Strict Virtual Call Integrity Checking for C++ Binaries

Mohamed Elsabagh
melsabag@gmu.edu

Dan Fleck
dfleck@gmu.edu

Angelos Stavrou
astavrou@gmu.edu

Department of Computer Science
George Mason University
Fairfax, VA 22030, USA

ABSTRACT

Modern operating systems are equipped with defenses that render legacy code injection attacks inoperable. However, attackers can bypass these defenses by crafting attacks that reuse existing code in a program's memory. One of the most common classes of attacks manipulates memory data used indirectly to execute code, such as function pointers. This is especially prevalent in C++ programs, since tables of function pointers (vtables) are used by all major compilers to support polymorphism. In this paper, we propose VCI, a binary rewriting system that secures C++ binaries against vtable attacks. VCI works directly on stripped binary files. It identifies and reconstructs various C++ semantics from the binary, and constructs a strict CFI policy by resolving and pairing virtual function calls (vcalls) with precise sets of target classes. The policy is enforced by instrumenting checks into the binary at vcall sites. Experimental results on SPEC CPU2006 and Firefox show that VCI is significantly more precise than state-of-the-art binary solutions. Testing against the ground truth from the source-based defense GCC VTV, VCI achieved greater than 60% precision in most cases, accounting for at least 48% to 99% additional reduction in the attack surface compared to the state-of-the-art binary defenses. VCI incurs a 7.79% average runtime overhead which is comparable to the state-of-the-art. In addition, we discuss how VCI defends against real-world attacks, and how it impacts advanced vtable reuse attacks such as COOP.

Keywords

Virtual table attacks; C++; Control flow integrity; Type-call pairing; Static binary analysis

1. INTRODUCTION

Presently, memory subversion remains an unsolved security threat. By manipulating control data, such as function pointers and return addresses, attackers can hijack the control flow of programs and execute arbitrary code. Even though modern systems are equipped with W⊕X and Data Execution Prevention (DEP), attackers can still achieve arbitrary code execution by repurposing existing code from the program memory, in what is known as code-reuse attacks. This can range from reusing blocks of instructions, such as Return Oriented Programming (ROP), to even reusing whole functions in a Function Reuse Attack (FRA).

The use of Control Flow Integrity (CFI) [1], which is a critical program security property, can assure the program does not execute unintended code. Unfortunately, constructing a sound and complete CFI policy has proven to be a challenging task [9]. Enforcing CFI is especially hard due to indirect control flow transfer, such as indirect calls through function pointers. The problem becomes even harder if the source code is not available. This makes binary-only solutions very desirable, since, in practice, the source code of many programs is not always available, and that includes many commercial products, 3rd party libraries, legacy software and firmware to name a few. Even if the source code is available, compiling in new protections is not always feasible or desirable, for instance, due to the presence of legacy code and compiler dependencies.

Indirect calls are prevalent in OOP languages in order to enable polymorphism. Of particular interest to us is C++, where all major compilers, including GCC, LLVM, and MSVC, support C++ polymorphism via tables of function pointers. This is also the case for compilers of closely related languages, such as C# and D. C++ supports class and function polymorphs by allowing derived classes to redefine base functions that are declared virtual. Each object of a class that (re)defines virtual functions stores a pointer (`vptr`) to a read-only table of pointers to virtual function definitions (called vtable for short). To invoke a virtual function, the compiler generates code that indirectly executes the corresponding function in the object's vtable (see Section 2). We refer to such code sites in the binary as virtual call (vcall) sites.

