# Object Flow Integrity

Wenhao Wang
wenhao.wang@utdallas.edu
The University of Texas at Dallas

Xiaoyang Xu
xiaoyang.xu@utdallas.edu
The University of Texas at Dallas

Kevin W. Hamlen
hamlen@utdallas.edu
The University of Texas at Dallas

## ABSTRACT

Object flow integrity (OFI) augments control-flow integrity (CFI) and software fault isolation (SFI) protections with secure, first-class support for binary object exchange across inter-module trust boundaries. This extends both source-aware and source-free CFI and SFI technologies to a large class of previously unsupported software: those containing immutable system modules with large, object-oriented APIs—which are particularly common in component-based, event-driven consumer software. It also helps to protect these inter-module object exchanges against confused deputy-assisted vtable corruption and counterfeit object-oriented programming attacks.

A prototype implementation for Microsoft Component Object Model demonstrates that OFI is scalable to large interfaces on the order of tens of thousands of methods, and exhibits low overheads of under 1% for some common-case applications. Significant elements of the implementation are synthesized automatically through a principled design inspired by type-based contracts.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Classes and objects**; **Incremental compilers**;

## KEYWORDS

security; control-flow integrity; object-oriented programming; binary transformation

## 1 INTRODUCTION

*Control-flow integrity* (CFI) [2] and *software fault isolation* (SFI) [61] secure software against control-flow hijacking attacks by confining its flows to a whitelist of permissible control-flow edges. The approach has proven successful against some of the most dangerous, cutting-edge attack classes, including *return-oriented programming* (ROP) [51] and other *code-reuse attacks* (CRAs) [5]. Attacks in these families typically exploit dataflow vulnerabilities (e.g., buffer overflows) to corrupt code pointers and thereby redirect control to attacker-chosen program subroutines. By validating each impending control-flow target at runtime before it is reached, CFI guards can often thwart these hijackings.

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3133986

CFI and SFI frameworks work by statically instrumenting control-flow transfer instructions in vulnerable software with extra *guard code* that validates each computed jump destination at runtime. The instrumentation can be performed at compile-time (e.g., [2, 4–6, 32, 36, 43–46, 55, 57, 71]) or on sourceless binaries (e.g., [40, 48, 58, 62, 65, 72–74]). This facility to harden source-free binary software is important for securing software in-flight—allowing third parties to secure dynamically procured binary software on-demand in a way that is transparent to code producers and consumers—and for securing the large quantity of software that is closed-source, or that incorporates software components (e.g., binary libraries) whose source code is unavailable to code consumers.

While the past decade has witnessed rapid progress toward more powerful, higher performance, and more flexible CFI enforcement strategies, there still remain large classes of consumer software to which these technologies are extremely difficult to apply using existing methods. Such limitations often stem from many source-aware CFI algorithms' need for full source code for the entire software ecosystem (e.g., even for the OS kernel, device drivers, and complete runtime system) in order to properly analyze application control-flows, or the difficulty of analyzing complex flows common to certain well-entrenched consumer software paradigms, such as GUI-interactive, event-driven, and component-based software applications. For example, although CFI has been applied successfully to some large applications, in our experience no CFI/SFI algorithm published in the literature to date (see §6) successfully preserves and secures the full functionality of Windows Notepad—one of the most ubiquitous consumer software products available.

The central problem is a lack of first-class support for architectures in which immutable, trusted software components have huge *object-oriented interfaces.* Programs like Notepad interact with users by displaying windows, monitoring mouse events, and sending printer commands. At the binary level, this is achieved by calling runtime system API methods that expect binary objects as input. The app-provided binary object contains a virtual method table (*vtable*), whose members are subsequently called by the runtime system to notify the app of window, mouse, and printer events. The call sites that target untrusted code, and that CFI algorithms must instrument, are therefore *not exclusively located within the untrusted app code*—many are within trusted system libraries that cannot be modified (or sometimes even examined) by the instrumentation process, since they are part of the protected runtime system.

Most CFI algorithms demand write-access to all system software components that may contain unguarded, computed jumps—including the OS, all dynamically loaded libraries, and all future updates to them—in order to ensure safety. In component-driven settings, where modules are dynamically procured on-demand via a distributed network or cloud, this is often impractical. Unfortunately, such settings comprise >98% of the world's software market,[1] including many mission-critical infrastructures that incorporate consumer software components.

One approach for coping with this pervasive problem has been to secure objects passed to uninstrumented modules at call sites within the instrumented modules, before the trusted module receives them (e.g., [57]). But this approach fails when trusted modules retain persistent references to the object, or when their code executes concurrently with untrusted module code. In these cases, verifying the object at the point of exchange does not prevent the untrusted module from subsequently modifying the vtable pointer to which the trusted module's reference points (e.g., as part of a data corruption attack). We refer to such attacks as *COnfused DEputy-assisted Counterfeit Object-Oriented Programming* (CODE-COOP) attacks, since they turn recipients of counterfeit objects [54] into confused deputies [31] who unwittingly invoke policy-prohibited code on behalf of callers.

Faced with such difficulties, many CFI systems conservatively resort to disallowing untrusted module accesses to trusted, object-oriented APIs to ensure safety. This confines such approaches to architectures with few trusted object-oriented system APIs (e.g., Linux), applications that make little or no use of such APIs (e.g., benchmark or command-line utilities), or platforms where the majority of the OS can be rewritten (e.g., ChromeOS [57]). The majority of present-day software architectures that fall outside these restrictive parameters have remained unsupported or receive only incomplete CFI security.

To bridge this longstanding gap, we introduce *object flow integrity* (OFI)—a systematic methodology for imbuing CFI and SFI systems with first-class support for immutable, trusted modules with object-oriented APIs. OFI facilitates safe, transparent flow of binary objects across trust boundaries in multi-module processes, without any modification to trusted module code. To maintain the deployment flexibility of prior CFI/SFI approaches, OFI assumes no access to untrusted application or trusted system source code; we assume only that trusted *interfaces* are documented (e.g., via public C++ header files or IDL specifications).

Our prototype implementation showcases OFI's versatility and scalability by targeting the largest, most widely deployed object-oriented system API on the consumer software market—Microsoft *Component Object Model* (COM) [27]. Most Windows applications rely upon COM to display dialog boxes (e.g., save- and load-file dialogs), create interactive widgets (e.g., ActiveX controls), or dynamically discover needed system services. To handle these requests in a generalized, architecture-independent manner, COM implements an elaborate system of dynamic, shared module loading; distributed, inter-process communication; and service querying facilities—all fronted by a vast, language-independent, object-oriented programming interface. Consequently, COM-reliant applications (which constitute a majority of consumer software today) have remained significantly beyond the reach of CFI/SFI defenses prior to OFI.

To keep our scope tractable, this paper does not attempt to address all research challenges faced by the significant body of CFI literature. In particular, we do not explicitly address the challenges

[1]https://www.netmarketshare.com

```
  Untrusted Module
1 CoCreateInstance(⟨clsid⟩, . . . , ⟨iid1⟩, &o1);
2 o1→Show(. . .);
                                       Trusted Module
3                                      o1→QueryInterface(⟨iid2⟩, &o2);
4                                      o2→GetOptions(. . .);
5                                      o2→Release();
6 o1→GetResult(&o3);
7 o3→GetDisplayName(. . .);
8 o3→Release();
9 o1→Release();
```

**Listing 1: Code that opens a file-save dialog box**

of optimizing the performance of the *underlying* CFI enforcement mechanism, deriving suitable control-flow policies for CFI mechanisms to enforce (cf., [54]), or obtaining accurate native code disassemblies without source code (cf., [66]). Our goal is to enhance existing CFI/SFI systems with support for a much larger class of target application programs and architectures without exacerbating any of these challenges, which are the focuses of related works.

In summary, our contributions are as follows:

- We introduce a general methodology for safely exchanging binary objects across inter-module trust boundaries in CFI/SFI-protected programs without varying trusted module code.
- A prototype implementation for Microsoft COM demonstrates that the approach is feasible for large, complex, object-oriented APIs on the order of tens of thousands of methods.
- A significant portion of the implementation is shown to be synthesizable automatically through a novel approach to reflective C++ programming.
- Experimental evaluation indicates that OFI imposes negligible performance overhead for some common-case, real-world applications.

Section 2 begins with an examination of CODE-COOP attacks and how they manage to evade incomplete CFI protections applied to source-free, component-based software. Section 3 presents OFI's approach to addressing these dangers. Our prototype implementation and its evaluation is presented in Sections 4 and 5, respectively. Related work is summarized in Section 6, and Section 7 concludes.

## 2 BACKGROUND

### 2.1 Inter-module Object Flows

To motivate OFI's design, Listing 1 presents typical C++ code for creating a standard file-open dialog box on a COM-based OS, such as Windows. The untrusted application code first creates a shared object $o_1$ (line 1), where $\langle clsid \rangle$ and $\langle iid_1 \rangle$ are global numeric identifiers for the system's `FileOpenDialog` class and `IFileOpenDialog` interface of that class, respectively. Method `Show` is then invoked to display the dialog (line 2).

While executing `Show`, the trusted system module separately manipulates object $o_1$, including calling its `QueryInterface` method to obtain a new interface $o_2$ for the object, and invoking its methods (lines 3–5). Once the user has finished interacting with the dialog and it closes, the untrusted module calls $o_1$'s `GetResult` method to obtain an `IShellItem` interface $o_3$ whose `GetDisplayName` method discloses the user's file selection (lines 6–7). Finally, the untrusted module releases the shared objects (lines 8–9).

```
1  LPCTSTR lpFileName = TEXT("dnscmmc.dll");
2  HMODULE hModule;
3  IUnknown *o1;
4  HRESULT(WINAPI *lpGCO)(REFCLSID, REFIID, LPVOID*);

6  hModule = LoadLibrary(lpFileName);
7  (FARPROC&) lpGCO = GetProcAddress(hModule, "DllGetClassObject");
8  lpGCO(⟨clsid⟩, ⟨iid_1⟩, (LPVOID*) &o1);

10 // ... code containing a data corruption vulnerability ...

12 IUnknown *o2;
13 o1→QueryInterface(⟨iid_2⟩, (LPVOID*) &o2);
```

**Listing 2: CODE-COOP attack vulnerability**

Safely supporting this interaction is highly problematic for CFI frameworks. All method calls in Listing 1 target *non-exported functions* located in trusted system libraries. The function entry points are only divulged to untrusted modules at runtime within vtables of shared object data structures produced by trusted modules. By default, most CFI policies block such control-flows as indistinguishable from control-flow hijacking attacks.

If one whitelists these edges in the control-flow policy graph to permit them, a significant new problem emerges: Each method call implicitly passes an object reference (the *this* pointer) as its first argument. A compromised, untrusted module can therefore pass a counterfeit object to the trusted callee, thereby deputizing it to commit control-flow violations when it invokes the object's counterfeit method pointers.

