Live Path CFI Against Control Flow Hijacking Attacks

Mohamad Barbar[1](✉), Yulei Sui[1], Hongyu Zhang[2], Shiping Chen[3], and Jingling Xue[4]

[1] University of Technology Sydney, Sydney, Australia
mbarbar@runbox.com
[2] University of Newcastle, Callaghan, Australia
[3] CSIRO/Data61, Sydney, Australia
[4] University of New South Wales, Sydney, Australia

Abstract. Through memory vulnerabilities, control flow hijacking allows an attacker to force a running program to execute other than what the programmer has intended. Control Flow Integrity (CFI) aims to prevent the adversarial effects of these attacks. CFI attempts to enforce the programmer's intent by ensuring that a program only runs according to a control flow graph (CFG) of the program. The enforced CFG can be built statically or dynamically, and Per-Input Control Flow Integrity (PICFI) represents a recent advance in dynamic CFI techniques. PICFI begins execution 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., edges are never removed when corresponding control flow transfers become illegal. 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 indirect callsites, thereby raising the bar against control flow hijacking attacks.

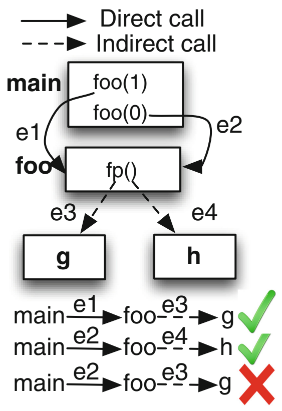
Keyword: Control Flow Integrity

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. In some domains, like embedded systems, these languages are almost ubiquitous. However, these unsafe languages are prone to memory corruption vulnerabilities (e.g., use-after-free and buffer overflows). 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. In realistic scenarios, attackers may be able to perform Turing complete computation by abusing memory vulnerabilities and using techniques like return oriented programming [1] and counterfeit object-oriented programming [2].

```
void (*fp)(void);
void foo(int n) {
  if (n) {
lpcfi_assign_const(fp, &g);
    fp = &g;
  } else {
lpcfi_assign_const(fp, &h);
    fp = &h
  }
//unsafe: modify the value of fp
lpcfi_check(fp);
fp();
}
void main(void) {
  foo(1)
  foo(0)
}
```

(a) Otherwise unsafe code protected by LPCFI.



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

Fig. 1. A motivating example to demonstrate the limitation of PICFI. (Colour figure online.)

Control Flow Integrity (CFI) has been proposed to prevent control flow hijacking [3]. CFI typically works by enforcing a control flow graph (CFG), which represents the programmer's intent - or rather, what can be inferred as legal and illegal control flow from the program. Edges in the CFG represent control flow transfers, and CFI aims to protect indirect control flow edges from being taken illegally. The protection offered by CFI is more effective if a more precise CFG is used. The CFG can be computed statically and this does not consider the fact that the legal status of indirect control flow transfers constantly changes during runtime. For example, when a function pointer is reassigned to a new value, an indirect call via that function pointer will call a new function target and calling the previous target would be illegal.

Insights. Per-Input Control Flow Integrity (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, and during runtime, once 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 CFI techniques which enforce a statically computed CFG. 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 used by PICFI leaves an attack surface: when an indirect transfer remains on the monotonically growing CFG but can never be legally executed again.

Motivating Example. 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 LPCFI approach to protect against this attack, and will be explained below. 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 (e3 is added to the CFG). After executing the *else* branch via `foo(0)` at line 16, `h` becomes a legitimate target (e4 is added to the CFG).

Figure 1b 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 $\texttt{fp()} \xrightarrow{e3} \texttt{g}$, 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 $\texttt{fp()} \xrightarrow{e3} \texttt{g}$ remains on PICFI's CFG. This conservative CFG allows attackers to redirect `fp()` to `g` by modifying `fp`'s value to be `g` via a memory corruption error [5], despite `foo` not being allowed to call `g` when `n`'s value is 0. Therefore, PICFI still provides an attacker opportunities to launch control flow hijacking attacks by treating "out-of-date" control flow edges as legitimate. This paper presents LPCFI, Live Path Control Flow Integrity, which aims to overcome this limitation of PICFI by both adding and removing CFG edges, allowing at most one outgoing forward edge from every indirect callsite at any one program point.