In an unprotected binary, an attacker with control over an object's memory or vtable can call any function within the program whenever the program uses the object's vtable to make a vcall. This is typically achieved by exploiting a memory access bug that enables overwriting the `vptr` in an object's memory, in what is known as a "vtable attack". Perhaps the most common class of enabler bugs in this category is the infamous use-after-free [2]. Here, a pointer to a freed object is used in a later program statement (a dangling pointer) to invoke one of the object's virtual functions. This dangling pointer can allow an attacker to execute arbitrary code if she can control the contents of the object's freed

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates

DOI: http://dx.doi.org/10.1145/3052973.3052976

memory, e.g., using heap overflows or heap spraying [12]. Such bugs are very prevalent in commodity desktop applications, such as office suites and browsers, since they are typically written in C++. Recent studies (e.g., [7, 22, 25]) suggested use-after-free vulnerabilities account for at least 69% of all vulnerabilities in browsers, about 50% of Windows 7 exploits, and 21% of all vulnerabilities in all operating systems.

In this paper, we present VCI, a static binary CFI system that retrofits C++ binaries with defenses against vtable attacks. VCI protects the binaries by enforcing a strict CFI policy that limits the number of callable function from vcall sites (see Section 3). VCI works on stripped binaries, without needing debug, symbol or type information. To determine valid function targets we developed algorithms to reconstruct several C++ semantics from binaries, namely: vtables, constructors, class layouts, class hierarchies, and vcalls (see Section 4). VCI exploits patterns in the assembly, and uses backward slicing and inter-procedural analysis to symbolically trace the `this` pointer expressions of objects across function boundaries. It builds a mapping between vcall sites and their target class types. It then instruments the binary by generating and injecting the integrity policy to enforce the mapping at runtime.

We implemented a prototype of VCI in C++ on Linux, using Dyninst [15] for binary parsing and rewriting. The prototype consists of ∼3500 SLOC for the analysis in addition to a ∼500 SLOC dynamic library where the integrity policy procedures reside. Experimental results (see Section 5) on the C++ SPEC CPU2006 benchmarks and Mozilla Firefox show that VCI significantly reduces the attack surface compared to the state-of-the-art binary vtable defenses. For instance, in comparison with VTint [43] and vfGuard [30], VCI achieved *at least* 96% and 48% additional reduction in the number of allowable vcall targets, respectively. In comparison to GCC VTV (source-based ground truth), VCI achieved the highest precision amongst other binary solutions, with 100% precision in some cases and greater than 60% precision for the majority of the test programs. Our experiments show that VCI incurs a low runtime overhead (∼7.79%), and can defend against real-world exploits including the recent COOP attacks [10, 34]. In summary, we make the following contributions:

- We present VCI, a binary analysis and rewriting tool that automatically analyzes and retrofits stripped C++ binaries with a strict defense against vtable attacks.
- We introduce multiple algorithms to reconstruct C++ semantics from binaries, without the need for source code, debug symbols, or symbol and type information. VCI employs these algorithms along with inter-procedural type propagation to resolve vcall targets.
- We introduce a strict and precise integrity policy that covers all three cases of fully, partially and unresolved vcall targets. VCI constructs and enforces the policy via static binary rewriting.
- We quantify the precision of VCI's policy on various C++ programs, and compare it to the precision of the state-of-the-art binary vtable defenses as well as to GCC VTV [37], the de facto standard source-based vtable defense of GCC. We show that VCI has significantly higher precision than the state-of-the-art binary solutions.
- We empirically quantify the effectiveness of VCI, discuss how it impacts COOP attacks, and benchmark its runtime overhead. We show that VCI can mitigate real-world attacks, and incurs a comparable overhead to existing solutions.

The rest of the paper is organized as follows: Section 2 provides an overview of relevant C++ primitives. In Section 3 we define the threat model, discuss vtable attacks and give an overview of our solution. Section 4 lays out the details of VCI. We evaluate VCI in Section 5, and discuss limitations and improvements in Section 6. We present related work in Section 7 and conclude in Section 8. In the Appendix, we provide additional technical details and discuss complementary policies.

2. BACKGROUND

Commodity applications, such as office suites and web browsers, are built with performance in mind. Given the sophisticated functionalities they provide, it is standard to use languages that provide sufficient levels of abstraction with a minimal performance penalty. Therefore, low-level object-oriented languages, such as C++, are typically the choice for their implementation. To enable polymorphism, C++ uses virtual functions. A function is declared virtual if its behavior (implementation) can be changed by derived classes. The exact function body to be called is determined at runtime depending on the invoking object's class.

