses that are taken

Limitation of Per-Input CFI

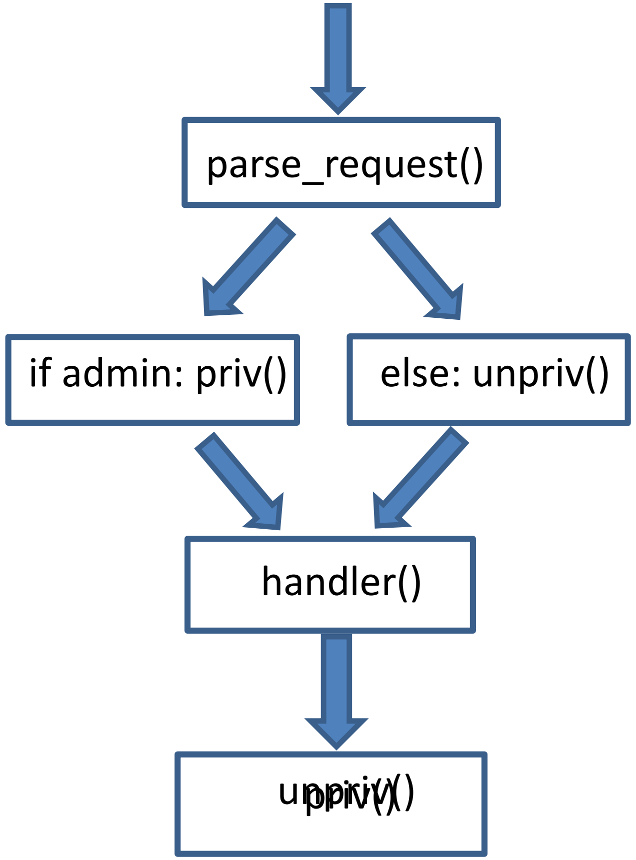
- Once transfer targets enabled, cannot be eliminated




```
void dispatch() {
 void (*handler)(struct request *) = 0;
 struct request req;

 while(1) {
   parse_request(&req);

   if (req.auth_user == ADMIN) {
     handler = priv;
   } else {
     handler = unpriv;
     // NOTE. buffer overflow
     strip_args(req.args);
   }

   handler(&req);
 }
}
```

PITTYPAT: Path-Sensitive CFI

- At each control transfer, verify based on points-to analysis of **whole execution path**




```
void dispatch() {
 void (*handler)(struct request *) = 0;
 struct request req;

 while(1) {
   parse_request(&req);

   if (req.auth_user == ADMIN) {
     handler = priv;
   } else {
     handler = unpriv;
     // NOTE. buffer overflow
     strip_args(req.args);
   }

   handler(&req);
 }
}
```

Assumptions

- Current approach only examines control security
- Non-control data is out of scope
- Not a memory safety solution

Challenges

- Collecting executed path information and share for analysis efficiently
- Trace information cannot be tampered
- Compute points-to relations online both efficiently and precisely

Our Solution Per Challenge

- Intel Processor Trace (PT)
- Incremental Online Points-to Analysis

Intel Processor Trace

- Low-overhead commodity hardware
- Compressed packets to save bandwidth
- CR3 filtering
- Trace information **shared & protected efficiently**

Incremental Points-to Analysis

- Input:
  - LLVM IR of target program
  - Metadata of mapping between IR and binary
  - Runtime execution trace