Let us revisit the example in Fig. 1 whilst taking into consideration LPCFI's instrumentation (highlighted in blue). During the first call to `foo`, $\texttt{fp()} \xrightarrow{e3} \texttt{g}$ 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 invalid edge $\texttt{fp()} \xrightarrow{e3} \texttt{g}$ from the CFG, and then adding $\texttt{fp()} \xrightarrow{e4} \texttt{h}$. This removal is important since the second call to `foo` via `foo(0)` is not allowed to call `g`, which PICFI ignores. LPCFI ensures only one legitimate (live) function target is allowed at any call path to an indirect callsite.

Challenges. Designing a CFI technique that overcomes the aforementioned limitation is challenging. Firstly, precise static analysis is required to find statements which may require instrumentation as the precision of static analysis directly correlates with the overhead reduction achieved. Only the statements which may modify or read the value of a function pointer should be identified by static pointer analysis for instrumentation. Secondly, function pointer values need to be correctly maintained in safe memory and the metadata data structure needs to be well designed to ensure efficient lookup and runtime checks.

Our Solution. LPCFI aims to ensure only edges which are currently "live" - can be legally taken - exist within the CFG. We have designed and implemented a new instrumentation approach which tracks function pointers and the address-taken function which they point to at any program point. A function pointer may only ever point to a single function object, so our instrumentation correctly updates which pointers point to which function objects in an efficient data structure in safe memory. We apply pointer analysis [6] to identify all statements which may potentially access the value of a function pointer, and instrument only those statements to minimise runtime overhead. Any callsite from a function pointer is checked to ensure the runtime value matches the value stored in safe memory.

This paper makes the following key contributions:

– We present LPCFI, a new dynamic control flow integrity technique that can protect against attacks undetected by the conservative monotonically growing CFG used by PICFI.
– We propose a new instrumentation approach coupled with a data structure to allow only one function to be a legal target for any indirect callsite.
– We have developed a proof-of-concept attack and defence to demonstrate the effectiveness of LPCFI in mitigating control flow attacks which are not protected by PICFI. This is publicly available at https://github.com/mbarbar/lpcfi.

2 Related Works

Often, CFI implementations determine policy (i.e. valid targets for an indirect control flow transfer at a particular time) according to only static information. This is limited in that some properties are impossible to determine statically, for example, the value of a function pointer reliant on user input.

Per-Input Control Flow Integrity (PICFI) is a CFI implementation which uses dynamic information to gradually build the CFG [4]. PICFI begins execution with an empty CFG; all indirect transfers of control are illegal. The CFG is gradually constructed by discovering valid targets for indirect control flow transfers during runtime according to program inputs. For example, when a function is called, that callsite becomes a valid target of return instructions, or when a function pointer is assigned a value, that value becomes a valid target for indirect callsites (constrained by the static CFG). However, these additions to the CFG are permanent; the CFG grows *monotonically.* This means that changes in target legality are not reflected in the CFG, and hence not enforced by PICFI. A target which is made legal by PICFI is regarded as legal for the rest of execution.

Offering improvements over PICFI, PittyPat [7], a very recent work, uses dynamic path-sensitive points-to analysis to further restrict the set of allowed function pointers at indirect callsites during runtime. Rather than considering just address activation, or the static points-to sets at a particular program point, PittyPat considers the points-to set of a function pointer at a particular program point only based on the *executed* program path. Hence PittyPat avoids PICFI's limitation of keeping previously legal targets which have become illegal. However, PittyPat has a strong dependency on specific hardware and a modified kernel. In contrast, LPCFI is a portable purely software-based approach without any hardware dependency.