2.1 Polymorphism and Virtual Tables

All major C++ compilers, including GCC, Clang/LLVM, MSVC, Linux versions of HP and Intel compilers, use vtables to dispatch virtual functions. A vtable is a reserved read-only table in the binary that contains function pointers to the definitions of virtual functions accessible through a polymorphic class. A polymorphic class is a class that declares, defines or inherits virtual functions.[1] Each virtual function in a class has a corresponding offset in the class' vtable which stores the address of the implementation body of the function in the code section. Whenever an object of some class type invokes a virtual function, the class' vtable is accessed, and the address at the corresponding function offset is loaded and indirectly called. If a class implements virtual functions, when an object of that class type is created, the compiler adds a *hidden* pointer to the class' vtable (the `vptr`). The compiler also generates code in the class' constructor to set the `vptr` to the address (effective beginning) of its corresponding vtable.

2.2 Virtual Call Dispatch

Since a vcall is always invoked on some object, the compiler has to decide how to pass the pointer of the object, i.e., the `this` pointer, to the callee. There are two widely adopted argument passing conventions for vcalls: `thiscall`, which is the default convention used by the MSVC compiler on Windows, and `stdcall` adopted by GCC, LLVM and other Linux compilers. In the `thiscall`, the `this` pointer is passed in the `ecx` register to the callee, while the remaining arguments are passed on the stack. In the `stdcall`, the `this` pointer is passed as an implicit argument on the stack (top of stack). The argument is implicit in the sense that it is not part of the callee function signature as seen by the developer.

[1]Unless explicitly stated, we use the term "class" to refer to "polymorphic class" in the rest of this document.

