Poster: Live Path Control Flow Integrity

Mohamad Barbar, Yulei Sui
University of Technology Sydney

Shiping Chen
CSIRO/Data61

Hongyu Zhang
University of Newcastle

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:

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 call site 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.

However, PICFI’s dynamic CFI 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 call site 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(8) 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 call site fp() at line 12 when foo is invoked for a second time via foo(8) at line 16. Unfortunately, the indirect call edge fp() → 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() → 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 opportunity 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...
Figure 2: Implementation of LPCFI's assignment based instrumentation and its internal data structure

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. Optimisation 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