Towards Interface-Driven COTS Binary Hardening

Xiaoyang Xu
Wenhao Wang
Kevin W. Hamlen
The University of Texas at Dallas
Computer Science Department
Richardson, TX, USA

Zhiqiang Lin
The Ohio State University
Computer Science & Engineering Department
Columbus, OH, USA

ABSTRACT

Hardening COTS binary software products (e.g., via control-flow integrity and/or software fault isolation defenses) is particularly difficult in contexts where the surrounding software environment includes closed-source, unmodifiable, and possibly obfuscated binary components, such as system libraries, OS kernels, and virtualization layers. It is demonstrated that many code hardening algorithms, when applied only to the user-level software products in such environments, leave open critical vulnerabilities that arise from mismatches between the application-agnostic security policies enforced by the system modules versus the application-specific policies enforced at the application layer.

To overcome this problem, a modular approach is proposed for hardening application-level software in such environments without the need to harden all other software in the environment with exactly the same protection strategy or policies. The approach embeds application-level protections within objects shared by interoperating modules. Modules that obey their interface specifications therefore receive an appropriate granularity of protection automatically when they invoke shared object methods. Experiences developing and refining this approach for Microsoft Windows environments are reported and discussed.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Software reverse engineering*;

KEYWORDS

control-flow integrity, object-oriented software, component-based software

ACM Reference Format:
Xiaoyang Xu, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. 2018. Towards Interface-Driven COTS Binary Hardening. In *The 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3273045.3273051

FEAST '18, October 19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5997-9/18/10…$15.00
https://doi.org/10.1145/3273045.3273051

1 INTRODUCTION

Hardening binary software applications against low-level exploits (e.g., control-flow hijacking and code reuse attacks [5, 18]) is widely recognized as an important step in defending software ecosystems. Software Fault Isolation (SFI) [23] and Control-Flow Integrity (CFI) [1] are two important examples of such hardening. Implementation approaches include XFI [6], PittSFIeld [12], Reins [27], STIR [26], CCFIR [30], bin-CFI [31], BinCC [24], Lockdown [16] TypeArmor [22] and OCFI [13]. However, most hardening techniques in the literature assume that interoperating software components are all hardened in the same way, using the same code transformation algorithm. For example, XFI's binary transformation entails instrumenting all reachable control-flow transfer instructions in all modules with guard code that checks for XFI-added security labels at jump destinations. This uniformity of enforcement is a prerequisite assumption of XFI's proof of safety [2].

VTable protections, which include source-aware [3–5, 8, 10, 11, 20, 28] and source-free [7, 17, 29] approaches for preventing or detecting vtable corruption at control-flow operations involving vtable method pointers, likewise typically require that all call sites where such pointers might be dereferenced must be uniformly instrumented with common guard code in order to be effective. If some pointers flow to call sites located within other modules compiled with a different pointer protection mechanism, control-flow security cannot be guaranteed.

Unfortunately, a large number of mission-critical software environments include diverse, interoperating components that are not all secured in exactly the same way. For example, the user interfaces of many critical infrastructure applications are implemented atop Microsoft Windows OSes, which purvey essential services to binary applications via closed-source, binary system libraries. These libraries are difficult to modify for a variety of reasons: some are digitally signed, others are aggressively optimized in ways that frustrate accurate disassembly even by the best reverse-engineering tools, and some are loaded dynamically (e.g., from cloud services) as applications execute and discover they need particular services. Similarly, many event-driven Linux applications are implemented atop toolkits such as GTK+[1], which dynamically serve user interface widgets and supporting library code on-demand, and which therefore may have been separately compiled with a diverse variety of different protection strategies.

