Enforcing **Unique** Code Target Property for Control-Flow Integrity

Hong Hu, Chenxiong Qian, Carter Yagmann,

Simon Pak Ho Chung, William R. Harris*,

Taesoo Kim, Wenke Lee

Georgia Tech

galois *

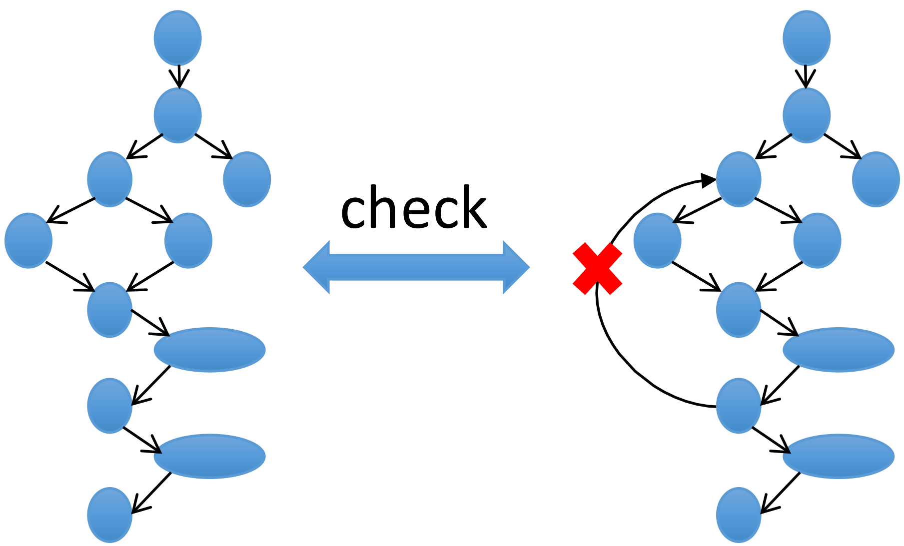
Control-flow attack

- Control-flow: the order of instruction execution

- Attackers use bugs to divert control flow
  - indirect control-flow transfer (ICT):
    - call *%rax, jmp *%rax, ret
  - func_ptr/ret_addr ==> &shellcode/&ROP_gadgets
  - the most common exploit method

Control-flow attack is getting harder

- Control-flow integrity (CFI)
  - Statically build control-flow graph (CFG)
  - Dynamically check with CFG



static CFG

run-time

Defenses

CCFIR

. . .

binCFI

. . .

MCFI

. . .

piCFI

. . .

TypeArmor

. . .

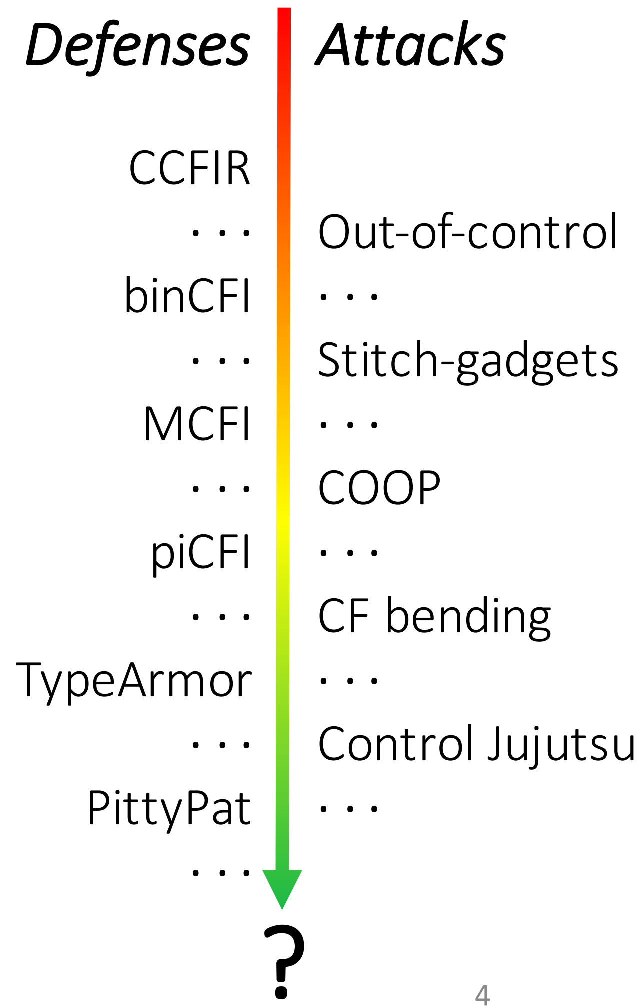
PittyPat

. . .