- Output: points-to relations on a **single execution path**

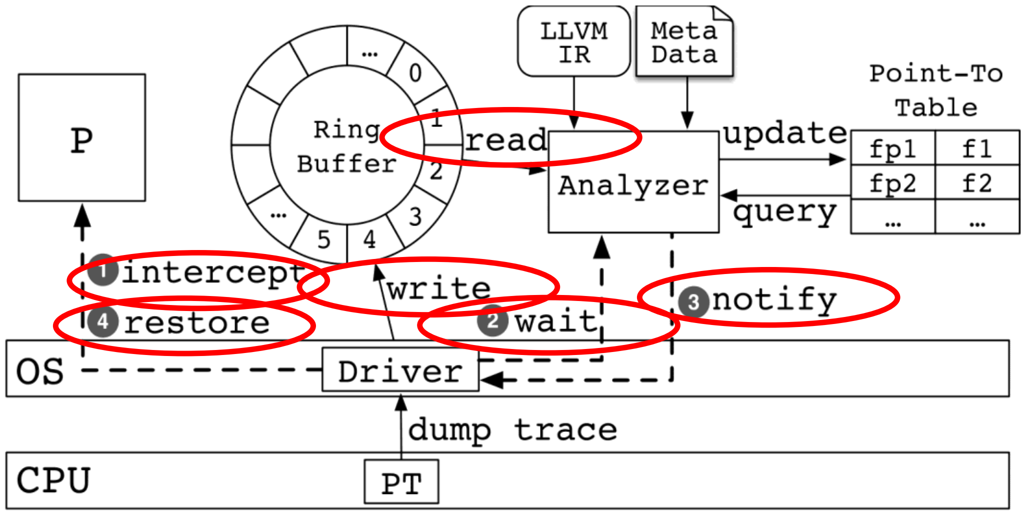
Things Differentiate Our Analysis

- Traditional static points-to analysis reasons about **all paths** for soundness
- Instead, we only reasons about points-to relation on **one single path**
- Maintain shadow **callstack** of instructions executed
- **Most precise enforcement** based on **control data** only

System Overview

- Monitor Module:
  - Kernel-space driver for PT
  - Shares taken branch information

- Analyzer Module:
  - User-space
  - Updates points-to relation based on trace



Challenging Language Features

- Signal handling
- Setjmp/Longjmp
- Exception Handling

Signal Handling

```
; Function Attrs: nounwind uwtable
define void @SIGKILL_handler(i32 %signo) #0 {
entry:
  ...
if.then:                                          ; preds = %entry
  ...
if.else:                                          ; preds = %entry
  ...
if.end:                                           ; preds = %if.else, %if.then
  ret void
}
; Function Attrs: nounwind uwtable
define i32 @main() #0 {
entry:
  %call1 = call void (i32)* @signal(i32 9, void (i32)* @SIGKILL_handler) #3
  ret i32 0
}
```

Setjmp/Longjmp

```
; Function Attrs: nounwind uwtable
define void @hello() #0 {
entry:
  ...
  call void @longjmp(%struct.__jmp_buf_tag* getelementptr inbounds ([1 x
%struct.__jmp_buf_tag], [1 x %struct.__jmp_buf_tag]* @resume_here, i32 0,
i32 0), i32 1) #4
  ...
}
; Function Attrs: nounwind uwtable
define i32 @main() #0 {
entry:
  ...
  %call1 = call i32 @_setjmp(%struct.__jmp_buf_tag* getelementptr inbounds
([1 x %struct.__jmp_buf_tag], [1 x %struct.__jmp_buf_tag]* @resume_here, i32
0, i32 0)) #5
  ...
```

Exception Handling

```
; Function Attrs: norecurse uwtable
define i32 @main() #4 personality i8* bitcast (i32
(...)* @__gxx_personality_v0 to i8*) {
entry:
  ...
  %call = invoke i32 @_Z3foov()
          to label %invoke.cont unwind label %lpad
invoke.cont:                                      ;
preds = %entry
  br label %try.cont
lpad:                                             ;
preds = %entry
  %0 = landingpad { i8*, i32 }
          catch i8* bitcast (i8** @_ZTIi to i8*)
          catch i8* bitcast (i8** @_ZTIc to i8*)
          catch i8* null
  ...
```