One apparent solution is to validate these object references on the untrusted application side at the time they are passed, but this introduces a TOCTOU vulnerability: Since shared COM objects are often dynamically allocated in writable memory, a compromised or malicious application can potentially modify the object's vtable pointer or its contents after passing a reference to it to a trusted module. Trusted modules must therefore re-validate all code pointers at time-of-use to ensure safety, but this breaks CFI's deployment model because it necessitates rewriting all the system libraries.

## 2.2 CODE-COOP Attacks

Listing 2 demonstrates the danger with a common Windows COM programming idiom that is vulnerable to CODE-COOP attack even with CFI protections enabled for all application-provided modules. Lines 6–8 dynamically load a COM library (e.g., dnscmmc.dll) and invoke its `DllGetClassObject` function to obtain an object reference o1. Line 13 later obtains a new interface o2 to the object.

A data corruption vulnerability (e.g., buffer overwrite) in line 10 can potentially allow an attacker to replace o1's vtable with a counterfeit one. CFI protections guarantee that line 13 nevertheless targets a valid `QueryInterface` implementation, but if the process address space contains any system COM library that has not undergone CFI instrumentation, the attacker can redirect line 13 to an unguarded `QueryInterface`. Since all `QueryInterface` implementations internally call other methods on `o1` (e.g., `AddRef`), the attacker can corrupt those to redirect control arbitrarily.

To demonstrate this, we compiled and executed Listing 2 on Windows 10 (Enterprise 1511, build 10586.545) with Microsoft Control Flow Guard (MCFG) [55] enabled, and nevertheless achieved arbitrary code execution. MCFG is a Visual Studio addition that compiles CFI guard code into indirect call sites, including line 13. The guards constrain the sites to a whitelist of destinations. Most Windows 10 system libraries are compiled with MCFG enabled so that their call sites are likewise protected, but many are not. We counted 329 unprotected system libraries on a clean install of Windows 10—many of them in the form of legacy libraries required for backward compatibility. (For example, some have binary formats that predate COFF, and are therefore incompatible with MCFG.) These include dnscmmc.dll (the DNS Client Management Console), which Listing 2 exploits. If an attacker can contrive to load any of them (e.g., through dll injection or by corrupting variable lpFileName in line 6), CODE-COOP attacks become threats. Since COM services obtain libraries dynamically and remotely on-demand, replacement of all 329 of the libraries we found with CFI-protected versions is not an antidote—universal adoption of MCFG across all software vendors and all module versions would be required.

Moreover, even universal adoption of MCFG is insufficient because MCFG cannot protect returns in component-based applications, which are the basis of many code-reuse attacks (e.g., ROP). Stronger CFI systems that do protect returns must likewise universally modify all binary components or suffer the same vulnerability. We consider the existence of at least some uninstrumented modules to be a practical inevitability in most deployment contexts; hence, we propose an alternative approach that augments arbitrary existing CFI approaches to safely tolerate such modules without demanding write-access to system code.

## 3 DESIGN

### 3.1 Object Proxying

OFI solves this problem by ensuring that trusted callee modules (i.e., potential deputies) never receive writable code pointers from untrusted, CFI-protected callers. Achieving this without breaking intricate object exchange protocols and without demanding full source code requires careful design. Our solution centers around the idea of *proxy objects*. Each time an object flows across an inter-module trust boundary, OFI delivers a substitute proxy object to the callee. There are two kinds of proxies in OFI:

- *Floor proxy objects* $\lfloor o \rfloor$ are delivered to trusted callees when an untrusted caller attempts to pass them an object $o$. (Floor objects are so-named because higher-trust tenants see them when "looking down" toward low-trust objects).
- *Ceiling proxy objects* $\lceil o \rceil$ are delivered to untrusted callees when a trusted caller attempts to pass them an object $o$. (Low-trust tenants see them when "looking up" toward high-trust objects.)

Functions $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ are inverses, so $\lfloor \lceil o \rceil \rfloor = \lceil \lfloor o \rfloor \rceil = o$. Thus, if one tenant passes an object to another, who then passes it back, the original tenant receives back the original object, making the proxying transparent to both parties.

At a high level, proxy objects are *in-lined reference monitors* (IRMs) [53] that wrap and mediate access to the methods of the objects they proxy. When called, their methods must (1) enforce

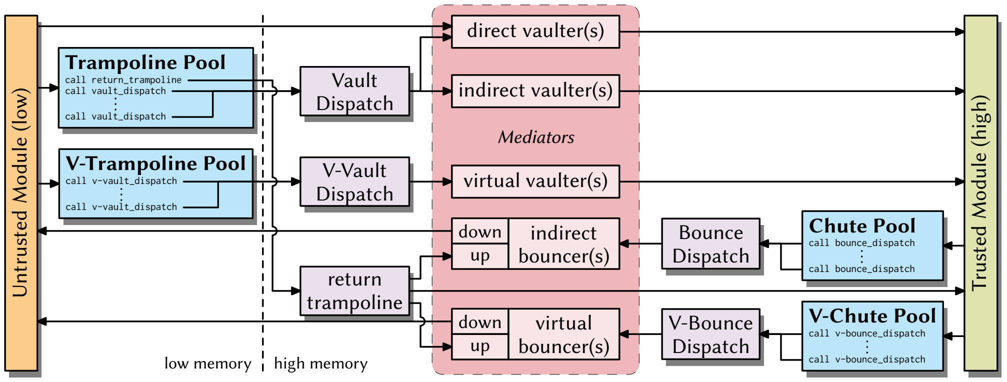

**Figure 1: Cross-module OFI control-flows**

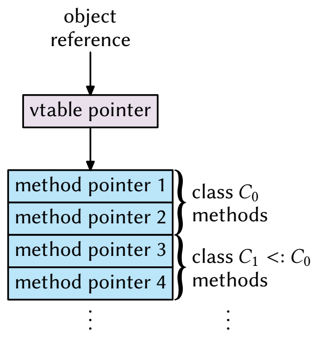

**Figure 2: Proxy object binary representation**

control-flow and dataflow guards that detect and prevent impending CFI violations, and (2) seamlessly purvey the same services as the object they proxy (whenever this does not constitute an integrity violation). In the literature, these requirements are known as IRM *soundness* and *transparency* [30, 35]. The soundness property enforced by a proxy object can be formalized as a type-based contract derivable from the method's type signature, as detailed in §3.2; transparency is achieved by the proxy's reversion to the original object's programming whenever the contract is satisfied.

When applying OFI to binary code without source code, it is not clear where to inject guard code that introduces these proxy objects. All of the calls in Listing 1 take the form of computed jump instructions at the binary level, whose destinations cannot generally be statically predicted. Injecting guard code that accommodates every possible proxy scenario at every computed jump instruction in the program would introduce unacceptable performance overhead.

To avoid this, OFI adopts a lazy, recursive approach to object proxying: At object creation points, OFI substitutes the created objects with proxy objects whose methods are *mediators* that enforce CFI guards before falling through to the proxied object's original programming. The mediators recursively introduce a new layer of proxying for any potentially insecure objects being passed as arguments. Thus, proxying occurs dynamically, on-demand, as each method is called by the various principals and with various object arguments. For example, OFI transforms line 6 so that $\lceil o_1 \rceil \rightarrow$GetResult points to mediator method `GetResult_vaulter`, whose implementation calls $o_1 \rightarrow$GetResult with *this* pointer equal to $\lfloor \lceil o_1 \rceil \rfloor = o_1$. When control returns to the mediator, it replaces out-argument $o_3$ with $\lceil o_3 \rceil$ and then returns to the untrusted caller. We refer to proxy methods that mediate low-to-high calls followed by high-to-low returns as *vaulters*, and those that mediate high-to-low calls followed by low-to-high returns as *bouncers*.

CODE-COOP attacks that attempt to deputize object recipients by corrupting proxy vtables are thwarted by storing proxy objects entirely within read-only memory. This is possible because proxy objects need no writable data; modern object exchange protocols like COM and CORBA require object recipients to access any data via accessor methods (e.g., to accommodate distributed storage), while the object's creator may access in-memory data fields directly. Thus, OFI proxies consist only of a fixed vtable and no data. Moreover, to avoid overhead associated with dynamically allocating them, our design assigns all proxy objects *the same* vtable. This allows the entire proxy object pool to be efficiently implemented as a single, read-only physical page of memory (possibly allocated to multiple virtual pages) filled with the shared vtable's address. Each such vtable pointer constitutes a complete proxy object, ready to be used as a fresh proxy during mediation. The vtable methods all call a central dispatcher method that consults the call stack to determine which proxy object and virtual method is the desired destination, and invokes the appropriate mediator implementation.

Figure 1 illustrates the resulting control-flows. When an untrusted module attempts to call a method of a shared object, the code pointer it dereferences points into a *v-trampoline pool* consisting of direct call instructions that all target OFI's *v-vault dispatch* subroutine. The dispatcher pops the return address pushed by the v-trampoline pool to determine the index of the method being called, and consults the stack's *this* pointer to determine the object. Based on this information, it selects and tail-calls the appropriate *virtual vaulter* mediator. The vaulter proxies any in-arguments, calls the trusted module's implementation of the method, then proxies any out-arguments, and returns to the caller.

In the reverse direction, trusted modules call into a *chute pool* that targets OFI's *bounce dispatch* subroutine, which dispatches control to a *virtual bouncer*. To safely accommodate the return of the untrusted callee to the trusted caller (which constitutes a control-flow edge from untrusted code to a non-exported trusted address, which many CFI policies prohibit), the bouncer replaces the return address with the address of a special *return trampoline* that safely returns control to the "up" half of the bouncer implementation.

This approach generalizes to direct untrusted-to-trusted calls and indirect (non-virtual) untrusted-to-trusted calls, which are both represented atop Figure 1. Direct calls are statically identifiable by (both source-aware and source-free) CFI, and are therefore statically replaced with a direct call to a corresponding *direct vaulter* implementation. Indirect, inter-module calls dereference code pointers returned by the system's dynamic linking API (e.g., `dlsym()` or `GetProcAddress()` on Posix-based or Windows-based OSes, respectively). OFI redirects these to trampoline pool entries that dispatch appropriate *indirect vaulters.* (Dynamic linking can also return pointers to statically linked functions, in which case the dispatcher targets a direct vaulter.)

Another benefit of this proxy object representation strategy is its natural accommodation of subclassing relationships. Callees with formal parameters of type $C_0$ may receive actual arguments of any subtype $C_1 <: C_0$; likewise, callers expecting return values or out-arguments of type $C_0$ may receive objects of any subtype $C_1 <: C_0$. It is therefore essential that proxy objects obey a corresponding subtyping relation that satisfies

$$C_1 <: C_0 \implies (\lfloor C_1 \rfloor <: \lfloor C_0 \rfloor) \land (\lceil C_1 \rceil <: \lceil C_0 \rceil) \tag{1}$$

in order to preserve computations that depend on subtyping.