A shadow stack is a *second* stack existing in memory used to ensure return instructions jump to the correct address [8,9]. Shadow stacks work by mirroring return addresses pushed onto the execution stack. Upon returning, the value on top of the shadow stack is compared with that on the execution stack, and if the comparison fails, an error is detected. If the shadow stack is safe from manipulation, shadow stacks perfectly protect return transfers. However, shadow stacks only protect backward edges but not forward edges like virtual calls.

CFI techniques have recently been used to protect against virtual table hijacking attacks in low-level object-oriented languages like C++. VTV [10], VTrust [11], and SafeDispatch [12] apply Class Hierarchy Analysis (CHA) to analyse virtual calls to enforce CFI. ShrinkWrap [13] aims to improve CHA based CFI by considering multiple and diamond inheritance. VIP [14] is a recent CFI technique that enforces a more precise call graph than CHA based approaches by using pointer analysis and a fast index-based instrumentation.

This work builds on an earlier version of our work [15].

3 LPCFI Approach

This section details our Live Path Control Flow Integrity approach designed to reduce the attack surface left by PICFI. Section 3.1 describes the program representation of a C/C++ program. Section 3.2 introduces the *fp-table*, the internal metadata design. Finally, Sect. 3.3 describes the instrumentation which operates on the *fp-table* to precisely update the dynamic CFG at runtime.

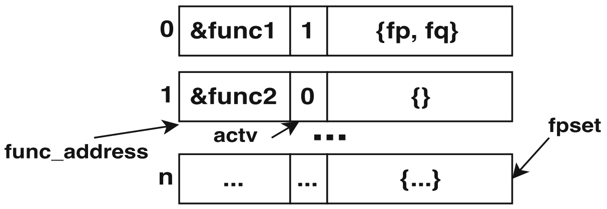


Fig. 2. Internal representation of the fp-table.

3.1 Program Representation

We represent programs in LLVM's SSA form following [6,16]. The set of all variables is separated 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 statement types: `const (p = &o)`, `copy (p = q)`, `store (*p = q)`, `load (p = *q)`, and `call (fp(...))`. Passing arguments into and returning results from functions are modeled by copies. A global variable initialisation is translated into one of the four types of assignments and analysed immediately at the beginning of the main function. For a `const` statement `p = &o` (allocation sites), `o` is a stack or global variable, or a dynamically created abstract heap object. We only analyse statements which access (modify or read) the value of a function pointer according to static pointer analysis [6].

3.2 Data Structures

LPCFI needs to store metadata in the *fp-table* (Fig. 2) to perform bookkeeping to update the dynamic CFG. The metadata is stored in a safe memory region which is accessed frequently for both reading and writing following [11].

LPCFI maintains the fp-table as shown in Fig. 2, which is a fixed size array (size 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 legally point to `func_address` at a particular program point during runtime. `pt(fp_table, fp)` returns the function that pointer `fp` points to. Overloaded `lookup(fp_table, &func)` returns the index of `&func` in the `fp_table`, and `lookup(fp_table, &fp)` returns the index of the function which `&fp` points to in the `fp_table`.

The fp-table is a simple yet efficient solution for fast lookup using a one dimensional array. The fp-table uses function addresses as keys for various reasons. Firstly, it can be a fixed size since functions which may have their addresses taken (`const` statements) at runtime are known statically. Secondly, the checking operation can perform lookups on the function that is about to be called (the runtime value) and retrieve its `fpset`. Finally, a data structure with function addresses as the key is required regardless to keep track of whether functions have been address-taken (activated) to guarantee a lower security bound of PICFI.

3.3 Instrumentation

LPCFI's instrumentation is placed immediately before the five statement types. We insert instrumentation for an assignment **only if** it may read/write a function pointer value as determined by Andersen's pointer analysis [6]. All instrumentations except the checking instrumentation write to the fp-table.

```
update(fp, &o) {
    // Check function object
    if(o not a function obj) return;
    // Search for index of object which fp points to
    oldInd = lookup(fp_table, &pt(fp_table,fp));
    // Remove fp from set of fp_table[oldInd]
    if (oldInd!=-1) remove(fp_table[oldInd].fpset, fp);
    // Search the index of function o in fp_table
    newInd = lookup(fp_table, &o);
    if (newInd==-1) error('not found');
    // Add fp to the new function pointer set
    add(fp_table[newInd].fpset, fp);
}
```

Fig. 3. Helper function `update` to remove and add pointers in the fp-table.