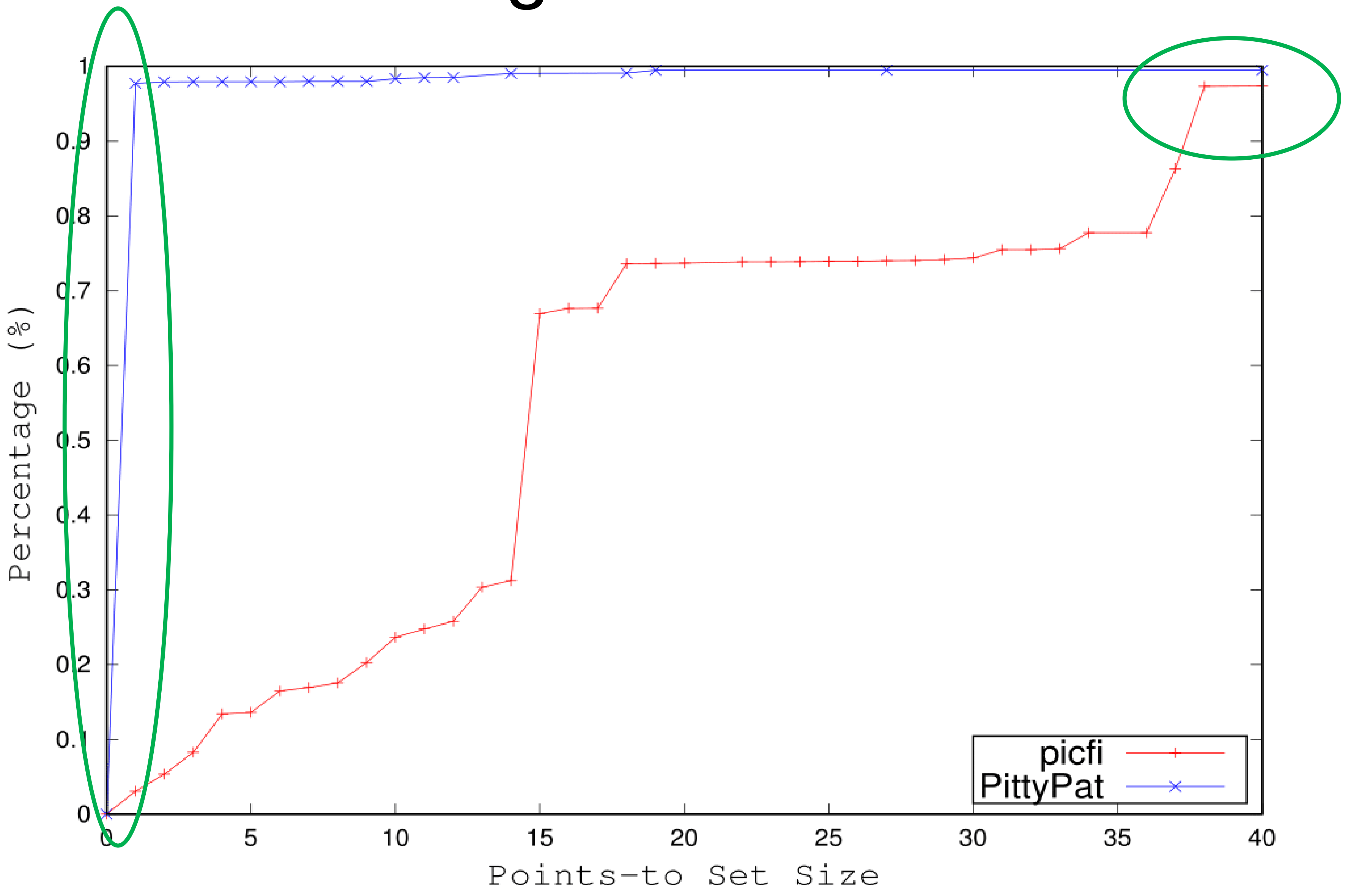
Optimizations on Analysis

- Only analyzing about calling context
- Maintains current executing IR block along with execution
  - To avoid decoding of PT traces and translation from binary address to IR
- Only analyze control-relevant functions and instructions

Evaluation

- Are benign applications satisfying path-sensitive CFI less susceptible to control hijacking attacks?
- Do malicious applications that satisfy weaker CFI mechanisms fail to satisfy current solution?
- Can we achieve path-sensitive CFI efficiently?