Control-flow attack is still possible

- Advanced attacks bypassing CFI
  - Out-of-control (oakland'14),
  - COOP (oakland'15),
  - Control-flow bending (usenix'15),
  - Code jujutsu (ccs'15)
- $|\text{allowed flow}| \gg |\text{real valid flow}|$
- The end of the story?
- $|\text{allowed flow}| = 1$

  <=> $\forall$ICT, $|\text{allowed target}| = 1$

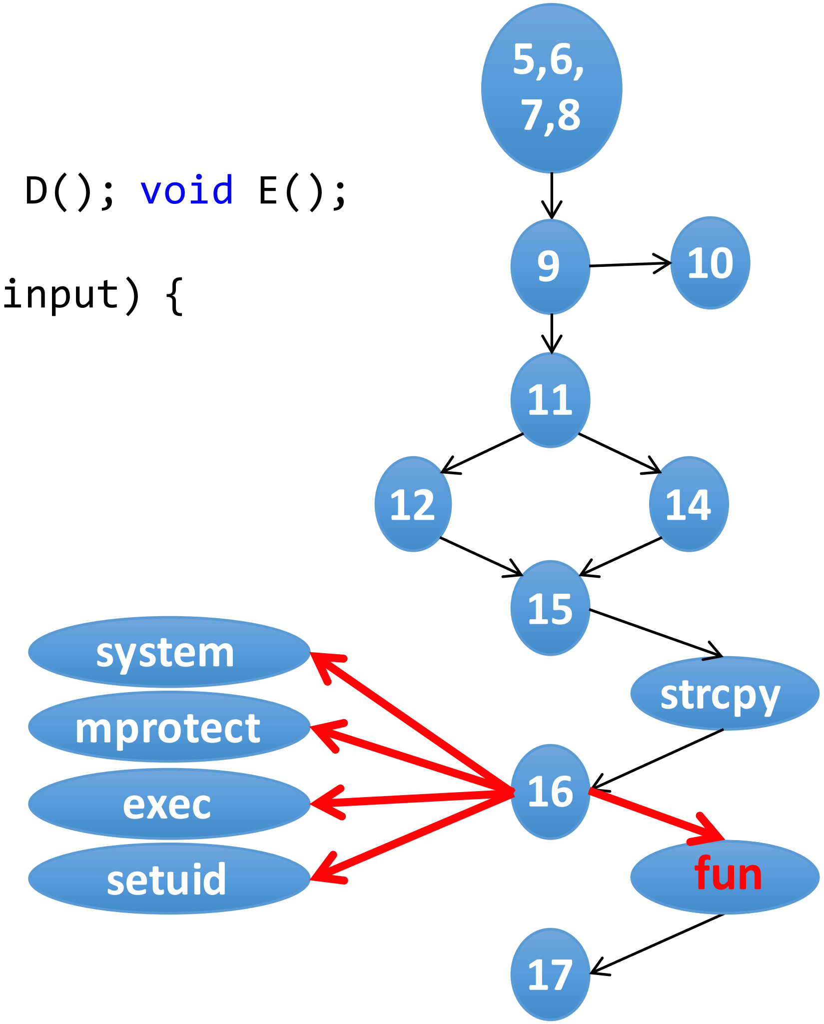


Our solution – uCFI

- Enforce unique code target property
  - only one valid target is allowed at runtime
- Efficient enforcement
  - 8% on SPEC CPU 2006
  - 4% on nginx
  - 1% on vsftpd

Example: control-flow attack

```
1  typedef void (*FP)();
2  void A(); void B(); void C(); void D(); void E();
3
4  void handleRequest(int id, char * input) {
5      FP arr[3] = {&A, &B, &C};
6      FP unused = &D;
7      FP fun    = NULL;
8      char buf[20];
9      if (id < 0 || id > 2)
10         return;
11     if (id == 0)
12         fun = arr[0];
13     else
14         fun = arr[id];
15     strcpy(buf, input);
16     (*fun)();    unknown before run
17 }
```



Example: control-flow integrity

- Identify valid target set *S* for each ICT
- For a run-time target ***t***: ***t*** ∈ *S* ? continue: abort
- Larger |*S*| => more attack

```
FP arr[3] = {&A, &B, &C};
FP unused = &D;
FP fun    = NULL;
if (id < 0 || id > 2)
   return;
if (id == 0)
   fun = arr[0];
else
   fun = arr[id];
(*fun)();
```

Method	**S** (id = 1)	**|S|**
no CFI	*	∞
Type-based CFI	A, B, C, D, E	5
Static CFI	A, B, C	3
piCFI	A, B, C, D	4
PittyPat	B, C	2
uCFI	**B**	**1**

Unique code target property

- UCT property:
  - for each invocation of an ICT,
  - ***one*** and ***only one*** allowed target