At the binary level, object vtables support inheritance as illustrated in Figure 2—ordering method pointers from most to least abstract class allows code expecting a more abstract class to transparently access the prefix of the vtable that is shared among all its subclasses. Instantiating all proxy objects with a shared, fixed vtable therefore allows all proxy objects to transparently subtype all other proxy objects (since their vtables are identical). This avoids introducing and consulting potentially complex runtime typing information for each object, which would lead to additional overhead related to protecting that information from malicious tampering.

## 3.2 Type-based Contracts

In order to reliably synthesize and interpose its mediation logic into all trust boundary-crossing method calls, OFI must base its mediation on a description of each interface that links the communicating modules. Since interfaces are collections of method type signatures, OFI therefore enforces a *type-based contract* [22] between caller and callee. That is, each trusted interface method's type signature encodes a set of contractual obligations on code pointers that must be enforced by OFI to ensure CFI-compliant operation. This type-theoretic foundation is essential for scalably automating OFI for large interfaces.

Figure 3 defines OFI contracts as a core subset of the type system used by major interface description languages, such as MIDL and CORBA IDL [21], for component communication. Interface methods have types $\tau \rightarrow_{cc} \tau'$, which denote functions from an argument list of type $\tau$ to a return value of type $\tau'$. Calling convention annotation *cc* is used by OFI to preserve and secure the call stack during calls. Classes, structures, and function argument lists are encoded as tuples $\tau_1 \times \tau_2 \times \cdots \times \tau_n$, which denote structures having $n$ fields of types $\tau_1, \ldots, \tau_n$, respectively. For convenience, named classes are here written as named types $C$ (in lieu of writing

$$\begin{array}{rlr}
\tau : \mathcal{U} ::= & \bot & \text{(security-irrelevant byte)} \\
& \mid \tau_1 \times \tau_2 & \text{(structures)} \\
& \mid \tau^s & \text{(arrays)} \\
& \mid \tau_1 + \tau_2 & \text{(unions)} \\
& \mid C & \text{(shared object classes)} \\
& \mid \tau \rightarrow_{cc} \tau' & \text{(functions)} \\
& \mid [dir]\tau* & \text{(pointers)} \\
& \mid \Sigma_{(v:\tau)} f & \text{(dependent pairs)} \\
& \mid \mu t.\tau \mid t & \text{(recursive datatypes)} \\
s ::= & n \mid \texttt{ZT} \ \textit{(zero-terminated)} & \text{(array sizes)} \\
n \in & \mathbb{N} & \text{(numeric constants)} \\
f : & \mathbb{N} \rightarrow \mathcal{U} & \text{(type dependencies)} \\
dir ::= & \texttt{in} \mid \texttt{out} \mid \texttt{inout} & \text{(argument directions)} \\
cc ::= & \texttt{callee\_pop} \mid \texttt{caller\_pop} & \text{(calling conventions)}
\end{array}$$

**Figure 3: A type system for expressing CFI obligations as OFI contracts**

out their usually large, recursive type signatures). Static-length arrays and zero-terminated strings have repetition types $\tau^n$ and $\tau^{\texttt{ZT}}$, respectively. Pointer arguments whose referents are provided by callers (resp. callees) have type $[\texttt{in}]\tau*$ (resp. $[\texttt{out}]\tau*$). Those with a caller-supplied referent that is replaced by the callee before returning use bidirectional annotation `[inout]`. Self- or mutually-referential types are denoted by $\mu t.\tau$, where $\tau$ is a type that uses type variable $t$ for recursive reference.

For example, Listing 1's `GetResult` method has type

$$\texttt{GetResult} : ([\texttt{in}]C_{\text{IFD}}* \times [\texttt{out}]C_{\text{ISI}}*) \rightarrow_{\texttt{callee\_pop}} \bot^4 \tag{2}$$

where $C_{\text{IFD}}$ and $C_{\text{ISI}}$ are the types of the `IFileDialog` and `IShellItem` interfaces. This type reveals that a correct vaulter for `GetResult` must replace the first stack argument (i.e., the *this* pointer) with a floor proxy of type $\lfloor C_{\text{IFD}} \rfloor$ before invoking the trusted callee, and then replace the second stack argument with a ceiling proxy of type $\lceil C_{\text{ISI}} \rceil$ before returning to the untrusted caller.

In addition to the usual types found in C, we found that we needed *dependent pair types* $\Sigma_{(v:\tau)} f$ in order to express many API method contracts. Values with such types consist of a field $v$ of some numeric type $\tau$, followed by a second field of type $f(v)$. Function $f$ derives the type of the second field from value $v$. For example, the contract of `QueryInterface` is expressible as:

$$\begin{aligned}
&\texttt{QueryInterface} : [\texttt{in}]C_{\text{IFD}}* \times \\
&\quad \Sigma_{(iid:\bot^{16})}(iid = \langle iid_1 \rangle \Rightarrow [\texttt{out}]C_1* \\
&\qquad\quad \mid iid = \langle iid_2 \rangle \Rightarrow [\texttt{out}]C_2* \mid \cdots) \rightarrow_{\texttt{callee\_pop}} \bot^4
\end{aligned} \tag{3}$$

This type indicates that the second stack argument is a 16-byte (128-bit) integer that identifies the type of the third stack argument. If the former equals $\langle iid_1 \rangle$, then the latter has type $[\texttt{out}]C_1*$, etc.

There is a fairly natural translation from interface specifications expressed in C/C++ IDLs, such as SAL, to this type system. Products ($\times$), repetition ($\tau^s$), sums (+), classes ($C$), functions ($\rightarrow$), pointers

$$\begin{aligned}
\mathcal{E}_x\llbracket\bot\rrbracket\,d\,p &= \{\} \\
\mathcal{E}_x\llbracket\tau_1\times\tau_2\rrbracket\,d\,p &= \mathcal{E}_x\llbracket\tau_1\rrbracket\,d\,p;\ \mathcal{E}_x\llbracket\tau_2\rrbracket\,d\,(p+|\tau_1|) \\
\mathcal{E}_x\llbracket\tau^n\rrbracket\,d\,p &= \big(n>0 \Rightarrow (\mathcal{E}_x\llbracket\tau\rrbracket\,d\,p;\ \mathcal{E}_x\llbracket\tau^{n-1}\rrbracket\,d\,(p+|\tau|))\big) \\
\mathcal{E}_x\llbracket\tau^{\mathsf{ZT}}\rrbracket\,d\,p &= \big(*p\neq 0 \Rightarrow (\mathcal{E}_x\llbracket\tau\rrbracket\,d\,p;\ \mathcal{E}_x\llbracket\tau^{\mathsf{ZT}}\rrbracket\,d\,(p+|\tau|))\big) \\
\mathcal{E}_x\llbracket\tau_1+\tau_2\rrbracket\,d\,p &= \mathcal{E}_x\llbracket\tau_1\rrbracket\,d\,p;\ \mathcal{E}_x\llbracket\tau_2\rrbracket\,d\,p \\
\mathcal{E}_x\llbracket\tau\rightarrow_{cc}\tau'\rrbracket\,d\,p &= {}_1\ \text{copy }\tau\text{ from caller to callee;} \\
&\quad {}_2\ \mathcal{E}_x\llbracket\tau\rrbracket\,(\mathtt{in})\,(\&\text{callee\_frame}); \\
&\quad {}_3\ r := \text{call }p; \\
&\quad {}_4\ \mathcal{E}_{x^{-1}}\llbracket\tau'\rrbracket\,(\mathtt{out})\,(\&r); \\
&\quad {}_5\ \mathcal{E}_{x^{-1}}\llbracket\tau\rrbracket\,(\mathtt{out})\,(\&\text{caller\_frame}); \\
&\quad {}_6\ \text{pop }\tau\text{ from }\mathit{opposite}(cc)\text{ stack;} \\
&\quad {}_7\ \text{return }r \\
\mathcal{E}_x\llbracket\mathtt{[}\mathit{dir}\mathtt{]}\tau*\rrbracket\,d\,p &= \big((\mathit{dir}\in\{d,\mathtt{inout}\}\wedge *p\neq 0)\Rightarrow \\
&\quad \text{match }\tau\text{ with }(\_\rightarrow\_)\Rightarrow *p := \&(\mathcal{E}_{x^{-1}}\llbracket\tau\rrbracket\,(\mathtt{in})\,(*p)) \\
&\qquad \mid C\Rightarrow *p := x(*p) \\
&\qquad \mid \_\Rightarrow \mathcal{E}_x\llbracket\tau\rrbracket\,d\,(*p)\big) \\
\mathcal{E}_x\llbracket\Sigma_{(v:\tau)}f\rrbracket\,d\,p &= \mathcal{E}_x\llbracket\tau\rrbracket\,d\,p;\ \mathcal{E}_x\llbracket f(*p)\rrbracket\,d\,(p+|\tau|) \\
\mathcal{E}_x\llbracket\mu t.\tau\rrbracket\,d\,p &= \mathcal{E}_x\llbracket\tau[\mu t.\tau/t]\rrbracket\,d\,p
\end{aligned}$$

**Figure 4: Mediator enforcement of OFI contracts**

($*$), and datatype recursion ($\mu$) are expressed in C++ datatype definitions as structures, arrays, unions, shared classes, function pointers/references, and type self-reference (or mutual self-reference), respectively. SAL annotations additionally specify argument directions and array bounds dependencies. Special dependencies involving class and interface identifiers, such as those in `QueryInterface`'s contract, can be gleaned from the system-maintained list of registered classes and interfaces.

OFI contract types are then automatically translated into effective procedures for enforcing the contracts they denote (i.e., mediator implementations). Figure 4 details the translation algorithm in the style of a denotational semantics[2] where $\mathcal{E}_x\llbracket\tau\rrbracket\,d\,p$ yields a procedure for enforcing the contract denoted by type $\tau$ with proxying function $x\in\{\lfloor\cdot\rfloor,\lceil\cdot\rceil\}$ in call-direction $d\in\{\mathtt{in},\mathtt{out}\}$ on the bytes at address $p$.

For example, $\mathcal{E}_{\lfloor\cdot\rfloor}\llbracket\tau_{\mathtt{GetResult}}\rrbracket(\mathtt{in})(\&\mathtt{GetResult})$ yields the implementation of `GetResult_vaulter`, where $\tau_{\mathtt{GetResult}}$ is the type in equation 2. The implementation first copies caller stack frame $\tau$ to a secure callee-owned stack (line 1). It then enforces the in-contract for $\tau$ (line 2), which replaces the argument of type $C_{\text{IFD}}$ with a proxy of type $\lfloor C_{\text{IFD}}\rfloor$, before invoking `GetResult` (line 3). Upon return, the out-contracts for return type $\tau'$ and frame $\tau$ are enforced (lines 4–5). In this case, return type $\tau'=\bot$[4] is security-irrelevant, but the out-contract for $\tau$ demands replacing stack object $C_{\text{ISI}}$ with proxy $\lceil C_{\text{ISI}}\rceil$. Finally, the frame of the participant (*viz.*, caller or callee) that did not already clean its stack is popped (line 6), and control returns to the caller (line 7). (The first and last steps are required because OFI separates untrusted and trusted stacks for memory safety, temporarily duplicating the shared frame.)

