Origin-sensitive Control Flow Integrity

Mustakimur Rahman Khandaker
Florida State University
mrk15e@my.fsu.edu

Wenqing Liu
Florida State University
wl16c@my.fsu.edu

Abu Naser
Florida State University
an16e@my.fsu.edu

Zhi Wang
Florida State University
zwang@cs.fsu.edu

Jie Yang
Florida State University
jyang@cs.fsu.edu

Abstract

CFI is an effective, generic defense against control-flow hijacking attacks, especially for C/C++ programs. However, most previous CFI systems have poor security as demonstrated by their large equivalence class (EC) sizes. An EC is a set of targets that are indistinguishable from each other in the CFI policy; i.e., an attacker can "bend" the control flow within an EC without being detected. As such, the large ECs denote the weakest link in a CFI system and should be broken down in order to improve security.

An approach to improve the security of CFI is to use contextual information, such as the last branches taken, to refine the CFI policy, the so-called context-sensitive CFI. However, contexts based on the recent execution history are often inadequate in breaking down large ECs due to the limited number of incoming execution paths to an indirect control transfer instruction (ICT).[1]

In this paper, we propose a new context for CFI, origin sensitivity, that can effectively break down large ECs and reduce the average and largest EC size. Origin-sensitive CFI (OS-CFI) takes the origin of the code pointer called by an ICT as the context and constrains the targets of the ICT with this context. It supports both C-style indirect calls and C++ virtual calls. Additionally, we leverage common hardware features in the commodity Intel processors (MPX and TSX) to improve both security and performance of OS-CFI. Our evaluation shows that OS-CFI can substantially reduce the largest and average EC sizes (by 98% in some cases) and has strong performance – 7.6% overhead on average for all C/C++ benchmarks of SPEC CPU2006 and NGINX.

1 Introduction

The foundation of our software stacks is built on top of the unsafe C/C++ programming languages. C/C++ provides strong performance, direct access to resources, and rich legacy. However, they lack security and safety guarantees of more modern programming languages, such as Rust and Go. Vulnerabilities in C/C++ can lead to serious consequences, especially for low-level software. Many defenses have been proposed to retrofit security into C/C++ programs. Control-flow integrity (CFI) is a generic defense against most, if not all, control-flow hijacking attacks. It enforces the policy that run-time control flows must follow valid paths in the program's control-flow graph (CFG). Since its introduction in the seminal work by Abadi et al. [2], there has been a long stream of research in CFI [1,3,6,9,11–14,16,17,21,25,28,29,31,38,40,41,43,44]. Many earlier systems aim at improving the performance by trading security for efficiency [25,41,43,44], making them vulnerable to various attacks [6, 13, 15, 16]. Recent work focuses more on improving the precision and security of CFI [14,17,21,38], which can roughly be quantified by the average and largest equivalence class (EC) sizes [21]. An EC is a set of targets indistinguishable from each other in the CFI policy; i.e., CFI cannot detect control flow hijacking within an EC. It has been demonstrated the control flow can be "bent" within the ECs without being detected, compromising the protection [6]. Therefore, there is a pressing need to further constrain the leeway of such attacks by reducing the average and largest EC sizes .

One way to improve the security of CFI is to refine the CFG with contextual information, the so-called context-sensitive CFI. Likewise, traditional CFI systems are context-insensitive because they do not collect and use the context information for validating the targets of an ICT. There are many choices of the contextual information. Existing context-sensitive CFI systems use the recent execution history as the context. For example, PathArmor uses the last few branches recorded by Intel processor's Last Branch Record (LBR) [38]; while PittyPat uses the detailed execution paths recorded by Intel processor trace (PT) [14]. Both PathArmor and PittyPat are said to be path-sensitive since they use execution paths as the context. A path-sensitive CFI policy essentially specifies that if the execution comes from this specific path, the ICT can

[1] We use ICT to denote forward indirect control transfers, *excluding* returns. An ICT can be either C-style indirect calls or virtual calls.

only go to that set of targets. There are often multiple paths leading to an ICT. Consequently, the target set of the ICT can be divided into smaller sets by those paths. Another common choice of the context is the call stack [21]. Since the call stack can be represented by its return addresses, such a system is often called call-site sensitive. If the context consists of only one level of return address, it is denoted as 1-call-site sensitive. Similarly, 2-call-site sensitive CFI uses two levels of return addresses as the context.

Execution history based context can substantially reduce the average EC size, but is much less capable in reducing the largest EC size. Unfortunately, the largest EC gives the attacker most leeway in manipulating the control flow without risking detection. For example, PittyPat reports the largest EC size of 218 in SPEC CPU2006, even though it is equipped with the detailed execution history [14]. The fundamental weakness of such context is that most programs only have a small number of execution paths that reach an ICT; i.e., the in-degree of a node (representing an ICT) in the CFG is usually small. If an ICT has hundreds of possible targets, at least one of the ECs will be relatively large. Therefore, such context is more capable in handling small to medium-sized ECs but insufficient for large ones. To address that, we need a more *distributed* context that is not concentrated on the ICT.

In this paper, we propose a new type of context for CFI, origin sensitivity. Origin-sensitive CFI (OS-CFI) takes the origin of the code pointer called by an ICT as the context. It supports both C-style indirect calls and C++ virtual calls with slightly different definitions for them: the origin for the former is the *code* location where the called function pointer is most recently updated; that for the latter is the location where the receiving object (i.e., the object for which the virtual function is called) is created. As usual, returns are protected by the shadow stack, implemented either in software [10,23] or hardware [19]. Our measurement shows that origin sensitivity is particularly effective in breaking down large ECs. For example, it can reduce the largest EC size of a SPEC CPU2006 benchmark from 168 to 2, a reduction of 99% (see Table 1).

We have implemented a prototype of OS-CFI for C and C++ programs. The prototype enforces an adaptive CFI policy that automatically selects call-site or origin sensitivity to protect an ICT in order to improve the system performance without sacrificing security. Its CFG is built by piggybacking on the analysis of a demand-driven, context-, flow-, and field-sensitive static points-to analysis based on SVF (Static Value-Flow Graph) [36]. Its reference monitors are implemented securely and efficiently by leveraging the common hardware features in the commodity Intel processors (MPX and TSX). Our evaluation with SPEC CPU2006, NGINX, and a few real-world exploits shows that the prototype can significantly reduce the average and largest EC sizes, and incurs only a small performance overhead: 7.6% on average for the SPEC CPU2006 and NGINX benchmarks.

In summary, this paper makes the following contributions:

- We propose the concept of origin sensitivity that can substantially reduce both the average and largest EC sizes to improve the security of CFI. Origin sensitivity is applicable to both C-style ICTs and C++ virtual calls. Both types of ICTs are equally important to protect C++ programs.

- We have built a prototype of OS-CFI with the following design highlights: we re-purpose the bound table of MPX to securely store and retrieve origins, and use TSX to protect the integrity of reference monitors; we piggyback on the analysis of SUPA, a precise static points-to algorithm, to built the origin-sensitive CFGs.

- We thoroughly evaluated the security and performance of the prototype with SPEC CPU2006, NGINX, and a few real-world exploits. In particular, we carefully studied the CFGs generated from the points-to analysis and revealed a number of its issues. Detailed CFG generation and measurement are often overlooked in the evaluation of previous CFI systems.

2 Origin Sensitivity

In this section, we first introduce the initial definition of origin sensitivity that is simple, powerful, but potentially inefficient. We then derive a more viable but still effective definition.

2.1 A Simple Definition

OS-CFI takes the origin of the code pointer called by an ICT as the context. If the ICT is a virtual call, the origin is defined as the code location where the receiving object is created, i.e., where its constructor is called; [2] The context of a C-style ICT is similarly defined. A typical example of this type of ICT is an indirect call to a function pointer. The origin of the function pointer is defined as the instruction that initially takes the function address stored in the function pointer.

Next, we use a real-world example from `471.omnetpp` in SPEC CPU2006 to illustrate the concept of the origin (Fig. 1). `471.omnetpp` is a discrete event simulator for large Ethernet networks, written in the C++ programming language. It relies heavily on macros to initialize many objects of the simulated network. Line 1 - 10 shows how simulated networks are initialized: it creates an `ExecuteOnStartup` object for each network to call the network's initialization code; The constructor of `ExecuteOnStartup` sets the private member `code_to_exec` (a function pointer) and adds itself to a linked list (Line 18 - 23). When the program starts, it calls all the queued `code_to_exec` function pointers (`setup` → `executeAll` → `execute`).