Although recompiling the universe of all software components with some uniform protection scheme is obviously one option for coping with this problem, doing so is unrealistic for many operating contexts. This motivates the development of a more modular

[1] https://www.gtk.org

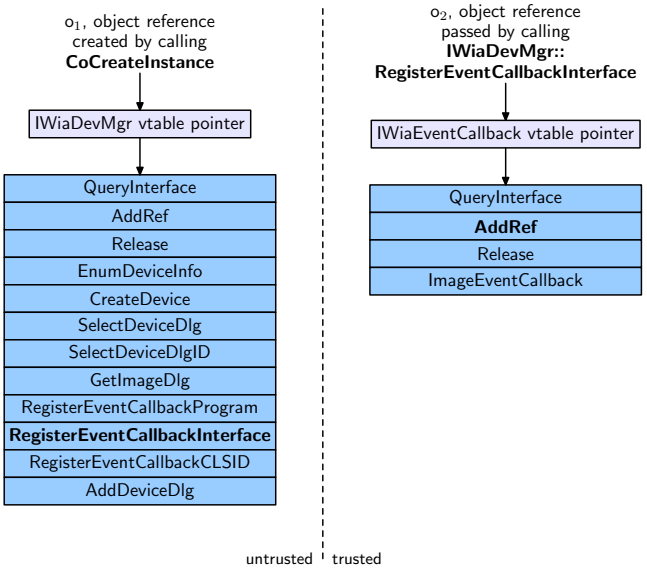

Figure 1: Object binary representation

```
Untrusted Module
1  CoCreateInstance(⟨clsid⟩, . . . , ⟨iid1⟩, &o1);
2  o1→RegisterEventCallbackInterface(. . ., o2, . . .);
                                        Trusted Module
3                                       o2→AddRef();
```

Listing 1: Code that registers a running application Windows Image Acquisition (WIA) event notification

```
1  MOV EAX, ⟨object⟩
2  MOV ECX, DWORD PTR DS:[EAX]
3  . . .
4  PUSH ⟨arguments⟩
5  . . .
6  PUSH EAX
7  CALL DWORD PTR DS:[ECX + ⟨index⟩]
```

Listing 2: Function call in assembly

methodology for hardening application code that relies on services implemented with diverse protections, but without the need to modify or even disassemble interoperating binary modules on which the application relies.

This paper reports on our experiences with a new interface-driven approach to securing commercial binary software products with component-driven design, and large, object-oriented APIs with thousands of vtable and method exchanges between dissimilar modules. Inspired by *Object Flow Integrity (OFI)* [25], our approach statically synthesizes CFI/SFI-preserving wrapper modules for immutable system modules from their interfaces. This facilitates a stronger form of SFI/CFI protection for COTS binary Windows applications than was previously possible without modifying the OS kernel and system libraries.

The remainder of the paper proceeds as follows: Section 2 demonstrates how applying previously published CFI hardening to application code without applying the same hardening to interoperating system modules results in exploitable critical vulnerabilities. Section 3 summarizes our interface-driven approach for closing such vulnerabilities without modifying system modules, followed by a detailed case-study in Section 4. Section 5 discusses future work directions, and Section 6 concludes.

2 ATTACK EXAMPLE

CFI and SFI binary hardening algorithms typically work by instrumenting all indirect jump sites in the software with guard code that blocks jumps to illegal destinations at runtime. This prevents many forms of control-flow hijacking, including many code-reuse attacks. However, when the enforcement cannot retrofit all modules, jumps in unmodified modules may remain unguarded, or guarded by a different and possibly inconsistent safety mechanism. This becomes problematic when interoperating modules exchange code pointers—a common practice of object-oriented software that shares objects. In such cases, the disparate guard code can fail to enforce the protection scheme expected by cross-module callees.

One approach to this problem is to secure the objects passed to uninstrumented modules at call sites within the instrumented modules (e.g., [21]). 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). These *COnfused DEputy-assisted Counterfeit Object-Oriented Programming* (CODE-COOP) attacks [25] deputize the receiving module [9] into violating the control-flow policy by passing them counterfeit objects [19].