[2]Here, notation $|\tau|$ denotes the size of data having type $\tau$.

Each contract enforcement (lines 2, 4, and 5) entails recursively parsing the binary datatypes of Fig. 3 and substituting code pointers with pointers to mediators that enforce the proper contracts. Structure, array, and union contracts are enforced by recursively enforcing the contracts of their member types. Function pointer contracts are enforced by lazily replacing them with mediator pointers, shared class contracts are enforced by proxying, and other pointer contracts are enforced by eagerly dereferencing the pointer and enforcing the pointee's contract. Dependent pairs are enforced by resolving the dependency to obtain the appropriate contract for the next datum. Finally, recursive types are enforced as a loop that lazily unrolls the type equi-recursively [12].

An OFI implementation can enforce the contract implied by a trusted interface by implementing mediator algorithm $\mathcal{E}_{\lfloor\cdot\rfloor}\llbracket\tau\rightarrow_{cc}\tau'\rrbracket(\mathtt{in})$ for each method signature $\tau\rightarrow_{cc}\tau'$ in the interface. Such mediators are vaulter implementations. Some rules in Fig. 4 invert proxy function $x$, prompting the enforcement to also implement bouncer mediators of the form $\mathcal{E}_{\lceil\cdot\rceil}\llbracket\tau\rightarrow_{cc}\tau'\rrbracket$. These mediate callbacks, such as those commonly used in event-driven programming. Bouncers also mediate methods by which trusted modules initiate unsolicited contact with untrusted modules, such as those that load untrusted libraries and invoke their initializers.

## 3.3 Trust Model

OFI's attacker model assumes that original, untrusted modules may be completely malicious, containing arbitrary native code, but that they have been transformed by CFI/SFI into code compliant with the control-flow policy. The transformed code monitors and constrains all security-relevant API calls and their arguments as long as control-flow stays within the sandbox (cf., [2, 65]). Malicious apps must therefore first escape the control-flow sandbox before they can abuse system APIs to do damage. OFI blocks escape attempts that abuse call sites in immutable modules that depend on objects or code pointers supplied by instrumented modules. It thereby extends whatever policy is enforced by the underlying CFI/SFI mechanism to those call sites. In order to defeat CODE-COOP attacks, the underlying CFI/SFI must therefore enforce a COOP-aware policy [54] for OFI to extend (see §6.3).

Control-flow policies consist of a (possibly dynamic) graph of whitelisted control-flow edges that is consulted and enforced by CFI/SFI guard code before each control-flow transfer from untrusted modules (but not before those from trusted modules). OFI requires that this graph omit edges directly from low- to high-trust modules; such edges must be replaced with edges into OFI's trampoline pools, to afford OFI complete mediation of such transfers.

A facility for read-only, static data is required for OFI to maintain tamper-proof proxy objects. This can be achieved by leveraging CFI/SFI to restrict untrusted access to the system's virtual memory API—untrusted modules must not be permitted to enable write-access to OFI-owned data or code pages.

To prevent untrusted modules from directly tampering with trusted modules' data, some form of memory isolation is required. SFI achieves this by sandboxing all memory-writes by untrusted modules (e.g., [37, 61]). CFI leverages control-flow guards to enforce atomic blocks that guard memory-writes (e.g., [20, 34, 41]).

Data fields of shared objects are conservatively treated as private; non-owners must access shared object data via accessor methods. This is standard for interfaces that support computing contexts where object locations cannot be predicted statically (e.g., in a distributed computations), including all COM interfaces. This affords the accessor methods an opportunity to dynamically fetch or synchronize requested data fields when they are not available locally.

Our design of OFI is carefully arranged to require almost no persistent, writable data of its own, eliminating the need to protect such data within address spaces shared by OFI with malicious modules. In multithreaded processes, OFI therefore conservatively stores its temporary data in CPU registers or other secured, thread-local storage spaces. There are three exceptions:

**Dynamic CFGs.** If the control-flow policy is dynamic (e.g., new edges become whitelisted during dynamic linking), then OFI requires a safe place to store the evolving policy graph. This is typically covered by the underlying SFI/CFI's self-integrity enforcement mechanisms.

**Object Inverses.** A small hash table associating objects with their proxies is required, in order to compute inverses $\lfloor\lceil\cdot\rceil\rfloor$ and $\lceil\lfloor\cdot\rfloor\rceil$. This can be confined to dedicated memory pages, admitting the use of efficient, OS-level memory protections. For example, on Windows desktop OSes we allocate a shared memory mapping to which a separate *memory-manager process* has write access, but to which the untrusted process has read-only access. OFI modules residing in untrusted processes can then use lightweight RPC to write to the hash table. CFI protections prevent untrusted modules from accessing the RPC API to perform counterfeit writes.

**Reference Counts.** To prevent double-free attacks, in which an untrusted module improperly frees objects held by trusted modules, object proxies maintain reference counts independent from the objects they proxy. When the proxy is first created, OFI increments the proxied object's reference count by one. Thereafter, acquires and releases of the proxy are not reflected to the proxied object; they affect only the proxy object's reference count. When the proxy's reference count reaches zero, it decreases the proxied object's reference count by one and frees itself. Proxy object reference counters are stored within the secure hash table entries (see above) to prevent tampering.

## 4 IMPLEMENTATION

### 4.1 Architecture

Our prototype implementation of OFI extends the Reins system [65]. We chose Reins because it realizes fully source-free SFI+CFI (including no reliance on symbol files), and it supports Windows platforms. This affords an aggressive evaluation of OFI's design in austere contexts that lack the benefits of source code and that must support extensive, complex object-oriented APIs, such as COM. Prior to the introduction of OFI enhancements, Reins could not support COM-dependent features of any target application; triggering such features induced its CFI protections to prematurely abort the application with a security violation.

Figure 5 depicts the system architecture. Untrusted native code binaries are first disassembled to obtain a conservative control-flow graph (CFG) policy. The policy dictates that only the control-flow paths statically uncovered and analyzed by the disassembly process are permissible. A binary rewriting module then injects guard code at all control-flow transfer sites to constrain all flows to the CFG.

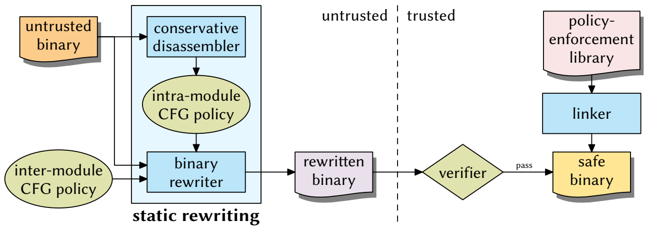

**Figure 5: Reins system architecture**

```
1  void VaultDispatch() {
2    __asm pop eax
3    PROLOGUE // create secure stack frame
4    __asm mov ret_addr, eax
5    index = (trampoline_pool_base − ret_addr) / TRAMPOLINE_SIZE;
6    vaulter_addr = get_vaulter(index);
7    __asm mov eax, vaulter_addr
8    EPILOGUE // pop secure stack frame
9    __asm jmp eax
10 }
```

**Listing 3: Vault Dispatch (abbreviated)**

OFI is agnostic to the particular guard code used to realize SFI/CFI, so we here assume merely that the underlying SFI/CFI implementation protects each control-flow transfer instruction with arbitrary (sound) code pointer validation or sanitization logic (see §6). (Reins employs SFI-style *chunking* and *masking* [37] for efficient sandboxing of intra-module flows, followed by CFI-style whitelisting of inter-module flows. This could be replaced with more precise but less efficient CFI-only logic without affecting OFI.) A separate verifier module independently validates control-flow safety of the secured binary code. This shifts the large, unvalidated rewriting implementation out of the trusted computing base.

Aside from adjusting the control-flow policy to incorporate OFI mediation, OFI extensions inhabit only the *policy enforcement library* portion of the architecture; no change to the disassembly, rewriting, verification, or linking stages was required. This indicates that OFI can be implemented in a modular fashion that does not significantly affect the underlying SFI/CFI system's internals.

The enhancements to the policy enforcement library introduce the inter-module control-flow paths depicted in Figure 1. Their implementations are detailed below.

### 4.2 Dispatcher Implementation

**Vault Dispatch.** OFI's Vault Dispatch subroutine directs control from a non-virtual trampoline to a corresponding vaulter. Listing 3 sketches its implementation. The index of the calling trampoline is first computed from the return address passed by the trampoline to the dispatcher (lines 2–5). Reins allocates exactly one trampoline in the pool for each non-virtual, trusted callee permitted as a jump destination by the policy. The index therefore unambiguously determines the correct vaulter for the desired callee (line 6). No CFI

```
 1 void VVaultDispatch() {
 2   __asm pop ecx
 3   __asm mov eax, [esp+4]
 4   PROLOGUE // create secure stack frame
 5   __asm mov ret_addr, ecx
 6   __asm mov ceiling_proxy_object, eax
 7   index = (vtrampoline_pool_base − ret_addr) / TRAMPOLINE_SIZE;
 8   trusted_object = floor(ceiling_proxy_object);
 9   if (!trusted_object) security_violation();
10   v_vaulter = get_v_vaulter(ceiling_proxy_object, index);
11   __asm mov eax, trusted_object
12   __asm mov [ebp+8], eax
13   __asm mov eax, v_vaulter
14   EPILOGUE // pop secure stack frame
15   __asm jmp eax
16 }
```

**Listing 4: Virtual Vault Dispatch (abbreviated)**

guards are needed here because CFI guard code in-lined into the untrusted call site has already constrained the flow to a permissible trampoline. Finally, the dispatcher tail-calls the vaulter (line 9).

The implementation therefore enforces the control-flow policy in four steps: (1) CFI guard code at the call site ensures that the call may only target trampolines assigned to permissible trusted callees. (2) The dispatcher implementation exclusively calls the vaulter that mediates the CFI-validated callee. (3) The vaulter implementation enforces the callee's OFI contract and exclusively calls the callee it guards. (4) The trusted callee never receives caller-writable object vtables; it only receives immutable proxy objects whose methods re-validate call destinations at time-of-callback. This secures the trusted callee against attacks that try to corrupt or replace the underlying object's vtable.