The ICT at Line 25 has the largest EC of this program with 168 targets. Call-site sensitivity is not useful here because there is only one call stack to the ICT. Processor-trace-based path

[2] If the object is a global variable, its constructor is conceptually added to a compiler-synthesized function that is called before entering `main()`.

```
1  #define EXECUTE_ON_STARTUP(NAME, CODE)  \
2    static void __##NAME##_code() {CODE;} \
3    static ExecuteOnStartup __##NAME##_reg(__##NAME##_code);
4
5  #define Define_Network(NAME) \
6    EXECUTE_ON_STARTUP(NAME##__net,\
7    (new NAME(#NAME))->setOwner(&networks);)
8
9  Define_Network(smallLAN);
10 Define_Network(largeLAN);
11
12 class ExecuteOnStartup{
13 private:
14   void (*code_to_exec)();
15   ExecuteOnStartup *next;
16   static ExecuteOnStartup *head;
17 public:
18   ExecuteOnStartup(void (*_code_to_exec)()){
19     code_to_exec = _code_to_exec;
20     // add to list
21     next = head;
22     head = this;
23   }
24   void execute(){
25     code_to_exec();
26   }
27   static void executeAll(){
28     ExecuteOnStartup *p = ExecuteOnStartup::head;
29     while (p){
30       p->execute();
31       p = p->next;
32     }
33   }
34 };
35 void cEnvir::setup(...){
36   try{
37     ExecuteOnStartup::executeAll();
38   }
39 }
```

Figure 1: Example to illustrate origin sensitivity

sensitivity can distinguish individual calls to `code_to_exec` (because it records each iteration of the while loop); but it is difficult to decide which target is valid because that depends on the unspecified order in which the constructors are called. Origin sensitivity can handle this case perfectly: the origin of `code_to_exec` is where the related function addresses are initially taken. For example, the macro at Line 9 creates a new function called `__smallLAN__net_code` and passes its address to the constructor of object `__smallLAN__net_reg`. Therefore, Line 9 becomes the origin of this function address. The origin is propagated through the program along with the function address when it is assigned to variables or passed as an argument, in a way similar to how the taint is propagated in taint analysis [33]. At the ICT, the origin is used to verify the target. Because only one function address can be taken at each origin, only one target is possible at the ICT. In other words, origin sensitivity *ideally* can reduce the EC size for this ICT from 168 to 1. The same security guarantee can be achieved for virtual calls because only one class of objects can be created at an origin (Section 2.2).

Execution history based context is limited by the in-degree of an ICT node in the CFG. Assuming the ICT node has n valid targets and m incoming edges, there exists at least one EC with more than $\lceil \frac{n}{m} \rceil$ targets (the pigeonhole principle). For example, the in-degree of the ICT in Fig. 1 is only one for call-site sensitivity; Call-site sensitivity thus cannot reduce this EC at all. The in-degree of this ICT for PathArmor is only 16 because LBR can only record 16 most recent branches. In contrast, origins are associated with the data flow of the program. It traces how function addresses are propagated in the program. Because of this, origin sensitivity can uniquely identify and verify a single target for each ICT. Moreover, this example clearly demonstrates that CFI systems for C++ programs must fully support C-style ICTs because many C++ programs use them (they may even have the largest ECs). Protection of virtual calls alone provides only minimal security.

2.2 A Hybrid Definition

The previous definition of origin sensitivity is conceptually simple but powerful because it can identify a unique target at run-time for each ICT. However, we need to track origins as function addresses are propagated throughout the program in a way similar to how taint is propagated – the origin is the source of the taint, and the ICT is the sink. It is well-known that taint analysis has high overhead, even though the performance of origin tracking could be much better because function addresses are usually not as widespread as the regular data (e.g., a network packet) [23]. This problem is more severe for C-style ICTs because function pointers are frequently copied or passed as arguments. It will not affect virtual calls as much for the following reason: the origin of a virtual call is the location where the receiving object's constructor is called. If an object is copied to another object, we essentially create a new object using its class' copy constructor or copy assignment operator. This creates a new origin for that object. There is thus no need to propagate the origin for objects.

To address the challenge, we propose a hybrid definition of origin sensitivity that combines the origin with call-site sensitivity. More specifically, we relax the definition of the origin as the code location where the related code pointer is most recently updated. In Fig. 1, the only function pointer is `code_to_exec` in the `ExecuteOnStartup` class. It is last updated in the class' constructor at Line 19; i.e., the origin of `code_to_exec` is just Line 19. Clearly, one origin cannot tell Line 9 and 10 (and other places not shown) apart. This can be solved by adding the call-site information to the origin. The origin can now be represented as a tuple of $(\mathcal{CS}, \mathcal{I}_o)$. $\mathcal{I}_o$ is the instruction that last updates the code pointer; $\mathcal{CS}$ is the immediate caller of the origin function (the function that contains $\mathcal{I}_o$). Under this new definition, the ICT at Line 25 has two origins: (Line 9, Line 19) and (Line 10, Line 19). Note how the two elements of the origin complement each other: $\mathcal{I}_o$ moves the context off the *current* execution path

Benchmarks	Language	Context-insensitive	1-call-site		2-call-site		Origin-sensitive	
		EC_L	EC_L	Reduce by	EC_L	Reduce by	EC_L	Reduce by
445.gobmk	C	427	427	0	427	0	427	0
400.perlbench	C	173	120	31%	113	35%	21	88%
403.gcc	C	54	54	0	54	0	42	22%
464.h264ref	C	10	2	80%	2	80%	1	90%
471.omnetpp	C++	168	168	0	168	0	2	99%
483.xalancbmk	C++	38	38	0	38	0	4	95%
453.povray	C++	11	11	0	11	0	10	10%

Table 1: Effectiveness of hybrid origin sensitivity in reducing the largest EC size (EC_L) as compared to call-site sensitivity

(that reaches the ICT); while $\mathcal{CS}$ adds extra information to $\mathcal{I}_o$ to separate different origins. We use call sites here because they can be directly fetched from the shadow stack in the user space. Other execution contexts such as last-branch record and processor trace can only be accessed in the kernel.

Interestingly, the addition of call sites does NOT make the context for virtual calls more powerful. The origin of a virtual call is where the constructor of the receiving object is called. C++'s constructors cannot be called virtually or indirectly.[3] As such, a call to the constructor can create an object of just one class. There is no ambiguity in the class created, hence no ambiguity in the virtual functions. As such, we keep using the object construction site alone as the origin for virtual calls.

Table 1 demonstrates the hybrid origin sensitivity's capability in reducing the largest EC size as compared to call-site sensitivity [21]. Specifically, we run and recorded the complete execution history of all the C/C++ benchmarks in SPEC CPU2006. We then parsed the history to construct the CFGs for origin and call-site sensitivity. For example, 1-call-site sensitivity uses the most recent return address as the context. For each ICT, we grouped the recorded targets by the last return addresses. Each group was an EC. We report the largest EC sizes in Table 1 for all the benchmarks having the largest EC size greater than or equal to 10 (ten other benchmarks have less than 10 targets for every ICT). The table shows that origin sensitivity consistently out-perform call-site sensitivity. Particularly, we can reduce the largest EC size of `471.omnetpp` by 99%, from 168 to 2. Neither call-site nor origin sensitivity is effective to `445.gobmk` because it contains a loop over a large static array of function pointers (the `owl_defendpat` array). `403.gcc` similarly has a large array (`operand_data`) used in a recursive function (`expand_complex_abs`). Common CFI policies cannot handle such cases because there is not sufficient information in the control flow to separate these targets apart. A similar case is shown in Fig. 4 of Section 4 with the code snippet.

3 System Design

In this section, we present the design of our LLVM-based prototype OS-CFI system in detail.

3.1 Overview

Since its inception, many CFI systems have been proposed. To separate OS-CFI from the existing work, we have the following requirements for its design:

- **Precision:** OS-CFI must improve the security by reducing the average and largest EC sizes. Large ECs are the weakest link in a CFI system since they provide the most leeway in "bending" the control flow within the CFI policy
- **Security:** context-sensitive CFI systems, including OS-CFI, have more complex reference monitors to collect and maintain the contextual information. As such, we must protect both the contextual data and the (temporary) data used by reference monitors.
- **Performance:** high performance overhead can severely limit the application of any defense mechanism. OS-CFI must have strong performance relative to the native system.
- **Compatibility:** OS-CFI must support both C and C++ programs. As previously mentioned, any defense for C++ programs must protect both virtual calls and C-style ICTs.

A CFI system consists of three major components: the CFI policy, the CFG generation, and the enforcement mechanism. OS-CFI enforces an adaptive CFI policy that applies either origin or call-site sensitivity for each ICT and adopts the shadow stack to protect returns. OS-CFI's CFG is generated with a precise context-, flow-, and field-sensitive static points-to analysis [36].[4] The enforcement mechanism of OS-CFI uses the hash-table based set-membership test with the hardware acceleration for metadata storage. Next, we describe each component in detail.

[3] Bjarne Stroustrup's C++ Style and Technique FAQ: "To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a 'call to a constructor' cannot be virtual." [8].

[4] Context sensitivity in the points-to analysis is, more precisely, call-site sensitivity. It is named as is for the historical reasons.

3.2 OS-CFI Policy

OS-CFI features an adaptive CFI policy [21] that applies either origin or call-site sensitivity to an ICT, decided by which one is more capable in reducing the EC size. If both have the same effectiveness, we prefer call-site sensitivity because it has lower overhead. If the EC size is already small without context, we just enforce the context-insensitive CFI for this ICT. In addition, call-site sensitivity can use multiple levels of call sites as the context. More levels generally improve the security but incur higher overhead. We limit call-site sensitivity in OS-CFI to at most three call sites. Note that origin sensitivity itself uses 1-call-site on its origins for C-style ICTs (Section 2.2).