- Enforcement:
  - collect *necessary* runtime info to infer the unique target

```
FP arr[3] = {&A, &B, &C};
FP fun    = NULL;
char buf[20];
if (id < 0 || id > 2)
   return;
if (id == 0)
   fun = arr[0];
else
   fun = arr[id];
strcpy(buf, input);
(*fun)();
```

- PittyPat[1] uses the same methodology,
- but ***fails*** to enforce UCT property

Challenges with Intel PT

- Intel PT only delivers control-data
  - TNT: branch taken / non-taken
  - TIP: ICT target
- <u>C1: *unique* target</u>
  - line 14: id = 1 or 2 ? |***S***| = 2
  - |***S***| = 479 for gobmk
- <u>C2: *efficient* analysis</u>
  - path reconstruction from PT trace is slow!
  - 30x slow down for sjeng
    - (based on our simple implementation)

```
5    FP arr[3] = {&A, &B, &C};
7    FP fun    = NULL;
8    char buf[20];
9    if (id < 0 || id > 2)
10       return;
11   if (id == 0)
12       fun = arr[0];
13   else
14       fun = arr[id];
15   strcpy(buf, input);
16   (*fun)();
```

uCFI – enforce unique target

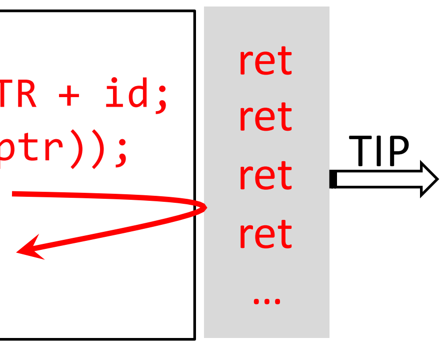
- Encode non-control data in some ICT

```
fun = arr[id];



strcpy(buf, input);
(*fun)();
```

uCFI – enforce unique target

- Encode non-control data in some ICT

```
fun = arr[id];
FP new_ptr = BASE_PTR + id;
assert(inBound(new_ptr));
(*new_ptr)();
strcpy(buf, input);
(*fun)();
```



uCFI – enforce unique target

- Encode non-control data in some ICT

```
fun = arr[id];
FP new_ptr = BASE_PTR + id;
assert(inBound(new_ptr));
(*new_ptr)();
strcpy(buf, input);
(*fun)();
```

ret
ret
ret
ret
…

TIP

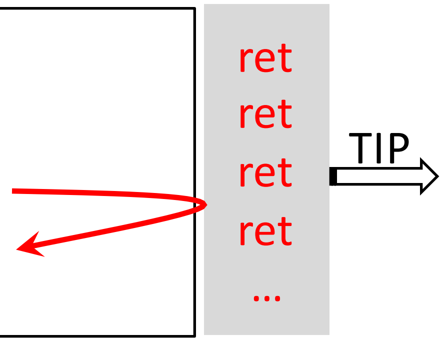
```
int read_data() {
    int packet = getPTPacket();
    int id = packet – BASE_PTR;
    return id;
}
```

- Restore non-control data in monitor process

uCFI – enforce unique target

- Encode non-control data in some ICT

```
fun = arr[id];

write_data(id);

strcpy(buf, input);
(*fun)();
```




```
int read_data() {
    int packet = getPTPacket();
    int id = packet – BASE_PTR;
    return id;
}
```

- Restore non-control data in monitor process
- write_data(x):
  - log arbitrary non-control-data into PT trace
  - enable analysis for <u>*unique*</u> target
  - current setting: 4M ret instrs ==> [-1024, 4M-1024]

Which data is necessary?

Constraining data: non-control-data affecting control-flow

1. *Control-data*: (similar to CPI[5])
   - a code pointer / a pointer of a known control-data
   - recursive data-flow analysis
2. *Control-instruction*:
   - Instructions operating on control-data
3. *Constraining-data*:
   - non-control data used in control-instructions
   - like, array index, condition in cmov

uCFI – perform efficient analysis

path reconstruction from PT trace is slow!

- Avoid (most) path reconstruction

```
FP arr[3] = {&A, &B, &C};

FP fun    = NULL;
char buf[20];
if (id < 0 || id > 2)
   return;
if (id == 0)  {

   fun = arr[0];
} else {

   fun = arr[id];
}
strcpy(buf, input);

(*fun)();
```

uCFI – perform efficient analysis

path reconstruction from PT trace is slow!

- Avoid (most) path reconstruction
  - assign an ID to each control-instruction
  - write_data(ID) into PT trace

```
write_data(ID1);
FP arr[3] = {&A, &B, &C};
write_data(ID2);
FP fun    = NULL;
char buf[20];
if (id < 0 || id > 2)
   return;
if (id == 0)  {
   write_data(ID3);
   fun = arr[0];
} else {
   write_data(ID4);
   fun = arr[id];
}
strcpy(buf, input);
write_data(ID5);
(*fun)();
```

uCFI – perform efficient analysis

path reconstruction from PT trace is slow!