The four assignment instrumentations share helper function `update(fp, &o)` in Fig. 3 which updates a function pointer `fp` to correctly point to function `o` by removing `fp` from the `fpset` of `fp`'s old points-to target (if it is a member) at line 7, and adding `fp` to `o`'s `fpset` at line 12. 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 3 in Fig. 3.

Handling Constant Assignments `fp=&func`: This case, as carried out by `lpcfi_assign_const` shown in Fig. 4, is simple as it is a direct assignment of a function address `func` to a function pointer `fp`. Upon executing this statement, LPCFI requires that, (1) `func` be regarded as activated, and (2) `fp` exclusively points to `func` in the fp-table.

Assignments of this form may execute multiple times for the same RHS value. Hence, functions will be *activated* multiple times. This does not affect correctness and runtime overhead for the activation operation is negligible.

```
lpcfi_assign_const(fp, &func) {
    // Search the index of &func in fp_table
    ind = lookup(fp_table, &func);
    if(ind==-1) error('not found');
    // Mark func as activated
    fp_table[ind].actv_bit = 1;
    // Update fp to point to func
    update(fp, &func);
}
fp = &func;
```

Fig. 4. Handling `const` statements using `lpcfi_assign_const`.

Handling Copy Assignments `p = q`: Represented by `lpcfi_assign_copy`, the second case is also straightforward as shown in Fig. 5. First, we obtain `pt(fp_table,q)`, the points-to target `o` of the RHS pointer `q` derived from the fp-table. Then, `p` is made to exclusively point to `q`'s pointee `o` if `o` is a function object, so both `p` and `q` are put into the `fpset` of object `o`.

```
lpcfi_assign_copy(p, q) {
    // Get the object that q points to in fp_table
    o = pt(fp_table, q);
    // Update p to point to o only if o is a function
    update(p, &o);
}
p = q;
```

Fig. 5. Handling `copy` statements using `lpcfi_assign_copy`.

Handling Load Assignments `p = *s`: `lpcfi_assign_load`'s implementation is shown in Fig. 6. Similar to handling the copy case, we first retrieve points-to target `o` of `*s` from the fp-table. `o` is checked to ensure that it has been activated (lines 5–8) (for a lower bound protection of PICFI, further discussed in Sect. 4.3). Then, `p` is made to exclusively point to the object `o` (line 10).

```
 1:  lpcfi_assign_load(p, *s) {
 2:      // Get the object that *s points to
 3:      o = pt(fp_table,*s);
 4:      // Search for the index of &o in fp_table
 5:      ind = lookup(fp_table, &o);
 6:      if(ind==-1) error('not found');
 7:      // Ensure o has been activated
 8:      assert(fp_table[ind].actv);
 9:      // Update p to point to o
10:      update(p, &o);
11:  }
     p = *s;
```

Fig. 6. Handling `load` statements using `lpcfi_assign_load`.

Handling Store Assignments `*r = q`: `lpcfi_assign_store`'s implementation is shown in Fig. 7. This case is similar to the `copy` case. The points-to target `o` of the RHS pointer `q` is retrieved via `pt(fp_table, q)`. Then, runtime value `*r` is made to exclusively point to the same as that which `q` does in the fp-table.

```
lpcfi_assign_store(*r, q) {
    // Get the object that *r points to
    o = pt(fp_table,q);
    // Update q to point to o
    update(*r, &o);
}
*r = q;
```

Fig. 7. Handling `store` statements using `lpcfi_assign_store`.

Handling Calls `fp(...)`: As shown in Fig. 8, whenever a call is made from a function pointer, the runtime value of the function pointer needs to be checked against its saved value in the fp-table. Furthermore, a check confirming that a callsite-to-target edge is within the static CFG must also be performed to guarantee a security lower bound of PICFI. If either check fails, LPCFI will report an error indicating an attempted illegal control flow transfer.