We adopt this policy to improve the performance without sacrificing the security: origin sensitivity is a powerful context that can substantially break down large ECs, but it has to collect and maintain more metadata at the run-time. On the other hand, most ICTs in a program have a small number of possible targets. For example, the largest EC size for `400.perlbench` is 173, but its second largest one is only 18. For small ECs, call-site sensitivity is mostly sufficient. We select call-site sensitivity as the secondary policy because last-branch registers (LBR) and processor trace (PT) can only be accessed in the kernel mode, even though they provide more fine-grained execution records. Call-sites instead can be directly fetched from the shadow stack in the user space.

3.3 CFG Generation

A complete and precise CFG is the foundation of any CFI systems. A CFG must be complete to ensure that the resulting CFI system has no false positives (valid control flows reported as invalid). False positives are detrimental to the usability of a security system. Meanwhile, a precise CFG can reduce false negatives, making the system more secure. CFGs can have different levels of precision. For example, a CFG that assumes each ICT can target any address-taken functions is complete but utterly imprecise. Most CFI systems utilize static points-to analysis to construct CFGs because such analysis is (supposedly) conservative and the generated CFGs are complete. The precision of the points-to analysis directly decides the quality of the generated CFGs. A precise points-to analysis is often context- and flow-sensitive, such as SUPA [34].

OS-CFI enforces an adaptive CFI policy that combines call-site and origin sensitivity, which require call-site and origin-sensitive CFGs, respectively. We represent these CFGs as a set of tuples:

- **Call-site sensitive CFG:** each tuple of this CFG has the following form: $(\mathcal{CS}_{1/2/3}, \mathcal{I}_i, \mathcal{T})$. $\mathcal{CS}$ represents the callers of the current function on the call stack. OS-CFI may use up to three call sites. $\mathcal{I}_i$ is the address of the ICT instruction itself. It is either a C-style ICT or a virtual call. $\mathcal{T}$ is the set of valid targets under this context.

```
typedef void (*Format)();
class Base {
protected:
  Format fmt;
public:
  Base(/* Base_o.vPtr, origin */) {
    // store_metadata(Base_o.vPtr, Base::vTavle,
    //                origin);
  }
  ~Base() {}
  virtual void set(Format fp) {
    fmt = fp;
    // store_metadata(fmt.addr, fp.value,
    //                 Base:set_loc1, Base::set_ctx);
  }
  void print() {
    // ccall_ref_monitor(fmt.addr, fmt.value);
    fmt();
  }
};
class Child : public Base {
public:
  Child(/* Child_o.vPtr, origin */) {
    // Base(Child_o.vPtr, origin);
    // store_metadata(Child_o.vPtr, Child::vTable,
    //                 origin);
  }
  ~Child() {}
  void set(Format fp) {
    fmt = fp;
    // store_metadata(fmt.addr, fp.value,
    //            Child::set_loc1, Child::set_ctx);
  }
  void print() {
    // ccall_ref_monitor(fmt.addr, fmt.value);
    fmt();
  }
};
void exec () {
  Base *bp = new Base(); // call constructor
  // vcall_ref_monitor(Base_o.vPtr,
  //                Base::vTable, Base::set())
  bp->set(&targetA);
  bp->print();

  Child ci;              // call constructor
  ci.set(&targetB);
  ci.print();

  bp = &ci;
  // vcall_ref_monitor(Child_o.vPtr,
  //                Child::vTable, Child::set())
  bp->set(&targetB);
  bp->print();
}
```

Figure 2: An example featuring C-style ICT and virtual call.

- **Origin sensitive CFG for C-style ICTs:** each tuple of this CFG has the form of $((\mathcal{CS}_o, \mathcal{I}_o), \mathcal{I}_i, \mathcal{T})$. $(\mathcal{CS}_o, \mathcal{I}_o)$ is the hybrid origin of the function pointer. In particular, $\mathcal{I}_o$ is the last store to the related function pointer; while $\mathcal{I}_i$ is where the function pointer is actually called.

- **Origin sensitive CFG for virtual calls:** each tuple of this CFG has the form of $(\mathcal{I}_o, \mathcal{I}_i, \mathcal{T})$. $\mathcal{I}_o$ is the location where the receiving object of a virtual call is constructed.

We use the C++ code in Fig. 2 to illustrate how the CFGs are generated (and later enforced). There are two classes, `Base` and `Child`. `Child` inherits `Base`. `Base` has a protected function pointer `fmt` that can only be set by virtual function `set`. `fmt` is called indirectly by the `print` function, which is overloaded in `Child`. As such, this example has both C-style ICTs (Line 18 and 36) and virtual calls (Line 43 and 53).

Our CFG construction algorithm is based on SVF, a static tool that "enables scalable and precise inter-procedural dependence analysis for C and C++ programs" [36]. SVF constructs a whole-program sparse value-flow graph (SVFG) that conservatively captures the program's def-use chains. SVFG is imprecise because it overestimates the points-to sets when constructing the def-use chains. SUPA is a client of SVF. It is an on-demand context-, flow-, and field-sensitive points-to analysis based on the SVFG. It improves the precision by refining away imprecise value-flows in the SVFG with strong updates [34]. Our CFGs are constructed on top of the refined SVFG of SUPA.

SUPA is a demand-driven points-to analysis. It traverses the program's SVFG reversely to compute the points-to sets. OS-CFI queries SUPA for every ICT in the program. In response, SUPA starts traversing the def-use chains to solve the request. OS-CFI piggybacks on SUPA during this traversal. Specifically, OS-CFI monitors the traversed nodes to identify the origin of the ICT. When SUPA stops the traversal, it has located the targets of the ICT, and OS-CFI has collected all the elements required to generate the tuples for the ICT. Next, we describe how OS-CFI generates the related tuples for the indirect calls in Fig. 2 since they are the more complex cases.

In Fig. 2, `Base` has a protected member function pointer `fmt`, which is called by `Base.print` and `Child.print`. Therefore, OS-CFI requests SUPA to resolve the points-to set for both uses of `fmt`. We describe the resolution of the first call to `fmt` by `Base.print` here. This indirect call to `fmt` is actually a use of the `fmt` filed of the `this` object. SUPA can create a def-use chain from Line 18 to the assignment of the `fmt` field at Line 12 because it is field-sensitive. This def-use chain is linked by the `bp` pointer created at Line 40. When traversing this def-use chain, OS-CFI marks the first store to `fmt` as the origin for the ICT. The traversal continues until SUPA has reached the call to the `set` function at Line 43. OS-CFI then marks Line 43 as the call-site for the origin. Now, SUPA has located the target of `fmt` (`targetA`). Note that SUPA is precise enough to exclude `targetB` from the points-to set of `fmt` at Line 18. The CFG tuple for Line 18 is ((Line 43, Line 12), Line 18, `targetA`). Tuples for other CFGs can be similarly constructed.

3.4 Enforcement Mechanism

Overview: we use a hash-table based set membership test to enforce the OS-CFI policy. Specifically, we create a hash table for each CFG and instrument the program (at the LLVM IR level) to collect the run-time metadata at the origins. OS-CFI verifies the targets at each ICT site by searching the hash table for matches. As mentioned before, the CFGs are encoded as tuples. The hash function simply takes each element of the tuple and xor them together. It is extremely fast and leads to few conflicts in practice. The hash function can be easily replaced if necessary.

In this section, we will describe the instrumentation in detail. Note that OS-CFI adopts both call-site and origin sensitivity. The context for the former is the return addresses on the call stack, which can be fetched from the shadow stack at the ICT sites. As such, call-site sensitivity is enforced (instrumented) only at the ICT sites. However, we need to instrument both the origin and ICT sites for origin sensitivity.

3.4.1 Instrumentation at Origin Sites

OS-CFI has different origins for C-style ICTs and virtual calls. We describe them separately.

C-style ICTs: the origin for this type of ICTs is defined as $(\mathcal{CS}_o, \mathcal{I}_o)$. $\mathcal{I}$o is the address of the origin (i.e., the instruction that last writes to the function pointer), and $\mathcal{CS}_o$ is the most recent return address on the call stack. Since we are instrumenting the origin, $\mathcal{I}_o$ is a known constant. $\mathcal{CS}_o$ can be retrieved directly from the shadow stack. To store the metadata, we use the address of the function pointer as the key and the context, $(\mathcal{CS}_o, \mathcal{I}_o)$, as the value. At the ICT site, we can recover the context with the function pointer address. Fig. 2 has been annotated with the calls to store metadata at Line 13 and 31 for `Base.fmt` function pointer.

Virtual calls: the origin for virtual calls is the location where the object is created $(\mathcal{I}_o)$. $\mathcal{I}_o$ is also a known constant at the origin site. To store this metadata, we use the object's `vPtr` pointer address as the key and $\mathcal{I}_o$ as the value. In C++, every object with virtual functions has a hidden member named `vPtr` that points to its `vTable`. `vTable` is used by the compiler for dynamic dispatching of virtual function calls. It is a table of virtual function pointers. Each virtual function of a class has a fixed offset in `vTable`. A virtual call is thus compiled as an indirect call to the corresponding entry in `vTable`. Initially, a sub-class inherits its base class' `vTable`. If the sub-class overrides a virtual function, it sets the related function pointer in `vTable` to its own function's address. Consequently, the virtual call can call either the base or sub-class' virtual function, decided by the class of the receiving object. COOP attacks essentially compromise the binding of `vPtr` and `vTable` [32]. After an object is created, its `vTable` will not be changed.