- Avoid (most) path reconstruction
  - assign an ID to each control-instruction
  - write_data(ID) into PT trace
  - Ignore all TNT packets
- Analysis

```
while(ID = decode_data())
   switch(ID)
      case ID1: pts[arr+0] = A;   pts[arr+1] = B;
                pts[arr+2] = C;             break;
      case ID2: pts[fun]   = NULL;          break;
      case ID3: pts[fun]   = pts[arr+0];    break;
      case ID4: id         = decode_data();
                pts[fun]   = pts[arr+id];   break;
      case ID5: if(pts[fun] != PT_packet)
                    abort();
```

```
write_data(ID1);
FP arr[3] = {&A, &B, &C};
write_data(ID2);
FP fun    = NULL;
char buf[20];
if (id < 0 || id > 2)
   return;
if (id == 0)  {
   write_data(ID3);
   fun = arr[0];
} else {
   write_data(ID4);
   fun = arr[id];
}
strcpy(buf, input);
write_data(ID5);
(*fun)();
```

uCFI – perform efficient analysis

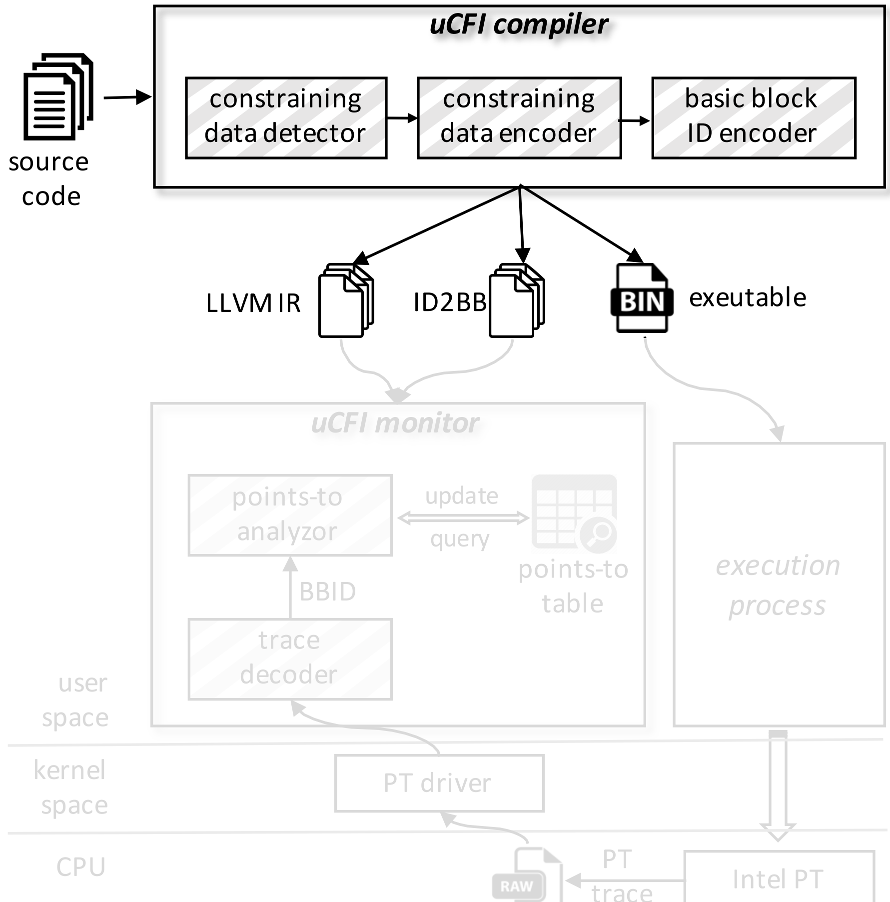
path reconstruction from PT trace is slow!

- Avoid (most) path reconstruction
  - assign an ID to each ~~control instruction~~ basic block w/ some control-instrs
  - Ignore all TNT packets
- Analysis ***efficiently***

```
while(ID = decode_data())
   switch(ID)
      case ID1: pts[arr+0] = A;   pts[arr+1] = B;
                pts[arr+2] = C;           break;
      case ID2: pts[fun]   = NULL;        break;
      case ID3: pts[fun]   = pts[arr+0];  break;
      case ID4: id         = decode_data();
                pts[fun]   = pts[arr+id]; break;
      case ID5: if(pts[fun] != PT_packet)
                    abort();
```