4 Implementation

We have developed a prototype with a step-by-step live demo to illustrate examples (those in Figs. 1 and 9) that can be protected by LPCFI but not by PICFI. They are publicly available at https://github.com/mbarbar/lpcfi.

```
lpcfi_check(fp, callsite) {
    // Get the object that fp points to
    o = pt(fp_table,fp);
    // Enforce control flow integrity
    assert(runtimeVal(*fp) == &o
        && edge(callsite, &o) ∈ static CFG);
}
fp(...);
```

Fig. 8. Handling `call` statements using `lpcfi_check`.

4.1 Instrumentation and Data Structure

In our open-source prototype, LPCFI's data structure (Fig. 2) and its instrumentation are implemented in an equivalent yet less efficient manner as a standalone library (i.e., `lpcfi.h`, `lpcfi.c`, `fptable.h` and `fptable.c`). In order to demonstrate the key idea and techniques easily, our prototype performs manual instrumention for the motivating example (Fig. 1) as available in `demo.c`.

At indirect callsites, a lookup operation through `lpcfi_check` is performed as discussed in Sect. 4.2. Assignment instrumentations are not idempotent so PICFI's optimsation strategy of patching out instrumentation can not be achieved. `lpcfi_assign_const` performs both function activation (which is idempotent) and fp-table modification. Function activation results in a bit being set and is negligible to the total runtime overhead.

Andersen's pointer analysis [6] is used to check whether pointer dereferences can read or write a function pointer value. This is conservative, so any statement determined to not access such a value is safe without runtime bookkeeping.

4.2 Lookup Operation on the fp-table

The lookup operation is important to LPCFI's metadata manipulation. This happens often, especially since checking function pointer callsites requires this search. During initialisation, the fp-table is sorted according to the `func_address` field for efficient searching. Then, a binary search can be performed on the fp-table with the `func_address` field as the key, an $O(\log n)$ operation.

Overhead mainly comes from the `update` helper function due to the search operation on the `fp-table` for assignment instrumentations. Optimisations can be implemented to improve the performance of the search, e.g., caching common searches with a hash map.

4.3 Security Guarantee

LPCFI guarantees security at a lower bound of PICFI but reduces the attack surface by removing spurious CFG edges during runtime. Following PICFI, LPCFI only allows an indirect call to target a function whose address has been taken (activated) if such callsite-target edge exists in the static CFG. However, LPCFI places a further restriction: that function pointers hold their last assigned value.

Calling a function pointer after it has been modified outside the standard assignment statements results in a raised assertion because assignment instrumentations are the only way to write to the fp-table, which the check operation relies on. Like PICFI, LPCFI enforces control flow integrity, not data flow integrity [17,18]. LPCFI does not ensure memory safety for code and data pointers (e.g., the pointers dereferenced in `load`/`store` statements are unprotected).

```
#include "privileges.h"
/* The header file contains function pointers                        */
/* 'volatile (void)(*priv)(void)' and 'volatile (void)(*nopriv)(void)' */
/* for accessing privileged and non-privileged system methods.       */
int main(void) {
    (void)(*op)(void);
    char password[7];
    while (true) {
        fgets(password, 7, stdin);
        if (strcmp(password, "secret") == 0) {
            lpcfi_assign_copy(op, priv);
            op = priv;
        } else {
            lpcfi_assign_copy(op, nopriv);
            op = nopriv;
        }
        // memory corruption vulnerability: modify the value of op
        lpcfi_check(op);
        op();
    }
}
```

Fig. 9. Password verification cope that is safe with LPCFI, but not with PICFI. (Colour figure online.)

5 Proof-of-Concept Attack and Defence

Figure 9 demonstrates LPCFI's effectiveness over PICFI with a proof-of-concept example in the presence of loops. This is a permission access scenario that allows a user to access a privileged or non-privileged call depending upon the password entered. This demo (including `extended-demo.c`, `privileges.c`, and `privileges.h`) is publicly available in the `extended-demo` folder in our release.