Forward Edge Points-to Set Size

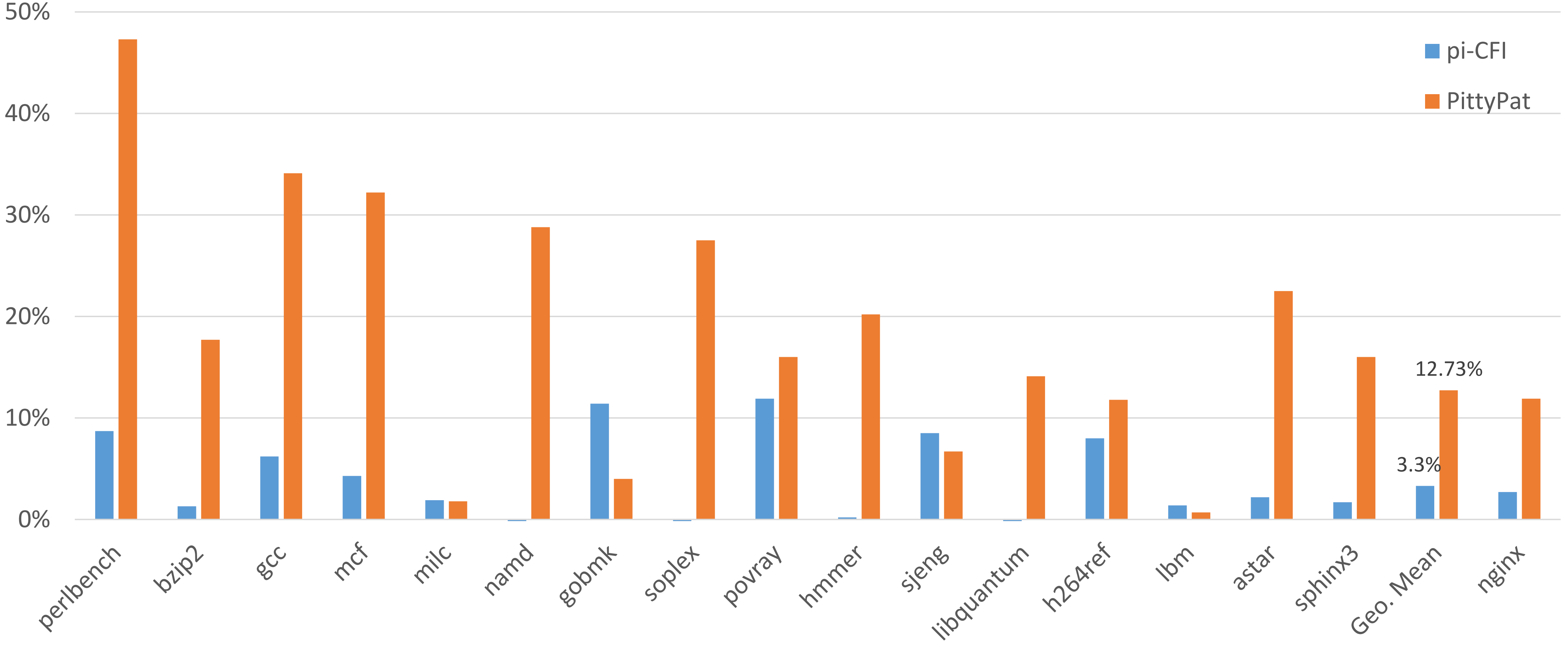


RIPE

- Contains various vulnerabilities that can be exploited to hijack control flow

- Passed all 264 benchmark suites that compiled in the testing environment

Performance Overhead



Limitations

- Non-control data corruption can not be detected
- Not reasoning about field sensitiveness for points-to analysis
- Performance might not be ideal as a CFI solution

Conclusion

- Define path-sensitive CFI
- Deploy practical mechanism for enforcement
- Strictly stronger security guarantees
- Acceptable runtime overhead in security critical settings