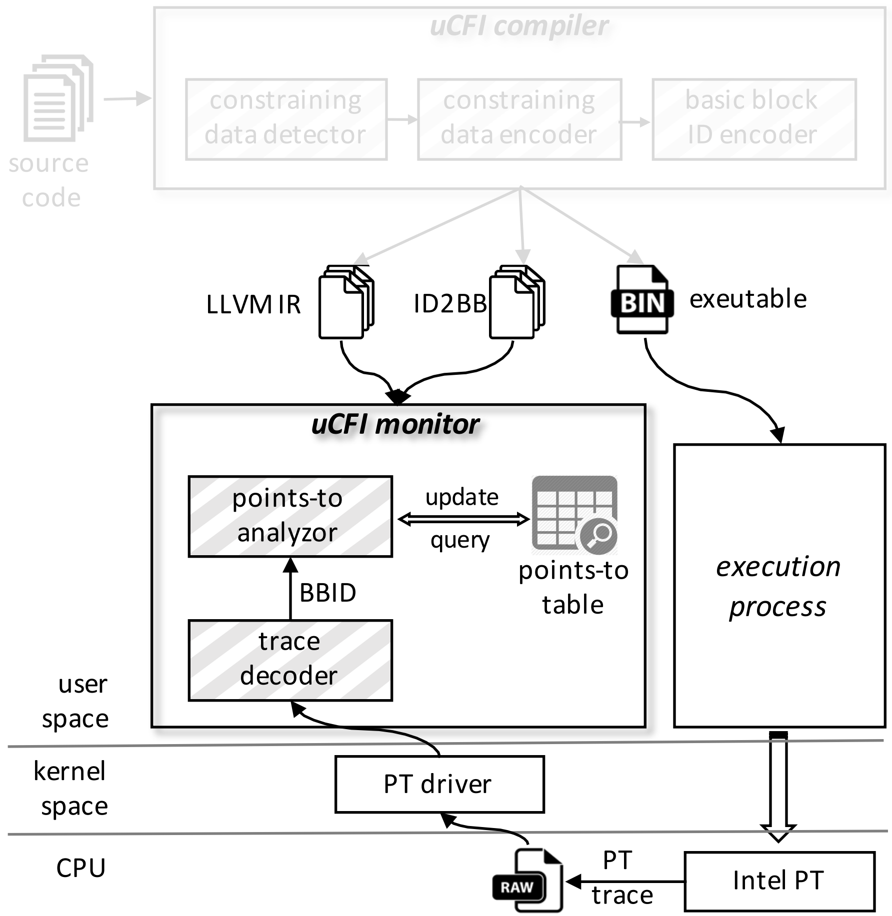
```
write_data(ID1);
FP arr[3] = {&A, &B, &C};
write_data(ID2);
FP fun    = NULL;
char buf[20];
if (id < 0 || id > 2)
   return;
if (id == 0)  {
   write_data(ID3);
   fun = arr[0];
} else {
   write_data(ID4);
   fun = arr[id];
}
strcpy(buf, input);
write_data(ID5);
(*fun)();
```

uCFI overview



- uCFI compiler
  - identify constraining data
  - encode constraining data
  - encode basic block ID

uCFI overview



- uCFI monitor
  - decode basic block ID
  - decode constraining data
  - perform points-to analysis
  - perform CFI check
  - **sync** with execution on critical system calls

Implementation

- x86_64 system
- uCFI compiler (1,652 SLOC) – based on LLVM 3.6
- uCFI monitor (4,310 SLOC)
- PT driver – based on Griffin[2] code

- IP filtering
  - 1 return instruction
  - 1 indirect call instruction

Evaluation – set up

- Benchmark
  - SPEC CPU 2006 (-O2)
  - nginx & vsftpd (default compilation script)

- Environment:
  - 8-core Intel i7-7740X CPU (4.30GHz), 32GB RAM
  - 64-bit Ubuntu 16.04 system