The reason we use `vPtr`'s address as the key (instead of the base address of the object, even though they both can uniquely identify the object) will be clarified as we discuss the metadata storage. The instrumentation is added to each class' constructor so that we only need to insert the code once (instead of once at each location where the constructor is called). Line 7 and 25 of Fig. 2 show the added code. We simply pass the origin from the object allocation site to the constructor as a hidden parameter. Note that the constructor of a sub-class calls the constructor of its base classes first. We thus add the code near the end of the constructor so that the metadata will not be mistakenly overwritten.

Metadata storage: the storage of the contextual information (i.e., the metadata) is a key design component of OS-CFI. The metadata of OS-CFI is organized as (key, value) pairs. The key is the address of the function pointer or the receiving object's `vPtr` pointer. The value is the origin associated with the key. We store the (key, value) pair at each origin site, and query the storage with the same key to retrieve the origin information at each ICT site. The performance and security of the storage is critical to OS-CFI. In our prototype, we uniquely (ab)use the hardware-based bound table of Intel MPX for metadata storage [18].

MPX is a hardware-based bound check system. With the support of the compiler, run-time, and kernel, MPX can check the bounds of memory access to prevent memory errors, such as buffer overflows and over-reads. However, whole program bound check is hard to implement correctly and efficiently, even with the hardware support [26]. In fact, the MPX support will be removed from GCC in version 9.0, after it was just integrated in 5.0 [27]. This leaves the whole MPX hardware free-to-use by OS-CFI and other (security) systems.

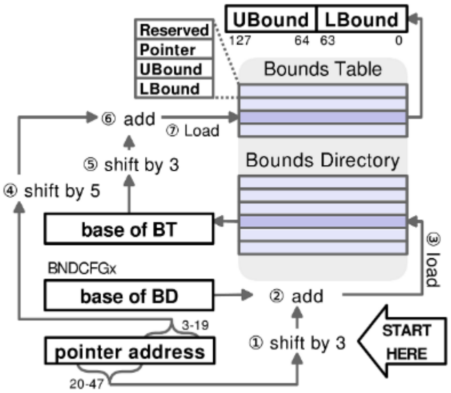

Figure 3: MPX operations, from Intel's manual [30]

MPX 's bound table is indexed by the address of a pointer (i.e., the key). Each key has its own unique bound table entry, which consists of the content of the pointer, the upper bound, and the lower bound. The bound table is organized and operates like a two-level page table, as shown in Fig. 3: the bounds directory points to the second-level bounds tables; each bounds table contains a number of bound entries. The pointer address is divided into two indexes. To locate a bound entry, MPX first indexes into the bounds directory to retrieve the base of the related bounds table, and then uses the second index to locate the related bound entry. If a bounds table does not exist, the kernel allocates a new one and links it to the bounds directory. The base of the bounds directory is stored in a special register, `BNDCFGx`, inaccessible to the user space.

We can store all the origins in the MPX bound table. Even though we are supposed to store the lower and upper bounds in this table, the hardware does not perform any validations on the bounds, as confirmed by both the official document and our experiments. Accordingly, we can store and retrieve arbitrary numbers in the bounds (after doing some simple calculations on these two numbers). This design not only significantly accelerates the access of the metadata but also improves the security: the MPX table stores the content of the key along with the bounds. When querying the table for a key, we need to provide the pointer's address and its content. If the provided pointer content mismatches that in the table, MPX will return an error. Therefore, *we can detect any manipulation of these pointers, after they have been stored, without the extra performance penalty.*

For virtual calls, OS-CFI uses the address of the receiving object's `vPtr` pointer, which points to the object's `vTable`, as the *key*. As such, OS-CFI can readily detect any COOP attack, which compromises the object's `vPtr` pointer [32], similar to how object-type integrity (OTI [4]) works. Note that OTI is not a complete protection for C++ virtual calls because the attacker can still call the "correct" virtual functions of an unintended object. In contrast, OS-CFI provides more comprehensive and complete protection for C++ programs because it not only enforces the precise CFI but also protects both virtual calls and C-style ICTs.

OS-CFI can use other keys for the virtual call. For example, it can use the address of the object itself as the key and a constant number as its content. We can retrieve the origin from the MPX table by this key and its "content", and then enforce the CFI against the unique target decided by the origin. This is because the origin (i.e., the location where the receiving object is constructed) identifies the exact class of the object, hence the unique target of the virtual call (Section 2.2). Our prototype uses the address of the `vPtr` pointer as the key because it is more natural for MPX (i.e., the `vPtr` pointer address and its *real* content). This is not strictly necessary since OS-CFI nevertheless can detect these attacks.

3.4.2 Instrumentation at ICT Sites

The instrumentation at each ICT site is rather straightforward: it first queries the metadata storage with the key and its content to retrieve the origin. If the origin exists, it further checks the corresponding hash table whether the origin and the target are valid for the ICT site. Use the indirect call as an example, we need to reconstruct the tuple of $((\mathcal{CS}_o, \mathcal{I}_o), \mathcal{I}_i, \mathcal{T})$. $(\mathcal{CS}_o, \mathcal{I}_o)$

is the origin fetched from the metadata storage; $\mathcal{I}_i$ represents the address of indirect call instruction; $\mathcal{T}$ is the target, i.e., the value of the function pointer. OS-CFI then queries the hash table whether this tuple is one of its items. If so, the indirect call is allowed. For call-site sensitivity, OS-CFI retrieves the return addresses from the shadow stack and uses a similar method to verify the target under this context.

3.4.3 Protection of Metadata

The MPX table is protected by ASLR. The 64-bit address space provides enough entropy to render brute force attacks difficult, if not impossible. Note that the access to the bounds directory and tables is implemented by the hardware, similar to the access to page tables. Particularly, the base of the bounds directory is stored in a kernel-mode register, inaccessible to the user space. Therefore, the address of the MPX table will not be leaked to the user space. This prevents the attacker from overwriting the metadata stored in the MPX table. We consider side-channel attacks out-of-scope. A number of defenses have been proposed to detect/mitigate them [7, 45]. If a stronger protection of the MPX table is necessary, we can use MPX's bound check to protect it, with a small additional overhead [4, 22]. Note that this use of the bound check does not conflict with OS-CFI's use of the MPX table since the bound check can be performed with just the bound registers.

The hash tables for CFGs are protected as the read-only memory and thus cannot be changed by the attacker. A subtle attack surface is the temporary data used by the reference monitors to search the hash tables. Context-sensitive CFI systems have more complex reference monitors, which have to use the memory (instead of all registers) to store the temporary data. This makes them vulnerable to race conditions in a brief time widow. To address that, we utilize the transactional memory (Intel TSX) to protect the reference monitors [21]. Specifically, TSX keeps tracks of the memory accessed by a transaction and aborts the transaction if any of that memory is changed by others (e.g., attacks). We enclose each reference monitor in a transaction and repeat the transaction if it fails because, with a very low probability, transactions could fail without attacks (e.g., because of cache conflicts).

3.4.4 CFG Address Mapping

Our CFGs are generated using SUPA, a LLVM-based points-to analysis. The resulting CFGs are accordingly encoded as the LLVM IR locations. However, the instrumentation requires the run-time addresses of the CFG nodes. We need to map the IR locations to the run-time addresses. Previous systems often use the debug information for this purpose, which works for function addresses but not as well for call sites because they are not in the symbol table. To address that, heuristics such as the code structure are used to infer the locations of call sites. This approach works most of the time but may not be reliable when the compiler optimization is turned on.

Benchmarks	Out of budget			Empty points-to sets	
	# of ICTs	SUPA	Type	# of ICTs	Type
400.perlbench	54	639	349	2	7
403.gcc	46	544	218	20	107
445.gobmk	22	1645	1637	1	4
447.dealII	0	-	-	23	37
450.soplex	0	-	-	157	11
453.porvray	47	317	79	22	24
471.omnetpp	37	143	44	67	21
483.xalancbmk	0	-	-	349	29
NGINX	141	1066	102	4	34

Table 2: Failed cases of SUPA and the improvements of our type-based matching. Column 3, 4, and 6 show the largest EC sizes for SUPA and the type-based matching. SUPA works for all other benchmarks.

OS-CFI solves this problem without using any heuristics. Specifically, we insert a custom label after each call instruction. We then use the label-as-value extension of Clang to store the label addresses to an array and assign the array to a custom section. The compiler will automatically convert these labels to the addresses. Note that the array has to be marked as used so that the later stages of the compiler will not optimize it away. These extensions are supported by both GCC and Clang/LLVM. A benefit of this approach is that OS-CFI theoretically can support ASLR because the loader will automatically fix these addresses when the program is loaded. This resolves the run-time addresses of call sites. For the rest of the data in the CFGs, we encode the ICT and origin sites as IDs (specifically the hashes of their source code locations) since their concrete values are irrelevant. The target function addresses are obtained from the symbol table. With the address mapping information, we can encode the CFGs in the hash tables.

4 Evaluation

In this section, we evaluate how effectively OS-CFI can improve the security by reducing the largest and average EC sizes and what is the performance overhead for some standard benchmarks. We also experimented with real-world exploits to demonstrate how OS-CFI can block them.