Before a detailed walkthrough of a CODE-COOP attack, we first show how objects are typically exchanged between modules with object-oriented interfaces. Listing 1 provides a code snippet disassembled from a Microsoft Paint binary. For this example, we assume that the Paint application code is untrusted, whereas the system DLLs it loads are trusted. The application code first creates a shared object o_1 (line 1), where ⟨*clsid*⟩ and ⟨iid_1⟩ are numeric identifiers for the desired system class and its `IWiaDevMgr` interface, respectively. Method `RegisterEventCallbackInterface` is then invoked to register a running application Windows Image Acquisition (WIA) event notification (line 2). This method takes argument o_2, which is a pointer to the `IWiaEventCallback` interface that the WIA system uses to send the event notification.

While executing `RegisterEventCallbackInterface`, the trusted system module calls object o_2's `Addref` method (line 3), which increments the reference count for the object. Listing 2 exhibits the code at the assembly level. The object is first moved to register `EAX` (line 1), and its method table is moved to register `ECX` (line 2). Then all arguments are pushed onto the stack (line 4), including the object (line 6). In the end, the corresponding method is called by indexing the method table (line 7).

Our attacker model assumes that untrusted modules might be completely malicious, containing arbitrary native code, but that

they have been transformed by a CFI algorithm into code compliant with the control-flow policy. Unfortunately, the code snippet in Listing 1 is vulnerable to CODE-COOP attack even with CFI protections enabled for the untrusted module. Such protections prevent the function call on line 2 from violating the control-flow policy, but line 3 is not protected in the same way because it resides in an unmodifiable system library. Argument o_2 passed into the trusted module can therefore potentially be corrupted to escape the CFI sandbox.

An object reference o_2 cannot be simply treated as a function pointer (e.g., for a signature check) because the reference points to an object containing a vtable pointer, as illustrated in Figure 1. The vtable stores many method pointers. Some of these methods create and return more objects containing new vtables and method pointers when called, creating a complex web of interconnected code pointer exchanges. Since dynamically generated vtables frequently reside in untrusted, writable memory, a data corruption vulnerability (e.g., buffer overwrite) can potentially replace the vtable of o_2 with a counterfeit one. This malicious replacement can happen after the function signature check (e.g., if the application is multithreaded or the callee retains a persistent reference to the object).

Thus, the counterfeit vtable can reroute object o_2's method `Addref` call to any location specified by the attacker (line 3). The policy mismatch occurs because the destination of the `Addref` call is computed from an untrusted code pointer, but the call site is located in a trusted, unmodifiable system library and cannot be instrumented directly with guard code.

Cross-module control-flow hijacks are recognized as a significant class of code-reuse attacks in practice. For example, they have been leveraged to hijack Chrome from within Google Native Client by exploiting differences between the CFI policies enforced by different interoperating browser modules [15]. Prior work has advanced compiler-side solutions that require recompiling all modules to the same protection strategy [14], while OFI [25] is currently the only proposed binary solution.

Our work is the first to admit and harmonize differing protection strategies through automated binary interface synthesis. The next section proposes a modular, source-free approach to this that avoids directly modifying any trusted modules.

3 TECHNICAL APPROACH

Our proposed solution instruments untrusted application binary code in such a way that trusted callee modules (i.e., potential victim deputies) never receive writable code pointers from untrusted, CFI-protected callers. Placing the entire object in read-only memory is infeasible because objects typically contain writable data adjacent to the vtable pointer, which cannot easily be moved without breaking the application. We therefore instead automatically substitute shared objects with read-only proxy objects when they flow across an inter-module trust boundary. All proxy objects and their vtables inhabit read-only memory so CODE-COOP attacks cannot corrupt proxy vtables.