Security – enforcing unique target

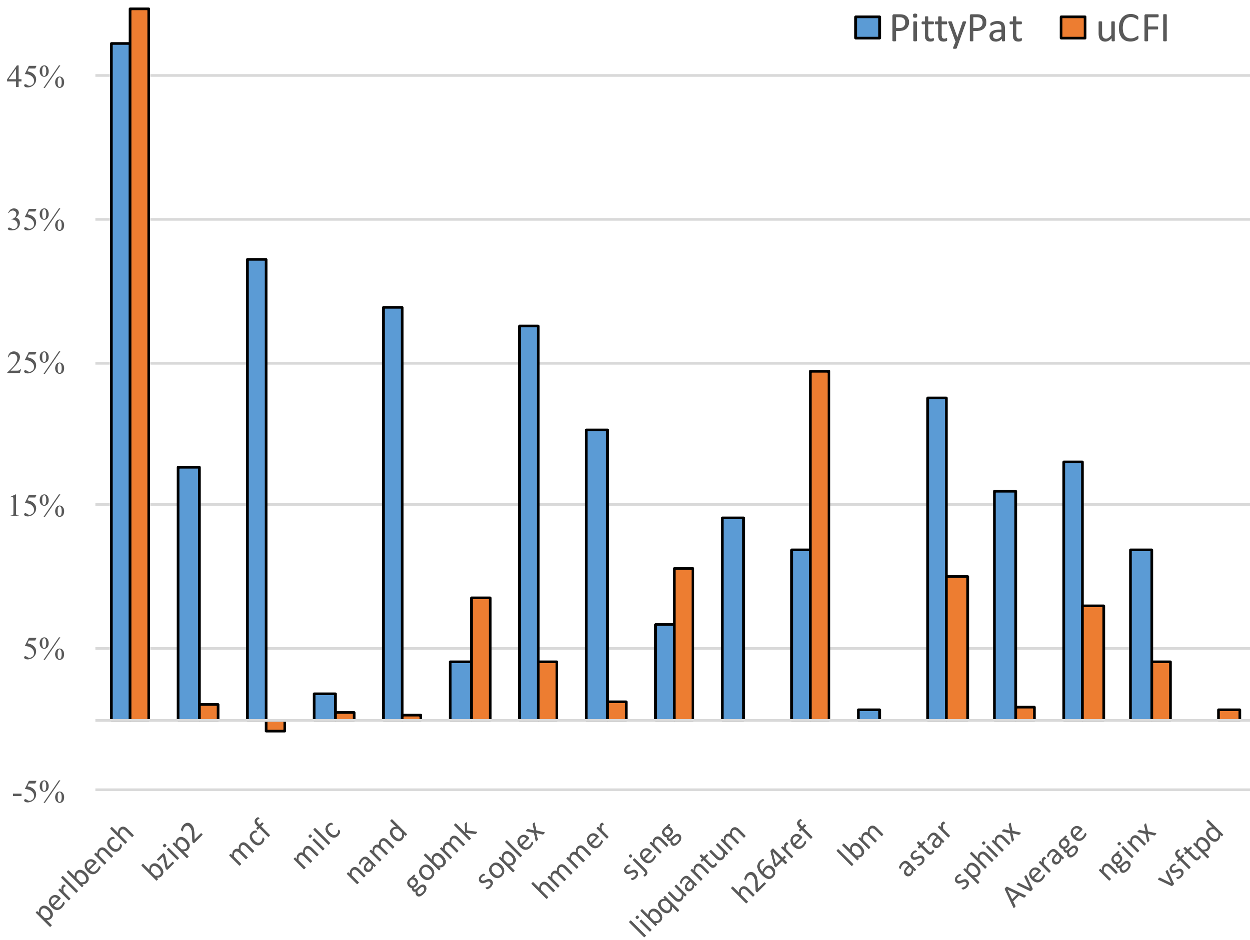
- Successfully enforce 1 target for tested programs
  - gobmk: 479/1, sjeng: 7/1, h264ref: 10/1

```
typedef int (*EVALFUNC)(int sq, int c);
static EVALFUNC evalRoutines[7] = {
    ErrorIt, Pawn, Knight, King, Rook, Queen, Bishop
};
int std_eval (int alpha, int beta) { ...
    for (j = 1, a = 1; (a <= piece_count); j++) {
        i = pieces [j]; ...
        score += (*(evalRoutines[piecet(i)]))(i,pieceside(i));
    }
}
```

Security – preventing attacks

Prog	Source	Type	Exploit	PiCFI	PittyPat	uCFI
ffmpeg	CVE-2016-10191	Heap overflow	Code pointer	✔	✔	✔
	CVE-2016-10190	Heap overflow	Code pointer	✔	✔	✔
php	CVE-2015-8617	Format string	Code pointer	✔	✔	✔
nginx	CVE-2013-2028	Stack overflow	Pointer of code pointer	✔	✔	✔
sudo	CVE-2012-0809	Format string	Code pointer	✔	✔	✔
COOP PoC	PittyPat	Stack overflow	Pointer of C++ object		✔	✔
sjeng	synthesized	-	Code pointer			✔
gobmk	synthesized	-	Code pointer			✔