4.1 Improvement in Security

The security of a CFI system can be measured by its CFGs, assuming the enforcement mechanism does not introduce imprecision. Particularly, the average and largest EC sizes reflect the overall quality of the CFGs [21]. OS-CFI's CFGs are derived from SUPA, a static points-to analysis. Therefore, the quality of its CFGs are affected by SUPA.

Advancements and issues of SUPA: SUPA is a scalable and precise context-, flow-, and field-sensitive points-to analysis. Public availability of such algorithms is, to the best of our knowledge, non-existent before the release of SUPA. Though, SUPA has its own issues. More specifically, SUPA is an on-demand points-to analysis. It allocates a specific amount of (configurable) budgets for each query. We found that, even with a generous budget on a relatively powerful machine (a 16-core Xeon server with 64GB of memory), SUPA can still run out of budgets for complex programs, such as `gcc` and `perlbench`. When that happens, SUPA may return wrong results in the points-to sets (e.g., functions with wrong signatures). In addition, SUPA may return empty results because of the language features it does not yet support (e.g., C++'s pointers to member functions). [5] When these issues were detected, we used a simple type-based matching to fix the points-to sets.

The results are listed in Table 2. Generally speaking, the type-based matching can substantially reduce the target sizes for the failed cases. For example, we can reduce the size of largest EC size of NGINX returned by SUPA from 1,066 to just 102. A noticeable exception is `gobmk`, which has more than 1,600 address-taken functions with the same signature (`(int, int, int, int)`). Our manual examination of the program shows that no ICTs in `gobmk` should have more than 500 targets.

It is no surprising that SUPA has some issues because scalable points-to analysis with multiple types of sensitivity is a hard problem. We suspect SUPA is still more scalable and/or precise than other publicly available points-to analysis algorithms, and expect these problems to be solved soon. However, these issues can put OS-CFI to a disadvantage currently – our CFGs are generated by piggybacking on the SUPA as it traverses the SVFG. For these failed cases, SUPA prematurely stops traversing the graph. Accordingly, we cannot generate call-site or origin sensitive edges for these failed ICTs. We instead have to fallback to the context-insensitive CFI for them. We would like to emphasize that *the issues of SUPA does not invalid the usefulness of origin sensitivity.* These are two orthogonal problems.

Effectiveness of OS-CFI: Table 3 shows how OS-CFI can significantly reduce both the average and largest EC sizes. This table focuses on measuring the effectiveness of origin sensitivity; the table thus does not take the ICTs that SUPA failed to resolve (Table 2) into consideration. We will present the overall results with all the ICTs in Table 4. Additionally, Table 3 compares OS-CFI against the context-insensitive CFG, which can be calculated directly from SUPA's points-to sets. It is technically difficult to compare the origin-sensitive CFG against path-sensitive CFGs, such as these in PathArmor [38] or PittyPat [14]: they both use online points-to analysis to calculate the valid targets, with the help of run-time information; i.e., their CFGs are dynamically generated and are, most likely, incomplete for a fair comparison. In addition, the comparison to call-site sensitive CFG has been shown in Table 1. Nevertheless, the absolute average and largest EC sizes OS-CFI can achieve still clearly show its effectiveness. For example, OS-CFI can reduce the largest EC size of `omnetpp` from 109 to 2, a 98% reduction. It can also reduce the average EC size of `gcc` by 96% from 30.7 to 1.3. Overall, OS-CFI can reduce the average and largest EC sizes by 59.8% and 60.2% on average, respectively.

Benchmark	# ICTs	No Context		OS-CFI		Reduce by	
		Avg	Lg	Avg	Lg	Avg	Lg
400.perlbench	79	23.8	39	2.8	10	88%	74%
401.bzip2	20	2.0	2	1.0	1	50%	50%
403.gcc	347	30.7	169	1.3	27	96%	84%
433.milc	4	2.0	2	1.0	1	50%	50%
445.gobmk	36	8.1	107	1.5	12	82%	89%
456.hmmer	9	2.8	10	1.0	1	64%	90%
464.h264ref	367	2.0	12	1.0	2	50%	83%
444.namd	12	2.5	3	1.0	1	60%	67%
447.dealII	79	2.1	3	1.2	3	43%	0%
450.soplex	317	1.0	1	1.0	1	0%	0%
453.porvray	45	9.3	17	1.6	5	83%	71%
471.omnetpp	331	5.7	109	1.0	2	83%	98%
473.astar	1	1.0	1	1.0	1	0%	0%
483.xalancbmk	1492	2.5	11	1.0	1	60%	91%
NGINX	248	9.4	43	1.1	19	88%	56%

Table 3: Improvement of precision by OS-CFI over context-insensitive CFI, shown by the significant reduction in the average (`Avg`) and largest (`Lg`) EC sizes.

```
1  typedef int (*EVALFUNC)(int sq,int c);
2  static EVALFUNC evalRoutines[7] = {
3    ErrorIt,
4    Pawn,
5    Knight,
6    King,
7    Rook,
8    Queen,
9    Bishop };
10
11 int std_eval (int alpha, int beta) {
12   for (j = 1, a = 1; (a <= piece_count); j++) {
13     score += (*(evalRoutines[piecet(i)]))
14                             (i,pieceside(i));
15   }
16 }
```

Figure 4: An example in `sjeng` where the ICT at Line 15 has no context in SUPA.

Overall statistics of OS-CFI: Table 4 shows the overall statistics of OS-CFI when applied to all the C/C++ benchmarks in SPEC CPU2006 and NGINX. The second and third columns show the number of C-style ICTs and virtual calls, respectively. It is clear that C++ programs often use C-style ICTs. Any protection for C++ programs thus must support both types of ICTs. OS-CFI enforces an adaptive policy where an ICT can be protected by either origin or call-site sensitivity. However, it may fall back to the context-insensitive policy if SUPA fails to resolve the points-to set for the ICT or if SUPA

[5] SUPA may also returns empty results for ICTs in the dead code.

Benchmark	#ICTs		OS-CFI / Adaptiveness												
	#c-Call	#vCall	Origin sensitive				Call-site sensitive				Context-insensitive			Overall	
			#ICTs	#Origins	Avg	Lg	#ICTs	Depth	Avg	Lg	#ICTs	Avg	Lg	Avg	Lg
400.perlbench	135	0	53	49	2.5	6	18	2	3.2	8	64	25.5	349	11.4	349
401.bzip2	20	0	20	4	1.0	1	0	0	0	0	0	0	0	1.0	1
403.gcc	413	0	249	139	1.0	1	88	2	1.0	1	76	29.8	218	3.4	218
433.milc	4	0	0	0	0	0	4	1	1.0	1	0	0	1	1.0	1
445.gobmk	59	0	29	12	1.4	3	7	3	1.0	1	23	661.7	1637	246.3	1637
456.hmmer	9	0	1	15	1.0	1	1	1	1.0	1	7	1.0	1	1.0	1
458.sjeng	1	0	0	0	0	0	0	0	0	0	1	7	7	7.0	7
464.h264ref	367	0	318	52	1.0	1	7	1	1.5	2	42	1.7	2	1.1	2
444.namd	12	0	12	30	1.0	1	0	0	0	0	0	0	0	1.0	1
447.dealII	7	95	73	59	1.0	1	3	2	1.0	1	26	27.9	37	6.7	37
450.soplex	0	357	0	0	0	0	0	0	0	0	357	1.2	11	1.2	11
453.porvray	38	76	37	29	1.5	5	8	3	1.0	1	69	14.4	79	7.5	79
471.omnetpp	39	403	276	243	1.0	1	21	2	1.0	1	145	27.5	44	9.2	44
473.astar	0	1	0	0	0	0	0	0	0	0	1	1.0	1	1.0	1
483.xalancbmk	18	2073	1486	1544	1.0	1	6	3	1.0	1	599	7.2	29	3.5	29
NGINX	393	0	184	169	1.0	1	37	3	1.0	1	172	13.8	102	6.6	102

Table 4: Overall distribution of ICTs among origin sensitive, call-site sensitive, and context-insensitive ICTs. The second column shows the total number of C-style indirect calls, while the third column shows the number of virtual calls. We omit the results of `mcf`, `libquantum`, and `sphinx3` from this table because they do not have ICTs in their main programs.Columns marked with `Avg` and `Lg` show the average and largest EC sizes, respectively.

fails to provide the context for the ICT. The latter could happen if the ICT uses global function pointers (e.g., Fig. 4). Specifically, the ICT in Line 13 calls global function pointers defined in the `evalRoutines` array. Because `evalRoutines` is initialized statically, SUPA will not generate any context for this ICT. Neither will origin or call-site sensitivity improve the precision of such cases because the target is decided by the index (`piecet(i)`). Even μCFI can only provide the same precision as context-insensitive CFI in this case because the constraint data (`piecet(i)`) can potentially be compromised before being captured by μCFI using processor trace [17].

In Table 4, most ICTs are protected by origin sensitivity. Interestingly, the number of origins (the 5^{th} column) is often less than the number of ICTs (the 4^{th} column) because some ICTs may share origins. Both origin and call-site sensitivity can reduce most of the average and largest EC sizes to less than 2 and 5, respectively [6]. Note that OS-CFI prefers call-site sensitivity over origin sensitivity. Origin sensitivity is used only if call-site sensitivity fails to provide sufficient security. Therefore, ICTs protected by origin sensitivity generally have larger ECs than those by call-site sensitivity. OS-CFI similarly prefers context insensitivity over call-site sensitivity. ICTs that SUPA failed to resolve are also context-insensitive. The majority of the largest ECs in the context-insensitive ICTs come from the problems in SUPA. We expect OS-CFI to substantially break down most of these ECs once the problems in SUPA are resolved.