Instrumented modules retain direct references to the original object, allowing them to write to data fields, but uninstrumented object recipients receive a read-only proxy. This works because modern

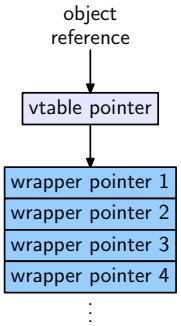

Figure 2: Proxy object binary representation

binary-level object exchange protocols, such as Component Object Model (COM), enforce an abstraction layer that requires object recipients to access data indirectly via accessor methods. (This allows shared objects to be located on remote machines during RPC.) As illustrated in Figure 2, our proxy objects' methods therefore wrap the methods of the underlying object to enforce control-flow guards that intervene whenever object recipients attempt to call one of the object's methods.

For each object-oriented API imported by an untrusted module, we write a wrapper in which every shared object argument is replaced by a proxy object. Thus, when a trusted module attempts to call a method of an object, it actually calls a wrapper method of the proxy object. Control then flows to a dispatch subroutine. The dispatcher pops the return address to determine the index of the method being called, and consults the stack's *this* pointer to identify the object. Based on this information, it selects and tail-calls a mediator that wraps and secures the original method according to its type signature. If original method involves object arguments, the mediator replaces them with corresponding proxies. Finally, the mediator passes the control to the original method.

The wrappers must also sometimes introduce new proxy objects in the reverse direction (i.e., during trusted-to-untrusted cross-module calls and returns) in order to secure methods that return new objects or interfaces. For example, if a trusted callee returns an object whose methods accept objects as arguments, the untrusted caller instead receives a proxy object whose wrappers substitute objects arguments with proxies before passing control back to the trusted module.

To assure complete mediation of these interfaces (which are often large and complex), our approach is conservative: The CFI policy is defined to block all cross-module control-flow edges except the ones implemented by the mediators. Inadvertent omission of an API from the mediator library therefore provokes a security abort at runtime. In practice, the mediator code is synthesized automatically from the interface descriptions (e.g., C header or IDL files), so that all documented interface members are automatically included.

Our approach defends against the attack shown in Section 2. After hardening the code in Listing 1, the shared object o_1 is replaced by its proxy. Original method `RegisterEventCallbackInterface` instead invokes a wrapper method of the proxy object of o_1. This wrapper method reroutes the control to the mediator of `RegisterEventCallbackInterface`. The mediator finds that object o_2

Table 1: Interoperating COM modules used in case study

Module	File Size			Code Segment Size			Rewriting Times (s)
	Old (KB)	**New** (KB)	**Increase** (%)	**Old** (KB)	**New** (KB)	**Increase** (%)	
mspaint.exe	6228	7094	14	557	886	59	104.78
mfc42u.dll	1137	2583	127	1025	1480	44	203.91

Table 2: COM interfaces

Interface	# Methods	DLL
IAccPropServices	12	oleacc
IDataObject	12	ole32
IEnumWIA_DEV_INFO	8	ole32
IMessageFilter	6	ole32
IMarshal	9	ole32
IMoniker	23	ole32
IOleClientSite	9	ole32
IPropertyStore	8	propsys
IRunningObjectTable	10	ole32
IShellFolder	13	shell32
IShellItem	8	shell32
IShellItem2	21	shell32
IStream	14	ole32
IUIApplication	6	uiribbon
IUICollection	10	uiribbon
IUICommandHandler	5	uiribbon
IUIFramework	12	uiribbon
IUIImage	4	uiribbon
IUIImageFromBitmap	4	uiribbon
IUIRibbon	6	uiribbon
IUISimplePropertySet	4	uiribbon
IUnknown	3	uiribbon
IWiaDevMgr	12	wiaservc
IWiaEventCallback	4	ole32
IWICBitmapDecoder	14	windowscodecs
IWICBitmapEncoder	13	windowscodecs
IWICBitmapFrameDecode	11	windowscodecs
IWICBitmapFrameEncode	14	windowscodecs
IWICImagingFactory	28	windowscodecs
IWICMetadataBlockReader	7	windowscodecs
IWICMetadataBlockWriter	12	windowscodecs
IWICStream	18	windowscodecs
IXMLDOMDocument	82	msxml6
IXMLDOMDocument2	88	msxml6