LPCFI's instrumentation is shown in blue (discussed below). In an infinite loop, a user is prompted for a password. If correct, function pointer `op` is set to function pointer `priv`, a privileged operation. If not, `op` is set to function pointer `nopriv`, a non-privileged operation. Finally, `op` is called and the loop begins anew. A memory vulnerability before the call allows an attackers to modify `op`.

If not instrumented, an attacker may change the value of `op` to any value, and the call will target that location. If the code was instrumented by PICFI, initially, the `op` call is deemed unable to target any location legally. The first time the password is entered incorrectly, the `op` call may reach the value pointed to by `nopriv`. Similarly, the first time the password is entered correctly, the `op` call may reach the value pointed to by `priv`. When *both* possible values have been activated, PICFI will see the `op` call as being able to legally take on either value until program exit. If a user enters the password incorrectly, they may modify the value of `op` to be that of the privileged function pointer, and PICFI will allow this call to be made. This is a problem when a malicious user uses the system after a privileged user.

When the code is instrumented with LPCFI (as shown in blue), this problem is remedied. When `op` is set to `priv` at line 12, the `op` call will only succeed if `op` retains the value it was assigned (`priv`). Similarly, when `op` is set to `nopriv` at line 15, for the `op` call to succeed, `op` must retain its value (`nopriv`). The fp-table is storing a **single** value - the most recently assigned value.

6 Conclusion

This paper presents LPCFI, a new dynamic control flow integrity technique that can protect against attacks undetected when using the monotonically growing CFG used by PICFI. LPCFI achieves a lower bound security guarantee of that promised by PICFI but reduces the attack surface left by PICFI using a new instrumentation approach and, with a specially designed data structure, ensures that indirect callsites from function pointers can only target at most one function.

References

1. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: CCS 2007, pp. 552–561 (2007)
2. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., Holz, T.: Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in C++ applications. In: S&P 2015, pp. 745–762 (2015)
3. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. **13**(1), 4:1–4:40 (2009)
4. Niu, B., Tan, G.: Per-input control-flow integrity. In: CCS 2015 (2015)
5. Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., Sidiroglou-Douskos, S.: Control jujutsu: on the weaknesses of fine-grained control flow integrity. In: CCS 2015, pp. 901–913 (2015)
6. Sui, Y., Xue, J.: SVF: interprocedural static value-flow analysis in LLVM. In: CC 2016, pp. 265–266 (2016)
7. Ding, R., Qian, C., Song, C., Harris, B., Kim, T., Lee, W.: Efficient protection of path-sensitive control security. In: USENIX Security 2017, pp. 131–148 (2017)
8. Sinnadurai, S., Zhao, Q., Wong, W.-F.: Transparent runtime shadow stack: protection against malicious return address modifications (2008)
9. Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., Necula, G.C.: XFI: software guards for system address spaces. In: OSDI 2006, pp. 75–88 (2006)

10. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in GCC & LLVM. In: USENIX Security 2014, pp. 941–955 (2014)
11. Zhang, C., Carr, S.A., Li, T., Ding, Y., Song, C., Payer, M., Song, D.: VTrust: regaining trust on virtual calls. In: NDSS 2016 (2016)
12. Jang, D., Tatlock, Z., Lerner, S.: SafeDispatch: securing C++ virtual calls from memory corruption attacks. In: NDSS 2014 (2014)
13. Haller, I., Göktaş, E., Athanasopoulos, E., Portokalidis, G., Bos, H.: ShrinkWrap: VTable protection without loose ends. In: ACSAC 2015, pp. 341–350 (2015)
14. Fan, X., Sui, Y., Liao, X., Xue, J.: Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In: ISSTA 2017, pp. 329–340 (2017)
15. Barbar, M., Sui, Y., Zhang, H., Chen, S., Xue, J.: Live path control flow integrity. In: ICSE 2018 (2018)
16. Sui, Y., Xue, J.: On-demand strong update analysis via value-flow refinement. In: FSE 2016, pp. 460–473 (2016)
17. Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: OSDI 2016, pp. 147–160 (2016)
18. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: OSDI 2014, pp. 147–163 (2014)