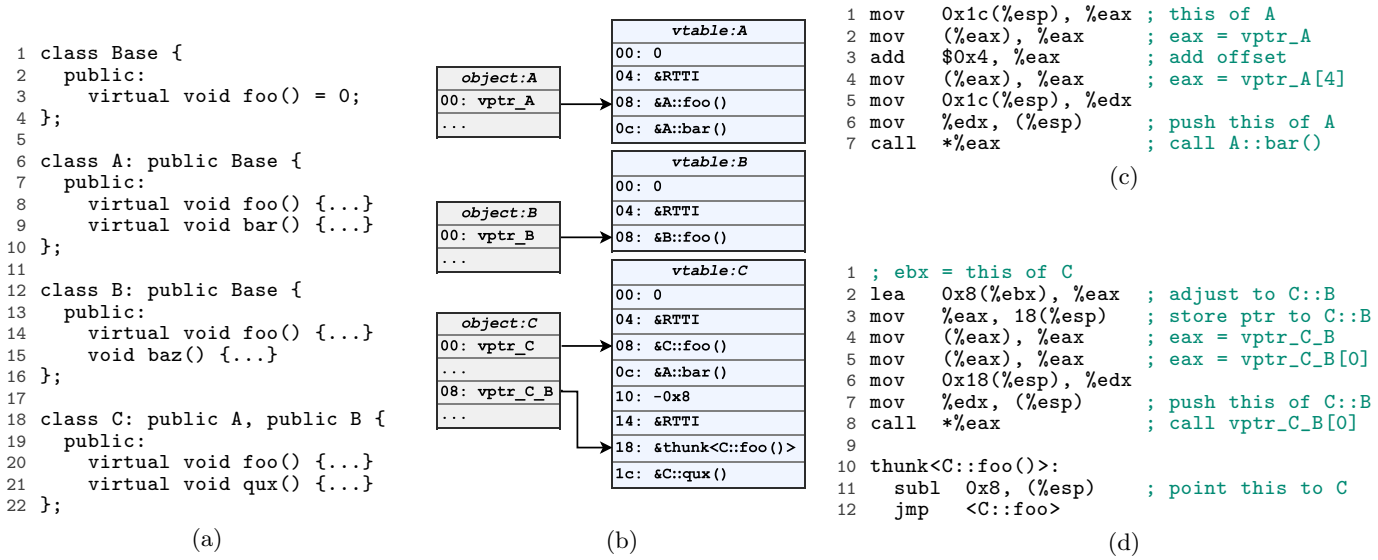

Figure 1: (a) Sample C++ classes. (b) Corresponding layouts of instances of classes `A`, `B`, `C`, and their vtables. (c) Assembly snippets for invoking `A::bar()`. (d) Assembly snippet for invoking `C::foo()` using a base pointer of type `B` (e.g., `B *ptr = new C(); ptr->foo();`). Note that `B::Baz()` is not virtual and therefore is not in the vtables.

Figure 1(c) shows the steps taken to dispatch a vcall based on the Itanium ABI, which comprises the following steps:

1) The `this` pointer of the target object is loaded and dereferenced.
2) An offset is added to the `vptr` to point to the vtable entry with the address of the target virtual function.
3) The adjusted `vptr` is dereferenced to load the address of the target virtual function (the vcall address).
4) The `this` pointer is pushed on the stack.
5) The virtual function is invoked by indirectly calling the vcall address.

Note that step 2 is optional, depending on the index of the target virtual function in the vtable. If it is the first function in the vtable, the offset is 0 and step 2 is omitted. If the virtual function takes arguments, they are all pushed before the `this` pointer at step 4. While steps 1 – 3 have to occur in that specific order due to data dependency, the ABI does not guarantee the order of steps 1 – 4. For example, pushing the `this` pointer and the arguments (step 4) can occur before step 1, or even in a different (predecessor) basic block.

In later sections we use this pattern as part of the algorithm to locate virtual call sites in the binary.

2.3 Inheritance

C++ supports single, multiple, and virtual inheritance. When a derived class inherits from base classes, the constructor of the derived class calls the constructor of each base class, in the order of inheritance. The derived class passes its `this` pointer to each base constructor. In the case of multiple inheritance, the `this` pointer is adjusted to point to the beginning of the base subobject in the derived object's memory layout. According to the Itanium ABI, inheritance of virtual functions is implemented using multiple vtables, one for each base class. When a derived class C inherits from base classes A and B, an object of type C would contain two subobjects of types A and B, each with its own vtable and `vptr`. The effective vtable of the derived class consists of a table of vtables (called VTT), one for each subobject type, in the order of inheritance, with the only exception that the derived class and the first subobject share the same `vptr`. Figure 1(b) illustrates the layout of vtables in memory for single and multiple inheritance.

This leads to the need for `this` pointer adjustments when using a base pointer to a derived class. For example, Figure 1(d) shows the assembly generated for invoking `ptr->foo()`, where `B *ptr = new C()`, i.e., `ptr` is of base class type besides the first base (first base is `A`, second is `B`). The compiler adjusts the pointer before the vcall to point to the subobject `B` in `C` (line 2). It then calls (indirectly) a thunk that re-points `this` to `C` then jumps to the actual (derived) function body. Similarly, `this` adjustments are used to access and invoke virtual functions of member class objects (more on this in Section 4.3). In VCI, we keep track of any adjustments done on identified `this` pointers, and reconstruct the inheritance hierarchy among polymorphic classes.

3. PROBLEM DEFINITION

Given a C++ program binary, VCI aims to protect the program against vtable attacks by enforcing a strict CFI policy at vcall sites. Specifically, VCI guarantees that for each vcall site, the vcall target is one of the class types that can be legitimately used by that particular vcall site, as statically inferred. If the condition is violated, VCI raises an alarm and terminates the program.

In the following, we discuss our assumptions and give a quick overview of vtable attacks in C++ binaries and how VCI operates.