Table 3: APIs with object exchanges

API	DLL	Object Type
CoCreateInstance	ole32	OUT
CoDisconnectObject	ole32	IN
CoGetClassObject	ole32	OUT
CoLockObjectExternal	ole32	IN
CoRegisterMessageFilter	ole32	IN & OUT
CreateFileMoniker	ole32	OUT
CreateStreamOnHGlobal	ole32	OUT
DoDragDrop	ole32	IN
GdipLoadImageFromStream	gdiplus	IN
GdipSaveImageToStream	gdiplus	IN
GetRunningObjectTable	ole32	OUT
OleCreateLinkFromData	ole32	IN & OUT
OleGetClipboard	ole32	OUT
OleSetClipboard	ole32	IN
OleIsCurrentClipboard	ole32	IN
OleIsRunning	ole32	IN
OleRun	ole32	IN
RegisterDragDrop	ole32	IN
SafeArrayPutElement	olaeut32	IN
SHBindToParent	shell32	OUT
SHCreateShellItem	shell32	IN & OUT
SHGetDesktopFolder	shell32	OUT

is passed from the untrusted module to the trusted module. Then the proxy object of object o_2 is generated and handed to the trusted module. Hardware write-protections prevent the proxy object's vtable from being corrupted. Therefore, even without modifying the trusted module, the `AddRef` call is guaranteed to target a permitted destination.

4 CASE STUDY

To demonstrate how our approach can harden closed-source, binary software against CODE-COOP attacks, and to exhibit some of the challenges, we next discuss our experience hardening a simple but representative Windows application: Microsoft Paint.

4.1 Object-oriented Design

Paint is a simple desktop application that has been included with all versions of Windows. Like most commercial software, it does not access system kernel services directly; rather, its design extends system-provided classes to construct objects that inherit kernel-accessing functionalities from their base methods. On Windows, such applications typically draw their base classes from the Microsoft Foundation Class (MFC) Library—a shared C++ library designed for event-driven software development. A large percentage of all Windows software is built atop MFC, but this design presents great challenges for traditional CFI because of the complex object exchanges it engenders at the binary level. Surveys of the prior CFI literature (cf., [25]) exhibit no examples prior to OFI where CFI was successfully evaluated against an MFC-based product without opening CODE-COOP vulnerabilities.

Since MFC is extremely tightly coupled to the applications with which it links, our approach treats both Paint and MFC as untrusted, application-level modules and leaves the others as trusted. To do so, we applied our automated binary retrofitting (built atop the OFI

Table 4: APIs with callback pointers

API	DLL
_beginthread	msvcrt
_beginthreadex	msvcrt
_initterm	msvcrt
_onexit	msvcrt
CallWindowProc	user32
ChooseColor	comdlg32
ChooseFont	comdlg32
DialogBoxIndirectParam	user32
DialogBoxParam	user32
CreateDialogIndirectParam	user32
CreateDialogParam	user32
EnumFonts	user32
EnumFontFamilies	gdi32
EnumFontFamiliesEx	gdi32
EnumObjects	gdi32
EventRegister	advapi32
GetOpenFileName	comdlg32
GetSaveFileName	comdlg32
PrintDlg	comdlg32
RegisterClass	user32
RegisterClassEx	user32
SendMessageCallback	user32
SetAbortProc	gdi32
SetProp	user32
SetWindowLong	user32
SetWindowsHookEx	user32

framework) to the Paint (mspaint.exe) and MFC (mfc42u.dll) binary libraries, and placed the retrofitted MFC in the retrofitted Paint application's load path, thereby overriding the system-level MFC. Table 1 reports the percentage increase of the file size and code segments, as well as the time taken to rewrite each module. After instrumenting, we manually tested all program features of Paint systematically. All features we tested exhibited full functionality. We measure the runtime overhead imposed by our approach as the ratio of time spent within the wrapper modules to the total runtime. Paint has an overhead of 0.38%.

