Poster: Live Path Control Flow Integrity

Mohamad Barbar, Yulei Sui
University of Technology Sydney

Hongyu Zhang
University of Newcastle

Shiping Chen
CSIRO/Data61

Jingling Xue
University of New South Wales

ABSTRACT

Per-Input Control Flow Integrity (PICFI) represents a recent advance in dynamic CFI techniques. PICFI starts with the empty CFG of a program and lazily adds edges to the CFG during execution according to concrete inputs. However, this CFG grows monotonically, i.e., invalid edges are never removed when corresponding control flow transfers (via indirect calls) become illegal (i.e., will never be executed again). This paper presents LPCFI, Live Path Control Flow Integrity, to more precisely enforce forward edge CFI using a dynamically computed CFG by both adding and removing edges for all indirect control flow transfers from function pointer calls, thereby raising the bar against control flow hijacking attacks.

CCS CONCEPTS

• **Security and privacy** → **Information flow control**;

KEYWORDS

Control Flow Integrity, Live Path, Hijacking Attacks

ACM Reference Format:

Mohamad Barbar, Yulei Sui, Hongyu Zhang, Shiping Chen, and Jingling Xue. 2018. Poster: Live Path Control Flow Integrity. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 2 pages. https://doi.org/10.1145/3183440.3195093

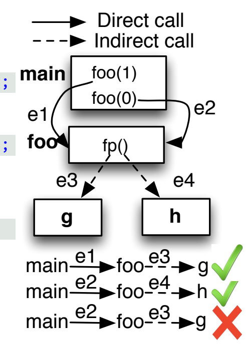
1 INTRODUCTION

Programs written in low-level languages, such as C and C++, make up the majority of performance-critical system software (e.g., web browsers and language runtimes) running on most computing platforms. However, these unsafe languages are prone to memory corruption vulnerabilities (e.g., use-after-free). An attacker may leverage these vulnerabilities to launch control flow hijacking attacks by changing the target of an indirect branch instruction to force a running program to execute at a location of the attacker's choice.

Existing Control Flow Integrity (CFI) techniques [1] aim to mitigate these adversarial effects by restricting a program's execution to its statically over-approximated control flow graph (CFG). PICFI [4] represents a recent dynamic approach to forward edge CFI. PICFI first pre-computes a static CFG as the upper bound for its dynamic one. PICFI starts with the empty CFG of a program. During runtime, only when a function address is taken (e.g., `p = &func`), it will add an edge from each indirect callsite to `func` if this edge is also found in the static CFG. Hence PICFI provides better security guarantees than conventional CFI which enforce a statically computed CFG.

```
1:  void (*fp)(void);
2:  void foo(int n) {
3:    if (n) {
4:  lpcfi_assign_const(&fp, &g);
5:      fp = &g;
6:    } else {
7:  lpcfi_assign_const(&fp, &h);
8:      fp = &h;
9:    }
10: //unsafe: modify the value of fp
11: lpcfi_check(&fp);
12:   fp();
13: }
14: void main(void) {
15:   foo(1);
16:   foo(0);
17: }
```

(a) Otherwise unsafe code protected by LPCFI.



(b) PICFI's CFG and feasible (green) and infeasible (red) paths not protected by PICFG

Figure 1: A motivating example to demonstrate the limitation of PICFI.

However, PICFI's dynamic CFG grows monotonically, i.e, edges added to the CFG are never removed. Hence, edges become permanently legal to take regardless of whether their legality changes over time. The conservatively constructed dynamic CFG by PICFI leaves an attack surface: when an indirect call transfer remains on the monotonic CFG but will never be legally executed again.

Figure 1 illustrates this limitation of PICFI via a proof-of-concept attack. Note that the lines marked in blue are instrumentation calls from our approach to protect against this attack, and will be explained later. PICFI begins execution with an empty CFG. Initially the indirect callsite `fp()` at line 12 cannot invoke any function legally. After executing the *if* branch via `foo(1)` at line 15, g becomes a legitimate target (e1 and e3 are added to the CFG). After executing the *else* branch via `foo(0)` at line 16, h becomes a legitimate target (e2 and e4 are added to the CFG).