**V-Vault Dispatch.** Dispatching virtual calls is similar but requires more steps. Listing 4 sketches its implementation. In this case the caller-provided *this* pointer is retrieved along with the trampoline index (lines 3 and 6). Since the destination is a vaulter, valid *this* pointers are always ceiling proxy objects. OFI applies the floor mapping ($\lfloor\cdot\rfloor$) to recover a reference to the trusted function it proxies (line 8). If this fails, a counterfeit object is detected, so OFI aborts with a security violation (line 9). Otherwise the correct vaulter is computed from the ceiling proxy and the index (line 10), the callee's *this* pointer is replaced with the proxied object (lines 11–12), and the vaulter is tail-called (line 15).

**Bounce Dispatch.** Dispatching non-virtual flows from trusted to untrusted modules is analogous to the vault dispatching procedure, except that the dispatcher targets bouncers rather than vaulters, and indexes the chute pool rather than the trampoline pool. The callee-provided return address is also replaced with the address of OFI's return trampoline, so that it can mediate the return.

The bouncer implementation(s) invoked by the dispatcher (see Listing 5) also first switch to a fresh, callee-writable stack (lines 2–9), to prevent the untrusted callee from corrupting trusted caller-owned stack frames before it returns. SFI memory guards prevent the callee from writing into the protected, caller-owned stack. OFI contracts carry sufficient information to implement this stack-switching transparently. For example, the contracts reveal the size

```
 1 void Bouncer() {
 2   PROLOGUE // create untrusted callee stack frame

 4   // switch to new fiber for down part of bouncer
 5   childinfo[0] = &parent_stack;
 6   childinfo[1] = argsize;
 7   childinfo[2] = untrusted_callee_addr;
 8   childfiber = CreateFiber(0, BouncerDown, childinfo);
 9   SwitchToFiber(childfiber);

11   // up part of the bouncer: return from untrusted callee
12   DeleteFiber(childfiber);
13   r = TlsGetValue(tlsindex);
14   enforce_ret_contract(r); // run $\mathcal{E}_{\lfloor\cdot\rfloor}[\![\tau']\!]$out (see Fig. 4)
15   enforce_out_contract(); // run $\mathcal{E}_{\lfloor\cdot\rfloor}[\![\tau]\!]$out (see Fig. 4)

17   // clean stack and return to trusted caller
18   __asm mov eax, r
19   __asm mov ecx, argsize
20   EPILOGUE // pop secure stack frame
21   __asm pop edx
22   __asm add esp, ecx
23   __asm push edx
24   __asm ret
25 }

27 void BouncerDown() {
28   // initialize callee stack
29   __asm sub esp, childinfo[1]
30   __asm mov esi, childinfo[0]
31   __asm mov edi, esp
32   __asm rep movs byte ptr [edi], byte ptr [esi]
33   __asm push offset return_trampoline

35   enforce_in_contract(); // run $\mathcal{E}_{\lceil\cdot\rceil}[\![\tau]\!]$in (see Fig. 4)

37   __asm mov eax, childinfo[2]
38   CFI_VALIDATE(eax)
39   __asm jmp eax
40 }
```

**Listing 5: Bouncer implementation (abbreviated)**

of the topmost (shared) activation frame and the calling convention, allowing that frame to be temporarily replicated on both stacks.

To facilitate efficient stack-switching, we leverage the Windows *Fibers* API [19]. In the trusted-to-untrusted direction, we first create a child fiber. The fiber's stack is arranged so that its return address targets the return trampoline, and the "down" part of the bouncer implementation (lines 27–40) is the child fiber's start address.

The "down" implementation copies the arguments to the new stack (lines 29–32) and then enforces the relevant typing contract on in-arguments (line 35) as described in §3.2, before falling through to the untrusted callee (lines 37–39). Crucially, the underlying object's method pointer is re-validated at time-of-call (line 38), to thwart CODE-COOP attacks.

On return, the return trampoline switches back to the parent fiber, which invokes the "up" half of the bouncer (lines 12–24). This enforces the contracts for return values and out-arguments (lines 14–15) as described in §3.2 before returning to the caller.

```
1  void VBouncerDown() {
2    // initialize callee stack
3    __asm sub esp, childinfo[1]
4    __asm mov esi, childinfo[0]
5    __asm mov edi, esp
6    __asm rep movs byte ptr [edi], byte ptr [esi]
7    __asm push offset return_trampoline
8
9    enforce_in_contract(); // run ℰ⌈·⌉⟦τ⟧in (see Fig. 4)
10
11   // get virtual function address through "this" argument
12   __asm mov eax, [esp+4]
13   __asm mov eax, [eax]
14   __asm mov eax, [eax+childinfo[2]]
15   CFI_VALIDATE(eax)
16   __asm jmp eax
17 }
```

**Listing 6: Virtual Bouncer-Down implementation (abbreviated)**

**V-Bounce Dispatch.** Dispatching virtual calls from trusted to untrusted modules (see Listing 6) is analogous to the bouncer dispatching procedure, except that the child is passed a vtable index rather than a callee entry point address. An extra step is therefore required within the "down" implementation to recover the correct callee method address from the "this" pointer's vtable (lines 12–14). Again, the result is re-validated at time-of-call (line 15) to block CODE-COOP attacks.

**Return Trampoline.** Whenever the trusted caller goes through a bouncer to an untrusted callee, the bouncer creates a new stack in which the return address targets OFI's return trampoline. CFI guards for inter-module return instructions must therefore permit flows to the return trampoline in place of the validated return address. For example, if the underlying CFI system enforces return-flows via a shadow stack, it must validate the return address on the shadow stack as usual, but then allow returning to the return trampoline instead. The return trampoline flows to the "up" half of the bouncer mediator, which returns to the CFI-validated return address stored on the shadow stack. This is the only piece of OFI's implementation that requires explicit cooperation from the underlying CFI implementation.

## 4.3 Automated Mediator Synthesis

When trusted interfaces are specified in a machine-readable format, mediator implementations for them can be automatically synthesized from callee type signatures (see §3.2). Such automation becomes a practical necessity when interfaces comprise thousands of methods or more.

Unfortunately, the only machine-readable specifications of many real-world APIs are as C++ header files, which can be quite complex due to the power of C's preprocessor language, compiler-specific pragmas, and compiler-predefined macros. For example, the `Windows.h` header, which documents the Windows API, defines millions of symbols and macros spanning hundreds of files, and is not fully interpretable by any tool other than Microsoft Visual C++ in our experience. The best tools for parsing them are the C++ compilers intended to consume them.

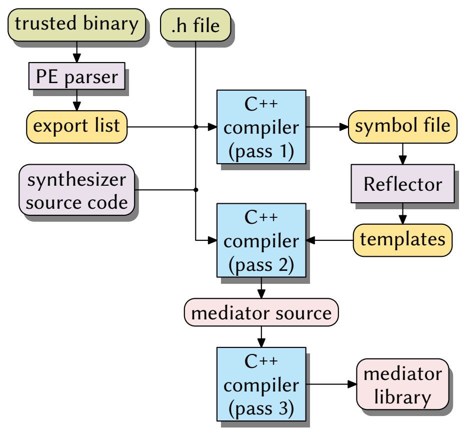

**Figure 6: Automated mediator synthesis**

```
void TrySubmitThreadpoolCallback_vaulter(char x) {
  fix_pointer(&x, 8);
  { _TP_CALLBCK_ENVIRON_V3* x =
          *((_TP_CALLBACK_ENVIRON_V3**) (&x + 8));
    if (x) {
      fix_pointer(&x→CleanupGroupCancelCallback, 8);
      fix_pointer(&x→FinalizationCallback, 8);
    }
  }
  EPILOGUE // pop stack frame
  __asm jmp TrySubmitThreadpoolCallback
}
```

**Listing 7: Synthesized vaulter implementation**

We therefore innovated a strategy of conscripting C++ compilers to interpret interface-documenting header files for us, using the resulting information to automatically synthesize mediation library code. Our strategy achieves static reflective programming for C++ without modifying the compiler, language, or header files. Specifically, our synthesis tool is a C++ program that `#includes` interface headers, and then reflects over itself to inspect function prototypes, structures, and their types. To achieve reflection on structures (which is not supported by C++17 [8]) the program reads its own symbol file in a multi-pass compilation.

Figure 6 illustrates the synthesis process. The interface header, list of exported functions (dumped from the trusted library's export table), and synthesizer source code are first compiled to produce a debug symbol file (e.g., PDB file). Our Reflector tool parses the symbol file to produce C++ templates that facilitate first-class access to the static types of all constituent structure and class members. By including the resulting templates into a second compilation pass, the program reflects upon itself and synthesizes the source code for appropriate mediation code (*viz.*, vaulters and bouncers). A third compilation pass applied to this synthesized mediation code yields the final mediation library.

As an example, Listing 7 shows an automatically synthesized vaulter implementation for the `TrySubmitThreadpoolCallback` Windows API function. In this case, the synthesizer has discovered that the trusted callee treats the top stack argument as a code pointer

```
1  typedef _TP_CALLBACK_ENVIRON_V3 typ1162;

3  template<> struct Reflect<typ1162> {
4    template <typename RetTyp>
5    static inline auto specialize(
6      typ11623 *obj,
7      auto(f)(decltype(typ1162::Version)*, . . . ,
8              decltype(typ1162::CleanupGroupCancelCallback)*,
9              decltype(typ1162::FinalizationCallback)*, . . .
10             )→RetTyp
11   )→RetTyp
12   {
13     return f(
14       &(obj→Version), . . . ,
15       &(obj→CleanupGroupCancelCallback),
16       &(obj→FinalizationCallback), . . .
17     );
18   }
19 }
```

**Listing 8: Reflective template (abbreviated)**

```
1  template <typename... FieldTypes>
2  void enforce_contract(FieldTypes...);

4  template <>
5  void enforce_contract() {}

7  template <typename... FieldTypes>
8  void enforce_contract(FARPROC *field1, FieldTypes... rest) {
9    // Generate C code to enforce FARPROC contract here...
10   enforce_contract(rest);
11 }

13 Reflect<typ1162>::specialize<void>((typ1162*)0, enforce_contract);
```

**Listing 9: Mediator synthesis via template recursion**

to an untrusted callee expecting 8 bytes of stack arguments (line 2). In addition, stack offset 8 holds a pointer to a structure which, if non-null (line 5), contains two more code pointers to untrusted callees expecting 8 bytes of stack arguments each (lines 6–7). Finally, since no out-arguments or return values need sanitization, the vaulter safely tail-calls the trusted callee for more efficient dispatch (line 11).

The typing information necessary to synthesize this implementation is exposed by our Reflect tool as a template of the form shown in Listing 8. The template introduces `Reflect<`$\tau$`>::specialize` as a general mechanism for specializing polymorphic template functions to the particular field types of any desired structure type $\tau$. Specifically, lines 7–10 declare a function parameter $f$ whose arguments are specialized to the field types of $\tau$. When called, `Reflect<`$\tau$`>::specialize(`$o, f$`)` therefore calls $f$ with a series of pointer arguments specialized to the types and locations of object $o$'s fields. (Reference $o$ is used only for pointer arithmetic, so need not be an actual object instance.)