4.2 API Surface

Table 3 lists all the system APIs with object arguments that Paint and MFC called during our experiments. There are 22 APIs from 4 different trusted modules. Column 3 reports the type of object argument in each API. An OUT-object argument (e.g., in `CoCreateInstance`) is usually an interface pointer returned from a trusted module. An IN-object argument (e.g., in `CoLockObjectExternal`) is usually an interface pointer that an untrusted module passes to a trusted module. More complex APIs can have IN-object and OUT-object arguments together. For example, API `SHCreateShellItem` passes an `IShellFolder` interface pointer to shell32.dll and receives an address of a pointer to a `IShellItem` interface after the API returns.

4.3 Object Exchanges

Table 2 reports the interfaces mediated by our guard code when running Paint and MFC. Column 2 reports the number of virtual methods (including inherited methods if any) in the vtable of each interface, and Column 3 reports the trusted module to which the interface belongs. Overall, we mediate 34 interfaces and 510 methods from 8 trusted modules. Among the interfaces and methods in Table 2, Table 5 reports the methods that have object arguments. An object argument can also be an OUT-object or an IN-object, similar to the APIs reported in Table 3.

As discussed in Section 3, for each API and virtual method, we synthesized a mediator in which OFI recursively substitutes both types of object arguments with appropriate proxy objects immediately before the cross-module call and immediately after the cross-module return.

4.4 Callbacks

Table 4 reports the APIs that have code pointers (*callbacks*) as arguments. Such an argument can be a direct code pointer (e.g., in `CallWindowProc`), a pointer to an array of callbacks (e.g., in `initterm`), or a pointer to a structure that has a callback in one or more of its fields (e.g., in `RegisterClass`). Paint and MFC import 14 such APIs from 5 trusted modules. We implemented a mediator for each of these APIs in which code pointer validation or sanitization secures the code pointer exchange against hijacking attacks.

5 FUTURE WORK

Although our approach successfully secures inter-module object exchanges in the presence of unmodifiable (e.g., system) modules, unmodified modules can still potentially contain other security weaknesses that might leave retrofitted applications vulnerable to attack. For example, if a trusted module retains a persistent reference to an object, but stores that reference in an unsafe location (e.g., memory that the retrofitting mechanism considers untrusted and application-writable), then a malicious module could replace the reference with a counterfeit object to implement a CODE-COOP attack despite our defense.

Our current prototype mitigates such vulnerabilities by leveraging software fault isolation (SFI) to isolate module data and stack segments from cross-module writes. However, this approach cannot support modules that need direct access to each other's memory (e.g., when trusted modules store object references into writable buffers provided by untrusted modules).

An important line of future research therefore entails the development of binary-level code analyses and tools that can discover the memory safety policies implicitly expected and enforced by interoperating modules with differing protection schemes. Future work should use such analyses to derive appropriate memory and control-flow safety policies for application-level retrofitting algorithms to enforce in order to ensure safety in the presence of unmodifiable libraries that have differing security expectations and requirements.

6 CONCLUSION

We have presented a modular approach that hardens application-level software without the need to modify interoperating modules on which the application replies. Our interface-driven approach

presented in this paper mediates object exchanges across inter-module trust boundaries with proxy objects, and therefore modules that obeys their interface specifications get protected when they call proxy object methods. We showed that coupled with OFI and CFI, the approach can effectively thwart CODE-COOP attacks by completely mediating the interfaces between trusted and untrusted modules.

ACKNOWLEDGMENTS

The research reported herein was supported in part by ONR awards N00014-14-1-0030 and N00014-17-1-2995, AFOSR award FA9550-14-1-0119, NSF awards #1513704 and #1834215, and an NSF I/UCRC award from Lockheed Martin.

REFERENCES

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

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1), 2009.

[3] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

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

[5] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 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)*, pages 243–255, 2015.