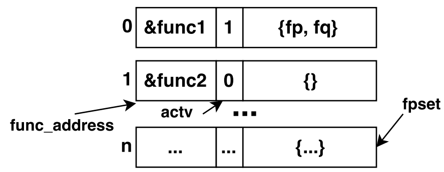
Figure 1(b) gives PICFI's CFG constructed immediately before the indirect callsite `fp()` at line 12 when `foo` is invoked for a second time via `foo(0)` at line 16. Unfortunately, the indirect call edge `fp()` $\xrightarrow{e3}$ g, which was added during the first execution of `foo`, has already become illegal to take since `fp` only points to `h` during the second execution at the time of calling `fp`. However, this spurious edge `fp()` $\xrightarrow{e3}$ g remains on the CFG. The conservative CFG allows attackers to redirect `fp()` to `g` by modifying `fp`'s value to be `g` via a memory corruption error [2], despite foo not being allowed to call g when `n`'s value is 0. Therefore, PICFI still provides an attacker opportunities to launch control hijacking attacks by treating "out-of-date", spurious control flow edges as legitimate. This paper presents LPCFI, Live Path Control Flow Integrity, which aims to overcome

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3195093

```
1:  lpcfi_assign_const(fp, &func) {  //function address-taken statement
2:  ind = lookup(fp_table, &func); //search the index of &func in fp_table
3:  if(ind==-1) error('not found');
4:  fp_table[ind].actv = 1;        //mark func as activated (achieve PICFI)
5:  update(fp, &func); }           //update fp to point to func
6:  fp = &func;
7:  lpcfi_assign_copy(p, q) {  //copy statement
8:  o = pt(fp_table, q)      //get the object that q points to in fp_table
9:  update(p, &o); }    //update p to point to o only if o is a function
10: p = q;
11: lpcfi_assign_load(p, *s) {           //load statement
12: o = pt(fp_table,*s)                  //get the object that *s points to
13: ind = lookup(fp_table, &o);          //search the index of &o in fp_table
14: if(ind==-1) error('not found');
15: assert(fp_table[ind].actv);          //ensure o has been activated
16: update(p, &o); }                     //update p to point to o
17: p = *s;
18: lpcfi_assign_store(*r, q) {          //store statement
19: o = pt(fp_table,*r)                  //get the object that *r points to
20: update(q, &o); }                     //update q to point to o
21: *r = q;
22: lpcfi_check(fp) {        //indirect call statement
23: o = pt(fp_table,fp)   //get the object that fp points to
24: assert(runtimeVal(*fp) == &o && edge(callsite, &o) ∈ static CFG); }
25: fp(...);
```

(a) Instrumentation provided by LPCFI.



(b) Internal representation of the `fp-table`.

```
26:  update(fp, &o) {  //helper function
27:  if(o not a function obj) return;
 //search index of the object that fp points to
28:  oldInd = lookup(fp_table, &pt(fp_table,fp));
 //remove fp from function pointer set of fp_table[oldInd]
29:  if(oldInd!=-1) remove(fp_table[oldInd].fpset, fp);
 //search the index of function o in fp_table
30:  newInd = lookup(fp_table, &o);
31:  if(newInd==-1) error('not found');
 //add fp into the new function pointer set
32:  add(fp_table[newInd].fpset, fp); }
```

(c) Helper function to remove and add function pointers.

Figure 2: Implementation of LPCFI's assignment based instrumentation and its internal data structure

this limitation of PICFI and provide stronger security by both adding and removing CFG edges, allowing at most one outgoing forward edge from every indirect callsite at any one point.

