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](http://www.gatech.edu) | [galois](http://www.galois.com)
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

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?
- \(|\text{allowed flow}| = 1\)
- \(\iff \forall ICT, |\text{allowed target}| = 1\)

**Defenses**

- CCFIR
- binCFI
- MCFI
- piCFI
- TypeArmor
- PittyPat

**Attacks**

- Out-of-control
- Stitch-gadgets
- COOP
- CF bending
- Control Jujutsu
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
typedef void (*FP)();
void A(); void B(); void C(); void D(); void E();

void handleRequest(int id, char * input) {
    FP unused = &D;
    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)();
}
Example: control-flow integrity

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

```
Method       | S (id = 1) | |S|
-------------|------------|---------
nocFI         | *          | $\infty$
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
```

6 FP unused = &D;
7 FP fun = NULL;
9 if (id < 0 || id > 2) return;
11 if (id == 0)
12     fun = arr[0];
13    else
14     fun = arr[id];
16 (*fun)();

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

• PittyPat\textsuperscript{[1]} uses the same methodology,
  • but **fails** to enforce UCT property

```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)();
```
Challenges with Intel PT

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

```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

```c
fun = arr[id];

strcpy(buf, input);
(*fun)();
```
uCFI – enforce unique target

• Encode non-control data in some ICT

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

TIP
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)();
  ```

- Restore non-control data in monitor process
  ```
  int read_data() {
    int packet = getPTPacket();
    int id = packet - BASE_PTR;
    return id;
  }
  ```
uCFI – enforce unique target

• Encode non-control data in some ICT

```c
fun = arr[id];
write_data(id);
strcpy(buf, input);
(*fun)();
```

• Restore non-control data in monitor process

• `write_data(x)`:
  • log arbitrary non-control-data into PT trace
  • enable analysis for **unique** 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\textsuperscript{[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

```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

```c
write_data(ID1);
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

```c
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();
```

```c
write_data(ID1);
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*

```c
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);
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

Source code

**uCFI compiler**
- constraining data detector
- constraining data encoder
- basic block ID encoder

LLVM IR

ID2BB

BIN

Executable

**uCFI monitor**
- points-to analyzor
  - BBID
- trace decoder
  - update query points-to table

**Execution process**

User space

Kernel space

CPU

PT driver

PT trace

Intel PT
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\cite{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

```c
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

<table>
<thead>
<tr>
<th>Prog</th>
<th>Source</th>
<th>Type</th>
<th>Exploit</th>
<th>PiCFI</th>
<th>PittyPat</th>
<th>uCFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffmpeg</td>
<td>CVE-2016-10191</td>
<td>Heap overflow</td>
<td>Code pointer</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td>CVE-2016-10190</td>
<td>Heap overflow</td>
<td>Code pointer</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>php</td>
<td>CVE-2015-8617</td>
<td>Format string</td>
<td>Code pointer</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>nginx</td>
<td>CVE-2013-2028</td>
<td>Stack overflow</td>
<td>Pointer of code pointer</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>sudo</td>
<td>CVE-2012-0809</td>
<td>Format string</td>
<td>Code pointer</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>COOP PoC</td>
<td>PittyPat</td>
<td>Stack overflow</td>
<td>Pointer of C++ object</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>sjeng</td>
<td>synthesized</td>
<td>-</td>
<td>Code pointer</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>gobmk</td>
<td>synthesized</td>
<td>-</td>
<td>Code pointer</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>
Efficiency – performance overhead

- 5% for SPEC
- 4.1% for nginx
- 0.8% for vsftpd

- perlbench
  - multiple process creation

protected exec vs. original exec
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

![Efficiency Graph](image-url)
Discussion – backward-edge CFI

• uCFI does not protect return address
• Integration with parallel shadow stack\cite{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.


## Discussion – difference from CPI

<table>
<thead>
<tr>
<th>Platform</th>
<th>Protect Stage</th>
<th>Blocked Bugs</th>
<th>Isolation</th>
<th>Safe?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPI</td>
<td>x86</td>
<td>prevention</td>
<td>spatial</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td>x86_64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>detection</td>
<td>spatial &amp; temporal</td>
<td>process</td>
<td>✔</td>
</tr>
<tr>
<td>uCFI</td>
<td>x86_64</td>
<td>detection</td>
<td>information hiding</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>