Next, we present a few case studies to illustrate how OS-CFI can successfully break down largest ECs in some programs of SPEC CPU2006.

4.1.1 Case Studies

Largest EC in `471.omnetpp`: Fig. 5 shows the virtual call in `471.omnetpp` with the largest number of targets – 35 targets in context-insensitive CFG. The related ICT is located in Line 5, which calls the virtual destructor declared in Line 10. Unlike constructors, destructors in C++ can be called virtually. `cObject` is the root class in `471.omnetpp`. It is inherited by many other classes, such as `CModuleType` (Line 12) and `cArray` (Line 17). Interestingly, `cArray` is a container of `cObject` even though itself is a sub-class of `cObject`. `cArray` has a clear function that calls `discard` on every contained object, which in turn calls the virtual destructor. Clearly, the ICT in Line 5 can target any virtual destructor of `cObject`'s sub-classes.

OS-CFI defines an origin for each location where an object of *cObject* or its sub-class is created. Because the constructor in C++ cannot be virtually called, each origin is associated with exactly one class. As such, OS-CFI can uniquely identify the specific destructor to be called; i.e., it can enforce a perfect CFI policy at Line 5 since the EC size is 1.

Largest EC in `483.xalancbmk`: The ICT with the largest EC size in `483.xalancbmk` is a C-style indirect call (Fig. 6, Line 11). The function pointer is defined in Line 4 as a private member of `XMLRegisterCleanup`. As such, it can only be set by function `registerCleanup` (Line 6). In Line 15 and 16, two objects of `XMLRegisterCleanup` are created. They register the cleanup function at Line 18 and 19, respectively.

The ICT at Line 11 have a EC size of 38. Since this is a C-style ICT, the origin is defined as $(\mathcal{CS}_o, \mathcal{I}_o)$. $\mathcal{I}_o$ is the location of the instruction that last writes to the function pointer (Line 7), while $\mathcal{CS}_o$ is the call sites of the store function (Line 18

[6] Table 3 and 4 cannot be compared directly because Table 4 includes the ICTs SUPA failed to resolve while Table 3 does not.

```
1  class cObject{
2  protected:
3    void discard(cObject *object){
4      if(object->storage() == 'D')
5        delete object;
6      else
7        object->setOwner(NULL);
8    }
9  public:
10   virtual ~cObject();
11 }
12 class cModuleType:public cObject{
13   ~cModule(){
14     delete [] fullname;
15   }
16 }
17 class cArray:public cObject{
18 private:
19   cObject **vect;
20 public:
21   clear(){
22     for (int i=0; i<=last; i++){
23       if (vect[i] && vect[i]->owner()==this)
24         discard( vect[i] );
25     }
26   }
27 }
```

Figure 5: Virtual call with the largest EC in `471.omnetpp`

```
1  class XMLRegisterCleanup
2  {
3  private:
4    XMLCleanupFn m_cleanupFn;
5  public :
6    void registerCleanup(XMLCleanupFn cleanupFn) {
7      m_cleanupFn = cleanupFn;
8    }
9    void doCleanup() {
10     if (m_cleanupFn)
11       m_cleanupFn();
12   }
13 }
14 XMLTransService::XMLTransService(){
15   static XMLRegisterCleanup mappingsCleanup;
16   static XMLRegisterCleanup mappingsRecognizerCleanup;
17 
18   mappingsCleanup.registerCleanup(reinitMappings);
19   mappingsRecognizerCleanup.registerCleanup
20                    (reinitMappingsRecognizer);
21 }
```

Figure 6: The ICT with the largest EC in `483.xalancbmk`

and 19). As such, OS-CFI can enforce a perfect CFI policy for this ICT with an EC size of 1.

Largest EC in `456.hmmer`: `456.hmmer` is a benchmark to measure the performance of searching a gene sequence database. It begins its execution by reading the HMM (Hidden Markov Models) file. This model file can have different versions and formats identified by its magic number. As such, the benchmark creates the `HMMFILE` structure with the `parser` function pointer (Line 9 and 10), and assigns the function pointer according to the model file's magic (Line 14-27). The function pointer is called at Line 31. In total, there are fifteen valid parsers.

```
1  struct hmmfile_s{
2    int (*parser)(struct hmmfile_s *,
3                  struct plan7_s **);
4  };
5  typedef struct hmmfile_s HMMFILE;
6  
7  HMMFILE *HMMFileOpen(char *hmmfile,
8                       char *env){
9    HMMFILE *hmmfp;
10   hmmfp = (HMMFILE*)
11           MallocOrDie(sizeof(HMMFILE));
12   hmmfp->parser = NULL;
13 
14   if(magic == v20magic){
15     hmmfp->parser = read_bin20hmm;
16     return hmmfp;
17   }else if (magic == v20swap){
18     hmmfp->parser = read_bin20hmm;
19     return hmmfp;
20   }else if (magic == v19magic){
21     hmmfp->parser = read_bin19hmm;
22     return hmmfp;
23   }else if (magic == v19swap){
24     hmmfp->parser = read_bin19hmm;
25     return hmmfp;
26   }
27   ...
28 }
29 int HMMFileRead(HMMFILE *hmmfp,
30         struct plan7_s **ret_hmm){
31   return (*hmmfp->parser)(hmmfp,
32                           ret_hmm);
33 }
34 int main(...) {
35   if((hmmfp = HMMFileOpen(hmmfile,
36                  "HMMERDB")) == NULL)
37     Die(...);
38   if(!HMMFileRead(hmmfp, &hmm))
39     Die(...);
40 }
```

Figure 7: The ICT with the largest EC in `456.hmmer`

Because `HMMFileRead` is called in the main function, call-site sensitivity is not useful for this case at all because there is just one call site. As such, OS-CFI applies the origin sensitivity for this ICT. It creates an origin for each assignment to `parser` (Line 15, 18, 21, 24...). Therefore, OS-CFI can enforce a perfect CFI policy for this ICT as well.

4.2 Security Experiments

We experimented with two real-world exploits and one synthesized exploit to show how OS-CFI can block them.

4.2.1 Real-world Exploits

We experimented with two vulnerabilities, CVE-2015-8668 in libtiff and CVE-2014-1912 in python. We used the existing PoC exploits to overwrite a function pointer in order to hijack the control flow. We first verified that the exploits work and then tested them again under the protection of OS-CFI.

CVE-2015-8668: This is a heap-based buffer overflow caused by an integer overflow. The program fails to sanitize the buffer size if the multiplication overflows (Fig. 8, Line 20). This causes the allocated buffer (uncomprbuf) to be too small, allowing the attacker to overflow the heap memory. A potential target of the attack is the TIFF object, which contains several function pointers. One of such function pointers is tif_encoderow, which is called by TIFFWriteScanline later in the program.

```
1  int TIFFWriteScanline(TIFF* tif, ...){
2    ...
3    status = (*tif->tif_encoderow)(tif, (uint8*) buf,
4      tif->tif_scanlinesize, sample); // <= exploit call-point
5  }
6  void _TIFFSetDefaultCompressionState(TIFF* tif){
7    tif->tif_encoderow = _TIFFNoRowEncode; // <= origin
8  }
9  TIFF* TIFFOpen(...){
10   ...
11   _TIFFSetDefaultCompressionState(tif);
12 }
13 int main(int argc, char* argv[]){
14   TIFF *out = NULL;
15   out = TIFFOpen(outfilename, "w"); // <= exploited object
16   ...
17   uint32 uncompr_size;
18   unsigned char *uncomprbuf;
19   ...
20   uncompr_size = width * length; // non-sanitized code and
21                                  // following memory allocation
22   uncomprbuf = (unsigned char *)_TIFFmalloc(uncompr_size);
23   ...
24   if (TIFFWriteScanline(out, ...) < 0) {}
25    ...
26 }
```

Figure 8: Sketch of the vulnerable code in libtiff v4.0.6.

The indirect call at Line 3 was protected in OS-CFI by origin sensitivity. OS-CFI identified twelve origins of tif_encoderow with twelve different targets. However, the only origin recorded during this exploit was the one in the _TIFFSetDefaultCompressionState function, and the corresponding valid target was _TIFFNoRowEncode. Although all twelve origins are possible for the ICT at Line 3, the run-time context allowed us to uniquely identify the only valid target. Our system successfully detected the exploit.