Efficiency – performance overhead



protected exec

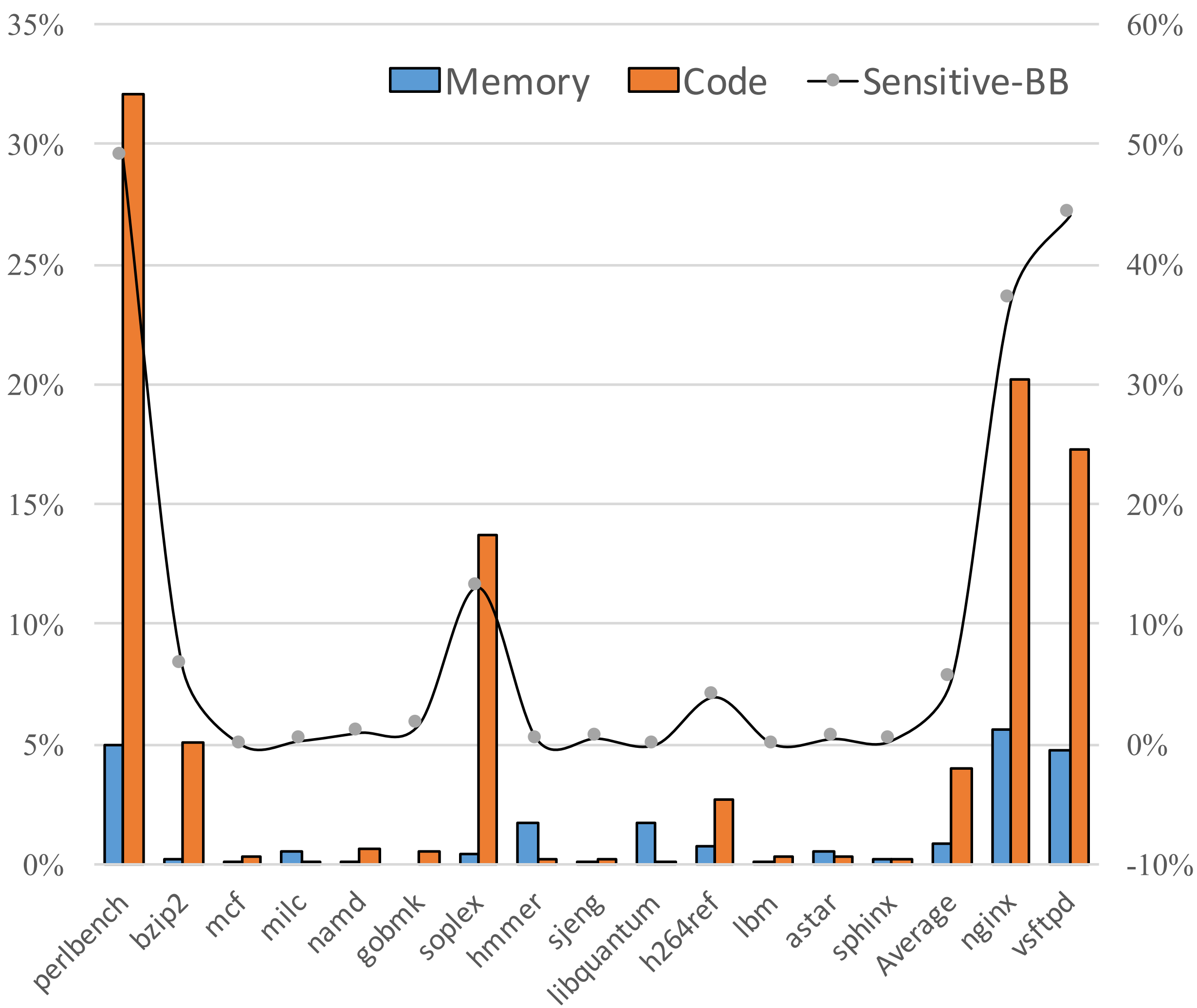
vs.