The specialized polymorphic function can then iterate over its type parameters using SFINAE programming idioms [59]. For example, Listing 9 uses the template to generate OFI mediator code to secure a security-relevant structure argument to an API function. Lines 1–2 first prototype a generic recursive template function that will recurse over all fields of an arbitrary structure. Lines 4–5 define the base case of zero fields. Lines 7–11 implement the particular case of enforcing the contract for a field of type `FARPROC` (i.e., generic function pointer field). (This is just one representative case; the full implementation has cases for all the types in Figure 3.) Code in line 9 treats argument `field1` as an index into the object layout where the field resides at runtime on the stack or heap.

Line 13 demonstrates specializing the generic template to a particular class type. The `Reflect` template sketched in Listing 8 is applied to specialize the generic `enforce_contract` template. This allows mediation code for tens of thousands of API methods to be automatically synthesized from just a few hundred lines of hand-written template code, keeping OFI's trusted computing base relatively small and manageable.

## 5 EVALUATION

Performance evaluation of OFI on CPU benchmarks (e.g., SPEC CPU2006) exhibits no measurable overhead because CPU benchmarks do not typically access object-oriented system APIs within loops, which is where OFI introduces overhead. To evaluate the effectiveness of OFI, we therefore tested our prototype with the set of binaries listed in Table 1. The test binaries were chosen to be small and simple enough to be amenable to fully automated binary reverse engineering and instrumentation (whose efficacy is orthogonal to OFI), yet reliant upon large, complex system APIs representative of typical consumer software (and therefore an appropriate test of our approach's practical feasibility). All experiments detailed below were performed on an Intel Xeon E5645 workstation with 24 GB RAM running 64-bit Windows 7. We have no source code for any of the test binaries.

Column 2 reports a count of the total number of libraries loaded (statically and dynamically) by each test program, and column 3 reports a count of all methods exported by those libraries. On average, each program loads 12 libraries that export about 7,500 trusted methods. Taking these statistics into consideration, although the test binaries are small-to-moderate in size, the trusted interfaces that must be supported to accommodate them are large. In total, we need to mediate the interfaces of 54 trusted system libraries that collectively expose 18,059 trusted methods, many of which have challenging method signatures involving code pointers, recursive types, class subtyping, dependent types, and object (or object-like) data structures.

### 5.1 Transparency

Without OFI extensions, none of the test programs ran correctly after CFI instrumentation. All COM-dependent operations—including dialog boxes, certain menus, and in some cases even application start-up—failed with a control-flow violation.

After adding OFI to the instrumentation, we manually tested all program features systematically. All features we tested exhibited full functionality. While we cannot ensure that such testing is exhaustive, we consider it similar to the level of quality assurance to which such applications are typically subjected prior to release.

**Table 1: Interactive COM applications used in experimental evaluation**

| Binary Program | # DLLs | # Interface Funcs | File Size Old (KB) | File Size New (KB) | File Size Increase (%) | Code Segment Size Old (KB) | Code Segment Size New (KB) | Code Segment Size Increase (%) | Rewriting Times (s) |
|---|---|---|---|---|---|---|---|---|---|
| calc | 17 | 11,755 | 758 | 1,263 | 67 | 330 | 514 | 56 | 19.00 |
| cmd | 8 | 7,321 | 296 | 521 | 76 | 139 | 225 | 62 | 5.85 |
| explorer | 27 | 15,324 | 2,555 | 3,611 | 41 | 701 | 1,056 | 51 | 29.60 |
| magnify | 16 | 14,073 | 615 | 751 | 22 | 91 | 136 | 50 | 4.74 |
| MCFG_Exploit | 7 | 2,074 | 11 | 34 | 209 | 4 | 23 | 475 | 0.89 |
| minesweeper | 8 | 6,560 | 117 | 153 | 31 | 16 | 35 | 119 | 1.08 |
| notepad | 14 | 10,441 | 176 | 248 | 41 | 43 | 72 | 67 | 2.21 |
| osk | 19 | 13,662 | 631 | 849 | 35 | 147 | 218 | 48 | 7.10 |
| powershell | 9 | 6,318 | 442 | 489 | 11 | 36 | 47 | 31 | 2.32 |
| solitaire | 8 | 6,379 | 56 | 109 | 95 | 24 | 54 | 125 | 1.44 |
| WinRAR | 17 | 7,536 | 1,374 | 2,928 | 113 | 1,008 | 1,554 | 54 | 70.92 |
| wmplayer | 9 | 6,997 | 161 | 186 | 16 | 12 | 25 | 108 | 0.84 |
| ***median*** | 12 | 7,429 | 369 | 505 | 41% | 67 | 104 | 59% | 3.53s |

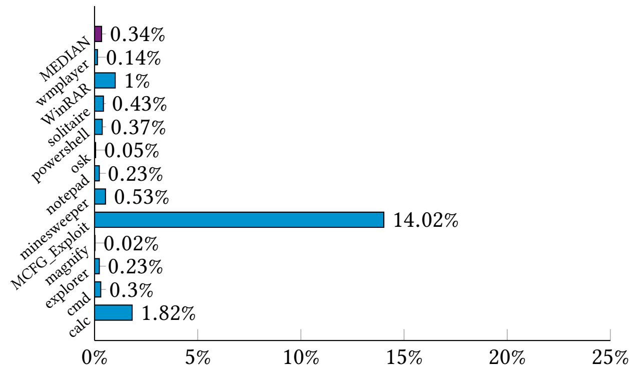

**Figure 7: OFI runtime overhead**

## 5.2 Performance Overheads

**Rewriting Time and Space Overheads.** Table 1 reports the percentage increase of the file size and code segment, as well as the time taken by OFI to rewrite each binary. Our prototype rewrites about 60KB of code per second on average. A rewritten binary increases in size by about 41%. Code segment sizes increase by about 59%. The large percentage increases exhibited by the MCFG_Exploit experiment (209% and 475%, respectively) are artifacts of the exceptionally small size of that program. (It is the synthetic MCFG exploit test reported in Section 2.2.)

**Runtime Performance.** Figure 7 reports OFI runtime overheads of the programs in Table 1. Since almost all object exchanges occur during application startup and in response to user events (e.g., mouse clicks), we created macros that open, manipulate, and close each test program as rapidly as possible. By running such a simulation in a loop for 1000 iterations, we obtain an average running time. We measure the runtime overhead imposed by OFI as the ratio of time spent within the OFI modules to the total runtime.

The median overhead is 0.34%; and no program has overhead larger than 2.00%, except for MCFG_Exploit—our proof-of-concept CODE-COOP exploit implementation. Its size is small, and its only runtime operation initializes a COM object, which involves OFI mediation, resulting in abnormally high percentage overheads. The remaining tests are common consumer apps. Of these, the calculator program returns the worst overhead of 1.82%. This is due to the fact that switching the calculator's mode between standard, scientific, programmer, and statistics requires frequent OFI mediation. Each such switch reconstructs the GUI via 3,500 method calls that involve shared code and/or object pointers, and thus OFI mediation. Nevertheless, we consider the 1.82% overhead to be modest and unnoticeable by users. All other test programs have runtime overheads below 1%, and the calculator's <2% worst-case overhead only occurs on mode changes.

The performance overheads reported in Figure 7 attempt to measure semi-realistic usage scenarios for user-interactive applications, which tend to be the ones that use COM the most. However, to derive a worst-case performance bound for OFI, we also created a set of micro-benchmarks. Each implements a non-interactive program that creates, manipulates, and destroys COM objects in a tight loop. Technical details for each benchmark are provided in Table 2. Although not realistic, these tests can measure the extreme worst-case scenario that a program constantly crosses the trust boundary without performing any other computation, triggering OFI mediation continuously.

Micro-benchmarking yielded a median overhead of 32.44%, with a maximum of just over 50%. We know of no realistic application that would exhibit these overheads in practice, but they reveal the overhead of instrumentation relative to the non-instrumented inter-module control-flow paths.

## 5.3 Security Evaluation

To assess OFI's response to attacks, we launched synthetic vtable corruption and COOP attacks against some programs rewritten by our prototype. We simulate COOP attacks by temporarily modifying the v-vault dispatcher to occasionally choose the wrong vaulter. This simulates a malicious caller who crafts a counterfeit object whose vtable pointer identifies a structurally similar (e.g., similarly typed) vtable but not the correct one.

**Table 2: Micro-benchmark overheads**

| No. | Description | Interfaces | Functions | Overhead |
|---|---|---|---|---|
| #1 | (1) creates object,<br>(2) destroys object | IUnknown | ::Release() | 50.67% |
| #2 | (1) creates object,<br>(2) raises and lowers ref count,<br>(3) destroys object | IUnknown | ::AddRef(), Release() | 41.30% |
| #3 | (1) creates open dialog object,<br>(2) adds controls,<br>(3) destroys object | IFileOpenDialog<br>IFileDialogCustomize | ::QueryInterface(), Release()<br>::AddPushButton(), AddMenu(), AddText(), AddControlItem(), Release(), | 41.69% |
| #4 | (1) creates open dialog object,<br>(2) binds file to shell object,<br>(3) retrieves file path,<br>(4) destroys all objects | IFileOpenDialog<br><br>IShellItem | ::SetFileName(), Show(), GetResult(), Release()<br>::GetDisplayName(), Release() | 32.22% |
| #5 | (1) creates open dialog object,<br>(2) binds files to array,<br>(3) binds elements to shell objs,<br>(4) retrieves the file paths,<br>(5) destroys all objects | IFileOpenDialog<br><br><br>IShellItemArray<br><br>IShellItem | ::GetOptions(), SetOptions(), SetFileName(), Show(), GetResults(), Release()<br>::GetCount(), GetItemAt(), Release()<br>::GetDisplayName(), Release() | 32.44% |
| #6 | (1) creates open dialog object,<br>(2) binds file to shell object,<br>(3) creates save dialog object,<br>(4) sets save-as default,<br>(5) binds saved file to new shell obj,<br>(6) retrives path of new shell obj,<br>(7) destroys all objects | IFileOpenDialog<br><br>IShellItem<br>IFileSaveDialog | ::SetFileName(), Show(), GetResult(), Release()<br>::GetDisplayName(), Release()<br>::SetSaveAsItem(), SetFileName(), Show(), GetResult(), Release() | 31.76% |
| #7 | (1) creates save dialog object,<br>(2) binds saved file to shell obj,<br>(3) retrieves the file path,<br>(4) creates shell link object,<br>(5) sets path as link target,<br>(6) saves link in persist storage,<br>(7) destroys all objects | IFileSaveDialog<br><br>IShellLink<br><br>IPersistFile | ::SetFileName(), Show(), GetResult(), Release()<br>::SetPath(), SetDescription(), QueryInterface(), Release()<br>::Save(), Release() | 31. 69% |

**Table 3: Attack simulation results**

| | | Security Aborts | | |
|---|---|---|---|---|
| Binary Program | # Attacks | Within Callee | After Return | Within OFI |
| calc | 5 | 1 | 4 | 0 |
| MCFG_Exploit | 1 | 0 | 0 | 1 |
| notepad | 5 | 0 | 5 | 0 |
| powershell | 5 | 1 | 4 | 0 |
| WinRAR | 5 | 3 | 2 | 0 |

Table 3 reports the attack simulation results. Each program in column 1 is exposed to 5 attacks. In each case, the attack quickly results in a security abort and premature termination; no control-flow policy violations were observed. Among the 20 attacks, the callee aborted in 5 cases (column 3), and the caller aborted after return in 15 cases (column 4).