CVE-2014-1912: this buffer overflow in python-2.7.6 is caused by the missing check of buffer size (Fig. 9, Line 28) before receiving the data into a Py_buffer object. Py_buffer has a member of the type PyTypeObject, which contains a function pointer tp_hash. tp_hash is used by the PyObject_Hash function to hash objects. The buffer overflow at Line 31 can be used to overwrite this function pointer.

```
1  int PyType_Ready(PyTypeObject *type){
2    ...
3    bases = type->tp_bases;
4    PyObject *b = PyTuple_GET_ITEM(bases, i);
5    if(PyType_Check(b))
6      inherit_slots(type, (PyTypeObject *)b); // <= origin context
7  }
8  static void inherit_slots(PyTypeObject *t, PyTypeObject *b){
9    ...
10   type->tp_hash = base->tp_hash; // <= origin
11 }
12 long PyObject_Hash(PyObject *v){
13   PyTypeObject *tp = v->ob_type;
14   if (tp->tp_hash != NULL)
15     return (*tp->tp_hash)(v); // <= exploit call-point
16 }
17 static PyObject *sock_recvfrom_into(...){
18   Py_buffer buf;
19   ...
20
21   if (recvlen < 0) {
22     goto error;
23   }
24   if (recvlen == 0) {
25     recvlen = buflen;
26   }
27
28   // missing check if (buflen < recvlen) {}
29
30   // vulnerable code
31   readlen = sock_recvfrom_guts(s, buf.buf, recvlen, flags, &addr);
32 }
```

Figure 9: Sketch of the vulnerable code in Python-2.7.6

Our algorithm identified the origin of tp_hash as Line 10 plus its call-site at Line 6. As such, origin sensitivity is ineffective for the indirect call at Line 15 because there is only one origin. Instead, 3-call-site sensitivity was used for this ICT. We counted 40 immediate call sites to the PyObject_Hash function. With three call-sites, we were able to limit the valid targets to a single candidate for each valid call stack. Our system also successfully prevented this exploit.

In both cases, OS-CFI not only blocked the exploits but also constrained the vulnerable ICTs to a single target at run-time.

4.2.2 Synthesized Exploit: a COOP Attack

We used the example code in Fig. 10 to demonstrate how OS-CFI can detect both vTable hijacking and control-flow hijacking for C++ objects. The example was inspired by PittyPat [14]. There are two virtual calls (Line 44 and 48) and two vulnerable functions (getPerson and isEmployee). The getPerson function contains a heap-based overflow, which allows the attacker to compromise the returned object's vPtr pointer, for example, to overwrite Employee's vPtr to Employer's vTable. The buffer overflow in isEmployee can overwrite res to always return true.

OS-CFI prevented both exploits. The first exploit was de-

```
class Person{
protected:
  SalaryAccount *salary = nullptr;
public:
  virtual void seeEvaluation()=0;
  virtual void seeSalary(){/*null derefernce*/}
};
class Employee : public Person{
public:
  void seeEvaluation(){/* show employee evaluation*/}
  void seeSalary(){/*employee has salary account*/}
};
class Employer : public Person{
public:
  void seeEvaluation(void){/*list of employee evaluation*/}
};
Person *getPerson(int id){
  char *name = (char*)malloc(10);
  Person *p;
  if (isEmployer(id)) {
    p = new Employer();
  } else {
    p = new Employee();
  }
  gets(name); // vulnerable gets()
  ...
  return p;
}
bool isEmployee(Person *member) {
  bool res = false;
  char name[10];
  // vulnerable strcpy()
  strcpy(name, member->getName());
  ...
  if(Employee *emp = dyn_cast<Employee*>(member)){
    if(emp != NULL)
      res = true;
  }
  return res; // attacker overwrite res
}
int main() {
  ...
  Person *member;
  member = getPerson(id);

  // if employee, can only see his/her evaluation
  // if employer, can see list of employee evaluation
  member->seeEvaluation(); // OTI protected

  // only employee has salary account
  if (isEmployee(member))
    member->seeSalary(); // CFI protected
}
```

Figure 10: A program vulnerable to COOP attack.

tected by the MPX table that failed to return the origin because the object's `vPtr` had been changed. The second exploit was blocked because SUPA understands `dyn_case` and correctly determines that `member` should be an object of `Employee`. OS-CFI thus refused to call other class' `seeSalary` function.

4.3 Performance Evaluation

We evaluated the performance of OS-CFI on all the C/C++ benchmarks in SPEC CPU2006 and NGINX. All the experiments were conducted on a server with the Xeon E3-1275 processor and 64 GB of memory, running the 64-bit Ubuntu 16.04.5 LTS Server system. The kernel is the standard Ubuntu kernel 4.4.0-139-generic. All the benchmarks were compiled with the LLVM Safestack, which protects return addresses by relocating them to a separate stack [23]. We use Safestack to demonstrate that OS-CFI is compatible with the return protection. Other return protection can be integrated with OS-CFI too, such as shadow stack [10] and Intel CET [19], with different security and performance impacts. Moreover, OS-CFI uses Intel TSX to protect its reference monitors. This will incur additional overhead. We measured the performance of the benchmarks with and without Intel TSX. All the SPEC CPU2006 benchmarks were measured with the reference data. The results are show in Fig. 11.

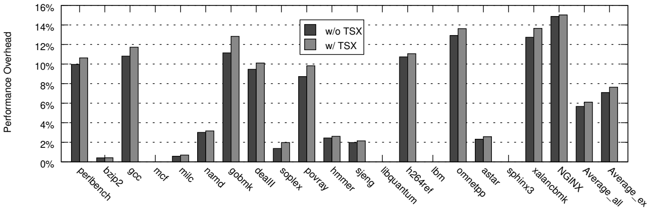

Figure 11: Normalized performance overhead, Average_ex shows the average overhead excluding three benchmarks that have no ICTs.

On average, OS-CFI incurred an overhead of 7.1% without Intel TSX and 7.6% with it. Note that three SPEC benchmarks (`mcf`, `libquantum`, and `sphinx3`) did not have any noticeable overhead because they do not have/use ICTs in their main programs (and Safestack has negligible overhead). We excluded them from the average. The overhead reduced to 5.7% and 6.1%, respectively, if these three benchmarks were included. The highest performance overhead was 14.9% for NGINX and 12.9% for `omnetpp`. The performance of OS-CFI is better than PathArmor (8.5%) [38] and PittyPat (12.7%) [14], even though OS-CFI is a whole program protection while PathArmor only protects a few selected syscalls and PittyPat uses separate threads to parse processor traces (this reduces the number of available CPU cores by half).

OS-CFI generates its CFGs by piggybacking on SUPA. Compared to the original SUPA, it took OS-CFI 5.3% longer on average to analyze these benchmarks, measured on the same machine as the performance evaluation. As shown in Table. 5, the average analysis time is about 3 hours, and the longest analysis time is 15.6 hours from `gcc`. We consider the overhead incurred by OS-CFI acceptable since the analysis is conducted only once offline.

5 Related Work

In this section, we compare OS-CFI to recent (context-sensitive) CFI systems. Since its introduction in 2005 [2], there has been a long list of research in CFI [1, 6, 9, 11–14, 16, 17, 24, 25, 28, 29, 31, 38, 40, 41, 43, 44]. A recent survey by Burow et al. compares many context-insensitive CFI systems [3]. Earlier CFI systems often trade security for performance, introducing

Benchmark	SUPA (s)	OS-CFI (s)	Overhead
400.perlbench	6083.2	6350.7	4.4%
401.bzip2	445.8	457.2	2.6%
403.gcc	53029.1	56231.7	6.0%
433.milc	3.9	4.0	2.6%
445.gobmk	4071.5	4246.4	4.3%
456.hmmer	10.9	11.8	8.3%
458.sjeng	2.6	2.6	0.0%
464.h264ref	372.1	382.0	2.7%
444.namd	15.6	16.7	7.1%
447.dealII	651.5	673.8	3.5%
450.soplex	1280.7	1340.2	4.6%
453.povray	4633.9	5304.0	14.5%
471.omnetpp	43929.0	45351.5	3.2%
473.astar	1.4	1.5	7.1%
483.xalancbmk	9703.7	10792.6	11.2%
NGINX	39860.2	41630.7	4.4%
Average	10255.9	10799.8	5.3%

Table 5: The analysis time of OS-CFI as compared to the vanilla SUPA algorithm. The unit of the analysis time in the table is seconds.

imprecision in both CFGs and enforcement mechanisms. For example, some of them enforce a coarse-grained CFG [37,43], making them vulnerable to attacks [13, 16]. Even precise context-insensitive CFI systems may be vulnerable because of their large EC sizes [6]. Compared to these systems, OS-CFI is a context-sensitive CFI system. Its origin-based context can effectively break down large ECs, improving the security.

An effective method to improve the precision of CFI is to use the contextual information to differentiate sets of targets. However, the addition of context imposes stringent demands on the system design, leading to more trade-offs and opportunities: first of all, a context-sensitive CFI system requires context-sensitive CFGs. It is well-known that context-sensitive points-to analysis does not scale well. The situation has been substantially improved with the recent release of SUPA [35]. However, scalable path-sensitive points-to analysis, needed by systems like PathArmor and PittyPat, is still unavailable; The second challenge is how to securely collect, store, and use contextual information with minimal performance overhead. In the following, we compare OS-CFI to three representative context-sensitive CFI systems: PathArmor [38], PittyPat [14], and μCFI [17]. Table 6 shows their key differences.