Let us revisit the example in Figure 1 whilst taking into consideration LPCFI's instrumentation (highlighted in blue). During the first call to `foo`, `fp()` $\xrightarrow{e3}$ `g` is added to the CFG via `lpcfi_assign_const`. A check is then performed to ensure that the indirect call transfer from `fp()` will reach the only legitimate target `g`. During the second call to `foo`, `lpcfi_assign_const` in the else branch updates the CFG by first removing the invalid edge `fp()` $\xrightarrow{e3}$ `g` from the CFG, and then adding `fp()` $\xrightarrow{e4}$ `h`. This removal is important since the second call to `foo` via `foo(0)` is not allowed to call `g`, which is ignored by PICFI. LPCFI ensures only one legitimate (live) function target is allowed at any call path to an indirect callsite.

2 APPROACH

We represent a program by putting it into LLVM's SSA form following [5]. The set of all variables is split into two subsets: top-level pointers (registers) whose addresses are not taken, and all potential targets, i.e., all address-taken objects of a pointer. In SSA, a program is represented by five types of statements: `const` (`p = &foo`), `copy` (`p = q`), `store` (`*p = q`), `load` (`p = *q`), and `call` (`fp(...)`).

LPCFI's implementation consists of instrumentation (Figure 2(a)) and a data structure (Figure 2(b)), on which the instrumentation performs bookkeeping to update the dynamic CFG. Like[3, 6], we use safe memory to store LPCFI's metadata. Within this safe memory, LPCFI maintains the `fp-table` as shown in Figure 2(b), which is a fixed size array (the length is the number of address-taken functions in the program) where each element holds: (1) the address of a function `func_address`, (2) an *activation* bit `actv`, and (3) a set `fpset` of function pointers which can legally point to `func_address` at a particular program point during runtime. `pt(fp_table, fp)` returns the function that function pointer `fp` points to. `lookup(fp_table,&func)` returns the index of `&func` in `fp_table`.

We perform instrumentation for an assignment **only if** it may read/write the value of a function pointer as determined by Andersen's pointer analysis [5]. Figure 2(a) details the instrumentation implementation for handling the five types of statements.

The four assignments share the same helper function `update(fp, &o)` in Figure 2(c), which updates a function pointer `fp` to correctly point to a function (e.g., `o`) by removing `fp` from the `fpset` of `fp`'s old points-to target (if it is a member of `fp_table[oldInd].fpset`) at line 29, and adding `fp` to `o`'s `fpset` at line 32. Note that pointer analysis is always an over-approximation. A pointer `q` resolved to point to a function statically, may not point to such at runtime. LPCFI will not perform any runtime update if the right hand side expression of an assignment (e.g., `... = q`) does not refer to a function object as shown at line 27. `lpcfi_check` is inserted immediately before an indirect callsite (lines 22-24). It checks the runtime value of a function target against the value stored in the `fp_table` to validate the indirect call transfer.

Discussion. Performance overhead mainly comes from the helper function due to the search operation on the `fp-table`. Optimisations can be implemented to improve performance of the search operation, e.g., a fast binary search, and caching with a hash map.

The activation bit is used to guarantee that only functions whose addresses have been taken will be considered as legitimate function targets at any indirect calls to provide a security lower bound of that of PICFI. The activation bit is set during runtime when a target has become legitimate at a `const` statement. It is then checked at any load statement which only loads an address-taken function.

3 PROOF-OF-CONCEPT ATTACK & DEFENCE

We have designed a proof-of-concept attack and its defence through LPCFI using the example in Figure 1. Together with our prototype tool, they are available at https://github.com/mbarbar/lpcfi.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (Nov. 2009), 4:1–4:40.

[2] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *CCS '15*. 901–913.

[3] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *ISSTA '17*. ACM, 329–340.

[4] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. In *CCS '15*. 914–926.

[5] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. https://github.com/unsw-corg/SVF. In *CC '16*. 265–266.

[6] Chao Zhang, Scott A Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. 2016. VTrust: Regaining trust on virtual calls. In *NDSS '16*.