original exec

- 7.9% for SPEC
- 4.1% for nginx
- 0.8% for vsftpd

- perlbench
  - multiple process creation

Efficiency – memory&code overhead



Memory overhead

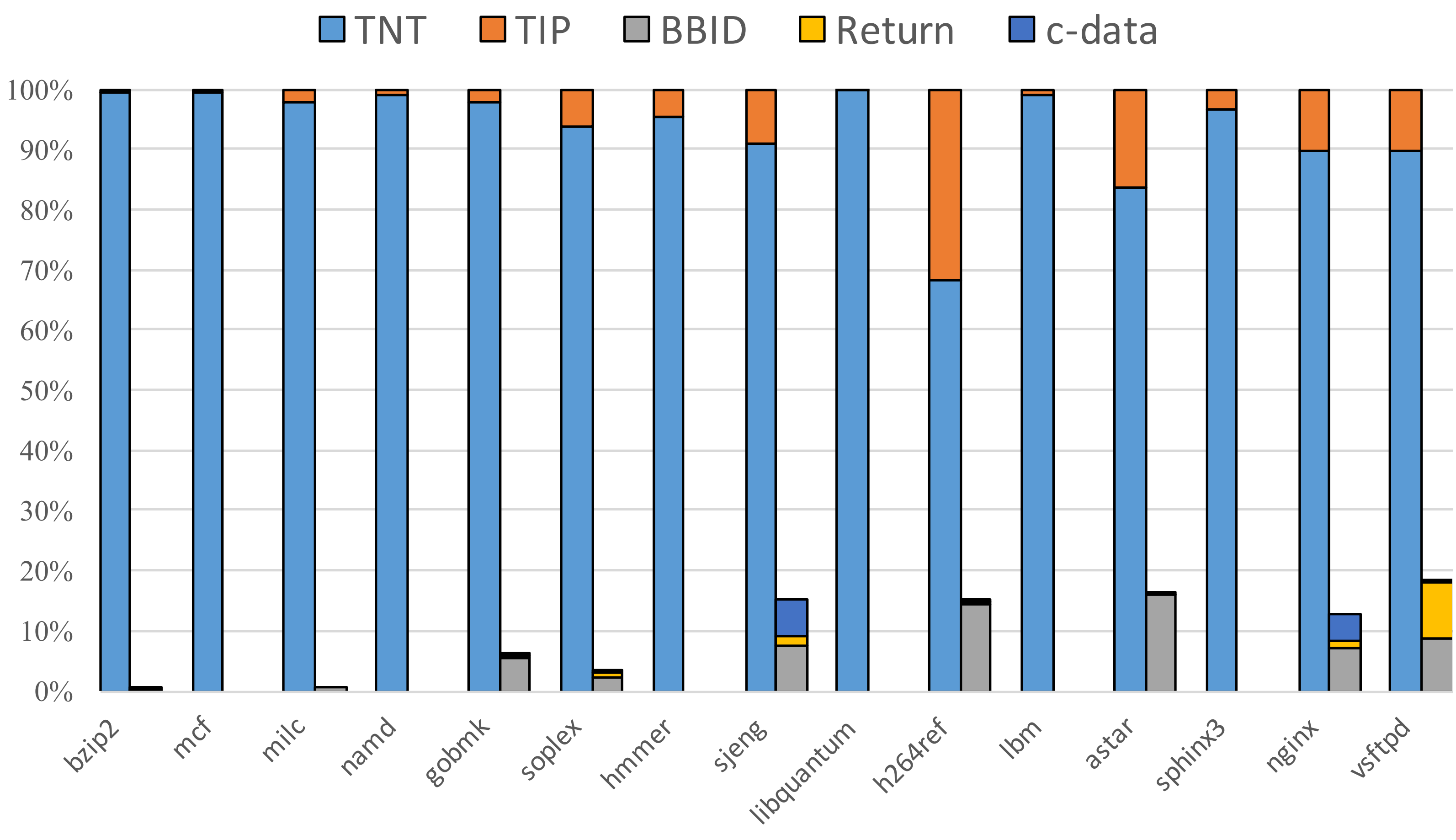
- 0.8% for SPEC
- 5.6% for nginx
- 4.8% for vsftpd

Code size overhead

- 4.0% for SPEC
- 20.3% for nginx
- 17.3% for vsftpd

Strongly related to sensitive-BB%

Efficiency – trace size reduction



Discussion – backward-edge CFI

- uCFI does not protect return address
- Integration with parallel shadow stack[7]
  - For compatibility checking only
  - 58 SLOC code in LLVM X86 backend
  - 2.07% extra overhead (SPEC), <1% overhead (nginx & vsftpd)
- Alternatives:
  - SafeStack (available in clang)
  - Intel CET (in the future)

Conclusion: uCFI

Security:

- Enforce Unique Code Target Property

Efficiency:

- (HW) Intel PT for control data
- (SW) write_data for non-control data

Open source:
https://github.com/uCFI-GATech

Related Work

1. **Efficient Protection of Path-Sensitive Control Security.** Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. USENIX 2017.
2. **GRIFFIN: Guarding Control Flows Using Intel Processor Trace.** Xinyang Ge, Weidong Cui, and Trent Jaeger. ASPLOS 2017.
3. **Per-Input Control-Flow Integrity**. Ben Niu and Gang Tan. CCS 2015.
4. **Practical Context-Sensitive CFI.** Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. CCS 2015.
5. **Code-Pointer Integrity**. Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. OSDI 2014.
6. **Practical Control Flow Integrity and Randomization for Binary Executables.** Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Oakland 2013.
7. **The Performance Cost of Shadow Stacks and Stack Canaries.** Thurston H.Y. Dang, Petros Maniatis, and David Wagner. AsiaCCS 2015.
8. **Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications.** Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Oakland 2015.
9. **Control-Flow Bending: On the Effectiveness of Control-Flow Integrity.** Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. USENIX Security 2015.

Discussion – difference from CPI

	Platform	Protect Stage	Blocked Bugs	Isolation	Safe?
CPI	x86	prevention	spatial	process	✔
	x86_64			information hiding	✗
uCFI	x86_64	detection	spatial & temporal	process	✔