PathArmor, PittyPat and μCFI all use the recent execution history recorded by Intel CPUs as the context, last branch record (LBR) for PathArmor and processor trace (PT) for the other two. LBR records only the last sixteen branches taken by the process; while PT provides more fine-grained record of the past execution. Unlike MPX that can be accessed directly in the user space, LBR and PT are privileged and only accessible by the kernel. Transition into and out of the kernel is an expensive operation. It is thus impractical to check these records for each ICT. To address that, PathArmor enforces the CFI policy at the selected syscalls; i.e., only a small part of the program is protected. PittyPat and μCFI redirect the trace to a separate process and verify the control flow there. They can verify all the ICTs but only enforce the results at the selected syscalls. The drawback of this design is that the usable number of CPU cores is effectively reduced by half. Because all three systems cannot enforce the CFI policy at every ICT, their focus is to protect the other part of the system from attacks. OS-CFI instead collects the context by inline reference monitors protected by Intel TSX. It is a whole-program protection that enforces the CFI policy at every ICT. In addition, all these three systems require to change the kernel. OS-CFI uses the stock kernel, whose general MPX support is sufficient.

OS-CFI derives its CFGs from a context-, flow-, and field-sensitive static points-to analysis. However, PathArmor and PittyPat enforce path-sensitivity. To the best of our knowledge, there is no scalable path-sensitive points-to analysis available (at least publicly). Both systems, as well as μCFI, instead utilize on-line points-to analysis, based on the recorded context. The design of μCFI is interesting in that it turns the constraint data into indirect control transfers, which are recorded by PT. This securely conveys the constraint data to the monitoring process. Unfortunately, it seems that this usage puts too much pressure on PT, causing PT to lose packets. This renders μCFI unsuitable for large programs. Indeed, it cannot handle the most demanding benchmarks in SPEC CPU2006, such as `gcc`, `omnetpp`, and `xalancbmk`, and the benchmarks are tested with the smaller train data, not the regular reference data. OS-CFI focuses on reducing the EC sizes. PathArmor and PittyPat are unlikely to achieve the same effectiveness because they use the execution history as the context. The largest EC sizes will remain significant because of the limited incoming paths towards a ICT. For example, PittyPat reports one large EC size of 218. The goal of μCFI is to enforce unique target for each ICT. This is achieved by considering the constraint data during verification. However, the constraint data can potentially be compromised before being captured by μCFI, as mentioned in the paper [17]. This weakens its security guarantee.

CPI is another closely related system. It can guarantee the integrity of all code pointers in the program by separating them and related critical data pointers in a protected safe memory region [23]. As such, CPI can prevent all the control-flow hijacking attacks. Compared to CPI, OS-CFI achieves a similar but slightly relaxed protection in the CFI principle (because OS-CFI still allows some leeway to manipulate the control flow). OS-CFI uses the MPX table to store its metadata. This usage can be applied in CPI as well to further improve its performance, as suggested by the paper itself. Burow et al.

Categories	**CFIXX**	**PathArmor**	**PittyPat**	**μCFI**	**OS-CFI**
Protected	Object type	Control flow	Control flow	Control flow	Control flow & Object type
Context	`vPtr` to `vTable` binding	last branches taken	Processor execution paths	Execution paths and constraint data	Origins of function pointers and objects
CFG	None	On-demand, constraint driven context-sensitive CFG	Abstract-interpretation based online points-to analysis	Run-time points-to analysis	CFGs based on context-, flow- and field-sensitive static points-to analysis
Coverage	Virtual calls	Selected syscalls	Whole program, enforced at selected syscalls	Whole program, enforced at selected syscalls	Whole program, enforced at every ICT
Required hardware	Intel MPX for metadata storage	Intel LBR for taken branches	Intel PT for execution history	Intel PT for execution history and control data	Intel MPX for metadata storage and Intel TSX to protect reference monitors
Kernel changes	No, built-in MPX support	Yes, enforce CFI on the syscall boundary	Yes, redirect traces and enforce CFI on syscall boundary	Yes, redirect traces and enforce CFI on syscall boundary	No, built-in MPX and TSX support
Runtime support	Library to track the type of each object	Per-thread control transfer monitoring	Additional threads to parse trace and verify control flow	Additional threads to parse trace and verify control flow	Hash based verification protected by TSX

Table 6: Comparison between OS-CFI and recent (context-sensitive) CFI systems

independently discovered the way to re-purpose the MPX table as a generic key-value store [5]. As a hardware accelerated data store, MPX can be used in a wide variety of security systems, especially considering that its bound registers can be used for high-performance SFI (software-fault isolation) [4, 22, 39].

Another closely related system is CFIXX [4], which enforces the object-type integrity (OTI). CFIXX prevents attacks such as COOP [32] from subverting an object's `vPtr` pointer. OTI is a complementary policy to CFI [4]. It requires and strengthens CFI to provide more complete protection. OS-CFI's protection of virtual calls uses the same key (but different metadata, i.e., the origin) as OTI as a by-product of using MPX to keep the metadata. As mentioned earlier, OS-CFI can use different keys in its design as long as it can retrieve the origin of the receiving object because the origin alone can uniquely identify the target. Overall, OS-CFI provides stronger security guarantee than CFIXX with its CFI for all ICTs. There are several other systems that focus on protecting virtual calls, such as VTrust [42] and SAFEDISPATCH [20]. OS-CFI supports both C-style ICTs and C++ virtual calls.

6 Summary

We have presented a new type of context for CFI systems – origin sensitivity. By considering the origins of function pointers and objects during the verification of control transfers, we can significantly improve the security of CFI by reducing the largest and average EC sizes. By leveraging the commodity hardware features such as MPX and TSX, our system incurs only a small overhead.

7 Availability

Our prototype is available as an open-source project at `https://github.com/mustakcsecuet/OS-CFI`.

8 Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Dr. Nathan Dautenhahn, for their insightful comments that helped improve the presentation of this paper. This project was partially supported by National Science Foundation (NSF) under Grant 1453020. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Niu, Ben and Tan, Gang , "Per-input Control-flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 914–926.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[3] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 16:1–16:33, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3054924

[4] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CFIXX: Object Type Integrity for C++," in *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018.

[5] N. Burow, X. Zhang, and M. Payer, "SoK: Shining Light on Shadow Stacks," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, ser. SP '19. Washington, DC, USA: IEEE Computer Society, 2019.

[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the 24th USENIX Security Symposium*, vol. 14, 2015, pp. 28–38.

[7] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-channel Attacks in Shielded Execution with Déjá Vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 7–18.

[8] "Bjarne Stroustrup's C++ Style and Technique FAQ," http://www.stroustrup.com/bs_faq2.html, p. 5.

[9] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-flow Integrity for Commodity Operating System Kernels," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 292–307.

[10] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15, 2015.

[11] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted Fine-grained Control-flow Integrity: Towards Efficient Protection of Embedded Systems against Software Exploitation," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[12] L. Davi and A.-R. Sadeghi, "Building Control-flow Integrity Defenses," in *Building Secure Defenses Against Code-Reuse Attacks*. Springer, 2015, pp. 27–54.

[13] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection," in *Proceedings of the 23Rd USENIX Conference on Security*, ser. SEC'14, 2014.

[14] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient Protection of Path-sensitive Control Security," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 131–148. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding

[15] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-grained Control-flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 901–913.

[16] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-flow Integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14, 2014.

[17] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1470–1486. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243797

[18] *Intel 64 and IA-32 Architectures Software Developerś Manual*, Intel.

[19] Intel, "Control-flow Enforcement," https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2018.

[20] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," in *NDSS*, 2014.

[21] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive Call-site Sensitive Control Flow Integrity," in *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P 2019)*, 2019.

[22] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 437–452. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064217

[23] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer Integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 147–163. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[24] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang, "Comprehensive and Efficient Protection of Kernel Control Data," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 4, pp. 1404–1417, Dec 2011.

[25] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque Control-flow Integrity," in *Proceedings of the 22th Network and Distributed System Security Symposium*, ser. NDSS '15, 2015.

[26] "Intel MPX Performance Evaluation for Bound Checking," https://intel-mpx.github.io/performance/.

[27] "GCC 9 Looks Set To Remove Intel MPX Support," https://www.phoronix.com/scan.php?page=news_item&px=GCC-Patch-To-Drop-MPX.

[28] B. Niu and G. Tan, "Modular Control-flow Integrity," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 577–587, 2014.

[29] Niu, Ben and Tan, Gang, "RockJIT: Securing Just-in-time Compilation Using Modular Control-flow Integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1317–1328.

[30] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches," *arXiv preprint arXiv:1702.00719*, 2017.

[31] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained Control-flow Integrity through Binary Hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[32] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*. IEEE, 2015.

[33] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10, 2010.

[34] Y. Sui and J. Xue, "On-demand Strong Update Analysis via Value-flow Refinement," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 460–473.

[35] "Demand Driven Pointer Annalysis," https://github.com/SVF-tools/SUPA.

[36] "Static Value-Flow Graph in LLVM," https://github.com/SVF-tools/SVF.

[37] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-edge Control-flow Integrity in GCC & LLVM," in *USENIX Security Symposium*, 2014, pp. 941–955.

[38] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-sensitive CFI," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.

[39] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-based Fault Isolation," in *Proceedings of the 14th ACM Symposium On Operating System Principles*, December 1993.

[40] Z. Wang and X. Jiang, "Hypersafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-flow Integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 380–395.

[41] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting Violation of Control-flow Integrity Using Performance Counters," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12.

[42] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "VTrust: Regaining Trust on Virtual Calls," in *NDSS*, 2016.

[43] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13, 2013.

[44] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13, 2013.

[45] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016.