Most of the security aborts take the form of SFI memory access rejections (e.g., when an untrusted caller attempts to write to an SFI-protected, callee-owned object). This is because OFI ensures that even if an incorrect vaulter is chosen, control still flows to a vaulter that enforces the contract demanded by its callee, and therefore the callee does not receive any policy-violating objects or code pointers. The callee might nevertheless receive incorrect (but not policy-violating) arguments, such as data pointers into inaccessible memory. In such cases, the callee safely aborts with a memory access violation. Other times the callee runs correctly but returns data or code pointers not expected by the caller, whereupon CFI or SFI protections on the caller side intervene.

The MCFG_Exploit attack (see Section 2.2) is detected within the OFI vaulter code when OFI identifies the counterfeit vtable.

## 5.4 Scalability

To exhibit OFI's scalability, we applied our prototype to Mozilla Firefox (version 48.0.1) for Windows, which is larger and more complex than our other test applications in Table 1. Like many large software products, Firefox is heavily multi-module—most of its functionalities are implemented in whole or part within application-level DLLs that ship along with the main executable. Applying

**Table 4: Browser experimental results**

| Binary Program | # DLLs | # Interface Funcs | File Size Old (KB) | File Size New (KB) | File Size Increase (%) | Code Segment Size Old (KB) | Code Segment Size New (KB) | Code Segment Size Increase (%) | Rewriting Times (s) |
|---|---|---|---|---|---|---|---|---|---|
| firefox.exe | 1 | 1,393 | 376 | 522 | 39 | 80 | 146 | 82 | 15.36 |
| browsercomps.dll | 8 | 7,611 | 43 | 100 | 133 | 28 | 57 | 103 | 5.15 |
| freebl3.dll | 3 | 4,166 | 329 | 688 | 109 | 236 | 359 | 52 | 41.77 |
| lgpllibs.dll | 2 | 3,359 | 50 | 110 | 120 | 36 | 59 | 64 | 6.14 |
| mozglue.dll | 3 | 3,374 | 104 | 238 | 129 | 84 | 134 | 59 | 15.33 |
| msvcp120.dll | 2 | 3,359 | 429 | 848 | 98 | 392 | 418 | 7 | 126.21 |
| nss3.dll | 5 | 4,422 | 1,662 | 3,972 | 139 | 1,360 | 2,310 | 70 | 242.72 |
| nssdbm3.dll | 2 | 3,359 | 84 | 216 | 157 | 72 | 131 | 83 | 14.27 |
| nssckbi.dll | 2 | 3,359 | 386 | 469 | 22 | 40 | 82 | 105 | 9.29 |
| sandboxbroker.dll | 5 | 5,193 | 198 | 381 | 92 | 100 | 182 | 82 | 20.06 |
| softokn3.dll | 2 | 3,359 | 137 | 339 | 147 | 112 | 202 | 81 | 20.67 |
| xul.dll | 36 | 12,657 | 51,251 | 104,116 | 103 | 31,184 | 52,865 | 70 | 5,662.68 |
| ***median*** | 3 | 3,367 | 264 | 425 | 115% | 92 | 164 | 75% | 17.71*s* |

OFI merely to firefox.exe hence does not provide much security. We therefore treated all modules in Table 4 as untrusted for this experiment. Similar to Table 1, column 2 in Table 4 counts the number of trusted libraries imported by each module, and column 3 counts the methods exported by each library. The other columns in Table 4 report file size increase, code segment size increase, and the time that OFI took to rewrite each module. On average, file sizes increase by about 115%, and code segments by about 75%.

One problem that we encountered was that Firefox's Just-In-Time (JIT) JavaScript compiler performs runtime code generation, which our Reins prototype does not yet support. (It conservatively denies execution access to writable memory.) Future work should overcome this by incorporating a CFI-supporting JIT compiler, such as RockJIT [45]. As a temporary workaround, for this experiment we installed a vectored exception handler that catches and redirects control-flows to/from runtime-generated code through OFI. This is potentially unsafe (because the runtime-generated code remains uninstrumented by CFI) and slow (because exception handling introduces high overhead), but allowed us to test preservation of Firefox's functionalities in the presence of OFI. All browser functionalities we tested exhibited full operation after OFI instrumentation.

To estimate the performance impact of OFI on the application, we conducted the same evaluation methodology as reported in Section 5.2, but subtracted out the overhead of the extra context-switches introduced by the exception handler. This yields an estimated overhead of about 0.84%.

## 6 RELATED WORK

### 6.1 SFI and CFI

SFI was originally conceived as a means of sandboxing untrusted software modules via software guards to a subset of a shared address space [61]. CFI refined this idea to enforce more specific control-flow graphs (CFGs) [1, 2]. Later work merged the two approaches for more efficient enforcement [4, 20, 37, 43, 64, 65, 69], so that today distinctions between SFI and CFI are blurred. We therefore here refer to CFI in a broad sense that includes both lines of research.

With the rise of return-oriented programming and code-reuse attacks (cf., [11, 52]), the impact of CFI research has increased in recent years. In addition to securing user-level application software against such threats, it has also been applied to harden smartphones [14, 39, 49], embedded systems [3], hypervisors [63], and operating system kernels [13, 25, 33]. CFI-enforcing hardware is also being investigated [15, 16, 18, 24, 28, 42, 68, 70].

Software CFI methodologies can be broadly partitioned into compiler-side *source-aware* approaches and binary-only *source-free* approaches. Source-aware CFI leverages information from source code to generate CFI-enforcing object code via a compiler. Examples include WIT [4], NaCl [69], CFL [5], MIP [43], MCFI [44], RockJIT [45], Forward CFI [57], CCFI [36], $\pi$CFI [46], and MCFG [55]. The availability of source code affords these approaches much greater efficiency and precision than source-free alternatives. For example, source code analysis typically reveals a much more precise CFG for CFI to enforce, and compilers enjoy opportunities to arrange data structure and code layouts to optimize CFI guard code.

MCFI highlights the need for better multi-module CFI enforcement algorithms and tools. To address this problem in source-aware settings, it introduces a modular, separate-compilation approach integrated into the LLVM compiler. However, this requires all modules to be recompiled with an MCFI-equipped compiler; environments where some modules are immutable, are dynamically procured in binary form, or are closed-source, are not supported.

In general, reliance on source code has the potential disadvantage of reducing deployment flexibility. Much of the world's software is closed-source, with source-level information unlikely to be disclosed to consumers due to intellectual property concerns and constraints imposed by developer business models. Software whose sources are available frequently link to or otherwise rely upon binary modules (e.g., libraries) whose sources are not available, requiring approaches for dealing with those source-free components. Finally, software distribution models that deliver binary code on-demand (e.g., as plugins, mobile apps, or hotpatches) usually lack readily available source code with which to implement additional third-party or consumer-side CFI protections.

Concerns over this inflexibility have therefore motivated source-free CFI approaches that transform and harden already-compiled binary code without the aid of source code. Examples include XFI [20], Reins [65], STIR [64], CCFIR [73], bin-CFI [74], BinCC [62], Lockdown [48] TypeArmor [58] and OCFI [40]. Source-free approaches face some difficult challenges, including the problem of effectively disassembling arbitrary native code binaries [66], and severe restrictions on which code and data structures they can safely transform without breaking the target program's functionality. Poorer performance than source-aware solutions typically results [7]. They also tend to enforce more permissive control-flow policies, since they lack source-level control-flow semantics with which to craft a tighter policy [54]. This has led to successful attacks against these coarse-grained policies (e.g., [9, 17, 26, 67]).

In contrast to this usual dichotomy, OFI is source-agnostic—it can extend any of the source-aware or source-free approaches listed above to enhance the security and compatibility of objects that flow between CFI-protected software modules and those lacking CFI protections. It does, however, require documentation of the API that links the interacting modules, as described in §3.2.

## 6.2 VTable Protection

VTable protections prevent or detect vtable corruption at or before control-flow operations that depend on vtable method pointers. Like CFI, there are both source-aware and source-free approaches:

On the source-aware side, GNU VTV [56], SafeDispatch [32], and VTrust [71] statically analyze source code class hierarchies to generate CFI-style guards that restrict all virtual method call sites to destinations that implement matching callees (according to C++ dynamic dispatch semantics). OVT-IVT [6] improves performance by reorganizing vtables to permit quick validation as a simple bounds check. CPI [34] heuristically derives a set of sensitive pointers, and guards their integrity to prevent control-flow hijacking. CPS [34] optimizes CPI to improve overheads for programs with many virtual functions by instrumenting only code pointers, but at the expense of less security for vtable pointers exploited by confused deputy attacks. Readactor++ [11] extends vtables into execute-only memory, where their layouts are randomized and laced with booby trap entries [10] to counter brute-force attacks. Shrinkwrap [29] refines VTV for tighter object inheritance precision.

On the source-free side, T-VIP [23] instruments virtual call sites with guard code that verifies that the vtable is in read-only memory and that the indexed virtual method is a valid virtual method pointer. VTint [72] additionally assigns them IDs that are dynamically checked at call sites. This ensures that instrumented virtual calls always index a valid vtable (though it cannot ensure that the indexed vtable is the precise one demanded by the original source code semantics). VfGuard [50] goes further and infers C++ class hierarchies and call graphs from native code through a suite of decompilation techniques. This yields a more precise, source-approximating CFI policy that can be enforced through static or dynamic binary instrumentation.

OFI differs from these approaches by focusing on protecting software modules that cannot be instrumented (e.g., because they cannot be modified, they have defenses that reject modification, or dynamic loading prevents them from being statically identified). Such immutability renders the vtable protections above inapplicable, since they must instrument all call sites where corrupted vtables might be dereferenced in order to be effective.