[6] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2006.

[7] Robert Gawlik and Thorsten Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, pages 396–405, 2014.

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

[9] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[10] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, 2014.

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

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

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

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

[15] Jorge Lucangeli Obes and Justin Schuh. A tale of two pwnies (part 1). Chromium Blog, May 2012. https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html.

[16] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 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)*, pages 144–164, 2015.

[17] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[18] Ahmad-Reza Sadeghi, Lucas Davi, and Per Larsen. 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)*, pages 55–61, 2015.

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

[20] Caroline Tice. Improving function pointer security for virtual method dispatches. In *GNU Cauldron Workshop*, 2012.

[21] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, pages 941–955, 2014.

[22] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 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)*, pages 934–953, 2016.

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

[24] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, pages 331–340, 2015.

[25] Wenhao Wang, Xiaoyang Xu, and Kevin W. Hamlen. Object flow integrity. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 1909–1924, 2017.

[26] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, 2012.

[27] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pages 299–308, 2012.

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

[29] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting virtual function tables' integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[30] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zo. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, pages 559–573, 2013.

[31] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, pages 337–352, 2013.

Table 5: Methods with object exchanges

Interface	Method(s)	Object Type
IRunningObjectTable	IRunningObjectTable::Register	IN
	IRunningObjectTable::GetObject	IN & OUT
IShellFolder	IShellFolder::EnumObjects	OUT
IShellItem2	IShellItem2::QueryInterface	OUT
	IShellItem2::GetPropertyStore	OUT
IStream	IStream::QueryInterface	OUT
IUIApplication	IUIApplication::OnViewChanged	IN
	IUIApplication::OnCreateUICommand	OUT
	IUIApplication::OnDestroyUICommand	IN
IUICollection	IUICollection::Add	IN
	IUICollection::GetItem	OUT
	IUICollection::Insert	IN
	IUICollection::Replace	IN
IUICommandHandler	IUICommandHandler::Execute	IN
	IUICommandHandler::UpdateProperty	IN
IUIFramework	IUIFramework::QueryInterface	OUT
	IUIFramework::Initialize	IN
IUIImageFromBitmap	IUIImageFromBitmap::QueryInterface	OUT
	IUIImageFromBitmap::CreateImage	OUT
IUIRibbon	IUIRibbon::LoadSettingsFromStream	IN
	IUIRibbon::SaveSettingsToStream	IN
IUISimplePropertySet	IUISimplePropertySet::QueryInterface	OUT
IUnknown	IUnknown::QueryInterface	OUT
IWiaDevMgr	IWiaDevMgr::RegisterEventCallbackInterface	IN & OUT
IWiaEventCallback	IWiaEventCallback::QueryInterface	OUT
IWICBitmapDecoder	IWICBitmapDecoder::QueryInterface	OUT
	IWICBitmapDecoder::GetFrame	OUT
IWICBitmapEncoder	IWICBitmapEncoder::Initialize	IN
	IWICBitmapEncoder::CreateNewFrame	IN & OUT
IWICBitmapFrameEncode	IWICBitmapFrameEncode::QueryInterface	OUT
	IWICBitmapFrameEncode::Initialize	IN
	IWICBitmapFrameEncode::WriteSource	IN
IWICImagingFactory	IWICImagingFactory::CreateDecoderFromFilename	OUT
	IWICImagingFactory::CreateEncoder	OUT
	IWICImagingFactory::CreateStream	OUT
IWICMetadataBlockReader	IWICMetadataBlockReader::GetReaderByIndex	OUT
IWICMetadataBlockWriter	IWICMetadataBlockWriter::InitializeFromBlockReader	IN
IWICStream	IWICStream::InitializeFromIStream	IN
	IWICStream::InitializeFromIStreamRegion	OUT
IXMLDOMDocument	IXMLDOMDocument::QueryInterface	OUT
	IXMLDOMDocument::save	IN