## 6.3 COOP Attacks

COOP [54] is a dangerous new attack paradigm that substitutes vtable pointers or vtable method pointers with structurally similar but counterfeit ones to hijack control-flows of victim programs. In the context of CFI-protected software, such attacks effectively hijack software without violating the CFI-enforced control-flow policy. They achieve this by traversing control-flow edges that are permitted by the policy but that were never intended to be traversed by the original program semantics. They therefore exploit limitations in the defender's ability to derive suitable policies for CFI to enforce—especially in source-free contexts.

OFI does not directly defend against COOP attacks because it does not suggest better policies for CFI to enforce. Rather, it extends defenses that do work against COOP to be effective in contexts where not all call sites can be instrumented with guard code. For example, WIT [4] can block COOP attacks in WIT-instrumented code, but not if the code links to uninstrumented modules to which it passes objects. In that context, a COOP attacker can flow counterfeit objects to unguarded call sites in the uninstrumented modules. Lacking guards, these sites traverse the prohibited edge prescribed by the object, resulting in policy violations.

When coupled with a CFI defense enforcing a suitably semantics-aware policy, OFI addresses this CODE-COOP attack. By completely mediating the interface between guarded and unguarded modules that share objects, it shields uninstrumented modules from counterfeit objects. OFI is the first defensive work to focus on this attack class.

## 6.4 Immutable Modules

We are not the first to identify immutable modules as a challenge for CFI. For example, source-aware CFI instrumentation of Chrome on ChromeOS identified two third-party libraries for which source code was not available, and that interact with instrumented modules through object-oriented interfaces [57]. Forward CFI's solution to this *mixed code* problem validates object references at call sites within instrumented modules. But this is insecure if the uninstrumented recipients retain persistent references to the shared objects, or if they execute concurrently with untrusted (instrumented) code. In both cases, the untrusted code may later corrupt the shared vtable pointers without calling them, leaving the uninstrumented module in possession of a corrupt, never-validated vtable.

In general, all prior source-aware and source-free CFI and vtable protection research must instrument all interoperating modules in order to thwart control-flow hijacking attacks. OFI is the first solution that accommodates immutable modules. In deployment contexts where the OS cannot be included in the instrumentation process, such modules can be extremely prevalent—potentially including most or all of the system libraries, plus an ongoing stream of incoming upgrades, patches, and extensions to them. OFI seeks to open such environments to CFI assistance.

## 6.5 Component-based Software Engineering

Microsoft COM [27] is presently the dominant industry standard for *component-based software engineering* [38] of native code modules in consumer software markets. Its many facets include Object Linking and Embedding (OLE), ActiveX, COM+, Distributed COM

(DCOM), DirectX, User-Mode Driver Framework (UMDF), and the Windows Runtime (WinRT). Microsoft .NET applications typically access Windows OS services via the .NET COM Interop, which wraps COM. This prevalence makes COM an appropriate (but challenging) test of OFI's real-world applicability.

Another primary competing standard is OMG's Common Object Request Broker Architecture (CORBA) [60]. CORBA resembles COM but enforces additional layers of abstraction, including an Object Request Broker (ORB) that has the option of supplying different representations of shared objects to communicating modules. OFI is therefore potentially easier to realize for CORBA than for COM, since it can take the role of a CFI-enforcing ORB. Interfaces that communicate between CORBA and COM have also been developed [47].

## 7 CONCLUSION

OFI is the first work to extend CFI security protections to the significant realm of mainstream software in which one or more object-exchanging modules are immune to instrumentation. It does so by mediating object exchanges across inter-module trust boundaries with the introduction of tamper-proof proxy objects. The mediation strategy is source-agnostic, making it applicable to both source-aware and source-free CFI approaches. A type-theoretic basis for the mediation algorithm allows for automatic synthesis of OFI mediation code from interface description languages.

A prototype implementation of OFI for Microsoft COM indicates that the approach is feasible without access to source code, and scales to large interfaces that employ callbacks, event-driven programming, interface inheritance, datatype recursion, and dependent typing. Experimental evaluation shows that OFI exhibits low overheads of under 1% for some real-world consumer software applications.

## ACKNOWLEDGMENTS

The research reported herein was supported in part by NSF awards #1513704 and #1054629, ONR award N00014-14-1-0030, AFOSR award FA9550-14-1-0173, and an NSF-supported I/UCRC award from Raytheon Company. Any opinions, recommendations, or conclusions expressed are those of the authors and not necessarily of the aforementioned organizations.

## REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*. 340–353.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009).

[3] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proceedings of the 23rd ACM Conference on Computer and Communications and Security (CCS)*.

[4] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*. 263–277.

[5] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented Programming: A New Class of Code-reuse Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 30–40.

[6] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*.

[7] Nathan Burow, Scott A. Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. 2016. Control-Flow Integrity: Precision, Security, and Performance. *CoRR* abs/1602.04056 (2016).

[8] Matúš Chochlík and Axel Naumann. 2016. Static Reflection (revision 4). C++ Standards Committee Paper P0194R0. (2016).

[9] Mauro Conti, Stephen J. Crane, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications and Security (CCS)*. 952–963.

[10] Stephen J. Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Booby Trapping Software. In *Proceedings of the 2013 on New Security Paradigms Workshop (NSPW)*. 95–106.

[11] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications and Security (CCS)*. 243–255.

[12] Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *Proceedings of the 20th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 50–63.

[13] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*. 292–307.

[14] Lucas Davi, Ra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. MoCFI: A Framework to Mitigate Control-flow Attacks on Smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*.

[15] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-Assisted Flow Integrity eXtension. In *Proceedings of the 52th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[16] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. 2014. Hardware-assisted Fine-grained Control-flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Proceedings of the 51th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[17] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium*. 401–416.

[18] Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen De Bosschere, Bart Preneel, Bjorn De Sutter, and Ingrid Verbauwhede. 2016. SOFIA: Software and Control Flow Integrity Architecture. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1172–1177.

[19] Joe Duffy. 2008. *Concurrent Programming on Windows*. Addison-Wesley.

[20] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 75–88.

[21] Chris Exton, Damien Watkins, and Dean Thompson. 1997. Comparisons Between CORBA IDL & COM/DCOM MIDL: Interfaces for Distributed Computing. In *Proceedings of the 25th Technology of Object-Oriented Languages and Systems Conference (TOOLS)*. 15–32.

[22] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 48–59.

[23] Robert Gawlik and Thorsten Holz. 2014. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*. 396–405.

[24] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 585–598.

[25] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*. 179–194.

[26] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*. 575–589.

[27] David N. Gray, John Hotchkiss, Seth LaForge, Andrew Shalit, and Toby Weinberg. 1998. Modern Languages and Microsoft's Component Object Model. *Communications of the ACM (CACM)* 41, 5 (1998), 55–65.

[28] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 173–184.

[29] István Haller, Enes Göktas, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. ShrinkWrap: VTable Protection without Loose Ends. In *Proceedings of the 31th Annual Computer Security Applications Conference (ACSAC)*. 341–350.

[30] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. 2006. Computability Classes for Enforcement Mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 1 (2006), 175–205.

[31] Norm Hardy. 1988. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *ACM SIGOPS Operating Systems Review* 22, 4 (1988), 36–38.

[32] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*.

[33] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proceedings of the 21st USENIX Security Symposium*. 459–474.

[34] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–163.

[35] Jay Ligatti, Lujo Bauer, and David Walker. 2009. Run-time Enforcement of Nonsafety Policies. *ACM Transactions on Information and Systems Security (TISSEC)* 12, 3 (2009).

[36] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. 941–951.

[37] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15st USENIX Security Symposium*.

[38] M.D. McIlroy. 1968. Mass Produced Software Components. In *Proceedings of the NATO Software Engineering Conference*, Peter Naur and Brian Randell (Eds.). 138–156.

[39] Castro Miguel, Costa Manuel, Martin Jean-Philippe, Peinado Marcus, Akritidis Periklis, Donnelly Austin, Barham Paul, and Black Richard. 2009. Fast Byte-granularity Software Fault Isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 45–58.

[40] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*.

[41] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: Compiler-Enforced Temporal Safety for C. In *Proceedings of the 9th International Symposium on Memory Management (ISMM)*. 31–40.

[42] Christoulakis Nick, Christou George, Athanasopoulos Elias, and Ioannidis Sotiris. 2016. HCFI: Hardware-enforced Control-Flow Integrity. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 38–49.

[43] Ben Niu and Gang Tan. 2013. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. 199–210.

[44] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 577–587.

[45] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-in-time Compilation Using Modular Control-flow Integrity. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. 1317–1328.

[46] Ben Niu and Gang Tan. 2015. Per-Input Control-flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. 914–926.

[47] M.K. Pawar, Ravindra Patel, and N.S. Chaudhari. 2013. Interoperability Between .Net Framework and Python in Component Way. *International J. of Computer Science Issues (IJCSI)* 10, 1 (2013), 165–170.

[48] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 144–164.

[49] Jannik Pewny and Thorsten Holz. 2013. Control-flow Restrictor: Compiler-based CFI for iOS. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*. 309–318.

[50] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*.

[51] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security (TISSEC)* 15, 1 (2012).

[52] Ahmad-Reza Sadeghi, Lucas Davi, and Per Larsen. 2015. Securing Legacy Software against Real-World Code-Reuse Exploits: Utopia, Alchemy, or Possible Future?. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 55–61.

[53] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Transactions on Information and Systems Security (TISSEC)* 3, 1 (2000), 30–50.

[54] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*. 745–762.

[55] Jack Tang. 2015. *Exploring Control Flow Guard in Windows 10*. Technical Report. Trend Micro Threat Solution Team.

[56] Caroline Tice. 2012. Improving Function Pointer Security for Virtual Method Dispatches. In *GNU Cauldron Work*.

[57] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*. 941–955.

[58] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*. 934–953.

[59] David Vandevoorde and Nicolai M. Josuttis. 2002. *C++ Templates: The Complete Guide*. Addison-Wesley.

[60] Steve Vinoski. 1997. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine* 35, 2 (1997), 46–55.

[61] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*. 203–216.

[62] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. 2015. Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*. 331–340.

[63] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*. 380–395.

[64] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. 157–168.

[65] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Securing Untrusted Code via Compiler-Agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. 299–308.

[66] Richard Wartell, Yan Zhou, Kevin W. Hamlen, and Murat Kantarcioglu. 2014. Shingled Graph Disassembly: Finding the Undecidable Path. In *Proceedings of the 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. 273–285.

[67] Patrick Wollgast, Robert Gawlik, Behrad Garmany, Benjamin Kollenda, and Thorsten Holz. 2016. Automated Multi-architectural Discovery of CFI-Resistant Code Gadgets. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS)*. 602–620.

[68] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)*. 1–12.

[69] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*. 79–93.

[70] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. 2015. Hardware-assisted Fine-grained Code-reuse Attack Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 66–85.

[71] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. 2016. VTrust: Regaining Trust on Virtual Calls. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*.

[72] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*.

[73] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zo. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*. 559–573.

[74] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Conference on Security (USENIX)*. 337–352.