3.1 Assumptions and Threat Model

We assume that: 1) attackers can read arbitrary readable memory, therefore bypassing any secret-based solution where the secret is stored in readable memory. 2) They can write arbitrary writable memory, including injecting vtables and modifying objects' layouts and contents. 3) They cannot

control the memory protection flags without injecting and executing foreign code. In other words, legitimate control flow transfers in the target program *cannot* allow the attacker to alter the memory protection of a specific memory region. Those assumptions cover most practical scenarios of attacks, without any unrealistic limits. We also assume that traditional arbitrary code execution defenses are live on the OS, such as ASLR and DEP. This work focuses on vcall protection, which is pertinent to forward-edge control transfers. We assume that other control flow transfers, including non-control-data attacks, are protected and cannot be used to redirect the flow of vcalls.

We assume that the binaries adhere to the Itanium ABI (see Appendix C); analyzing non-compliant or obfuscated binaries is outside the scope of this work. We assume the binaries are stripped from auxiliary information, such as debug and symbol information, including the C++ RTTI (see Appendix D), and we do not assume any particular optimization level. While these assumptions complicate our analysis, we believe it is unreasonable to assume the presence of side information when dealing with stripped binaries. The analysis performed in this paper assumes knowledge of function entry points in the binaries. We depend on Dyninst [15] in this regard, which has shown outstanding identification accuracy of function entry points in stripped binaries [20, 40], outperforming various well-established static analysis tools.

Finally, various constructs presented in this paper require instruction-level analysis that depends on the semantics of the instruction set being parsed. Therefore, we tailor our discussion in this paper to x86_32 (e.g., call parameters passed on stack instead of in registers as in x86_64). Nevertheless, the approach itself does not put any assumptions on the underlying architecture and can be implemented for other instruction sets without an issue.

3.2 Vtable Attacks

By exploiting a memory access bug (e.g., use-after-free[2]), an attacker can launch vtable attacks and achieve arbitrary code execution by overwriting a C++ object's memory with contents of his or her choice (e.g., via heap spraying [12]). The attacker can inject a fake vtable or perhaps redirect the object's `vptr` to an existing vtable. Without loss of generality, vtable attacks in C++ can be divided into three categories [17, 22, 27, 30, 43]:

1) Vtable corruption. This is a legacy attack, where legitimate vtable contents are overwritten. The attack is prevented by all major compilers by storing the vtables in a read-only memory region.
2) Vtable injection. Here, the attacker first injects a fake vtable into memory, then points the vtable pointer of a hijacked object to the injected vtable. The injected vtable can therefore point to arbitrary functions or gadgets in the executable memory of the process.
3) Vtable reuse. This attack operates the same way as a vtable injection attack, except that the attacker does not inject any counterfeit vtables in memory. Instead, the attacker reuses already existing vtables in the process memory.

While the state-of-the-art binary vtable defenses reduce the vtable attack surface, they do not extract sufficient semantics from the binaries, and therefore enforce imprecise policies that allow a very liberal number of target functions per vcall site. To address this limitation, we introduce VCI, a binary rewriting system that fully protects against vtable corruption, injection, and significantly reduces the attack surface of vtable reuse in C++ binaries. In the following, we give an overview of how VCI operates and give a simplified example of a retrofitted program and the integrity policy enforced by VCI.

3.3 Overview of VCI

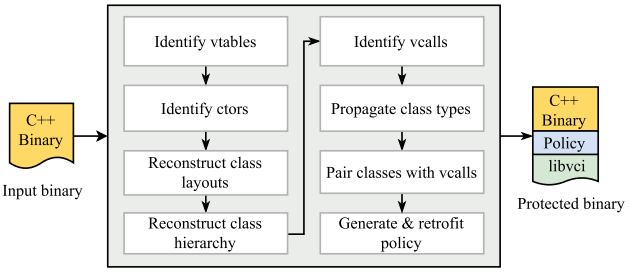

Figure 2: Overview of VCI. The input to VCI is a binary file (executable or library), and the output is a binary file retrofitted with integrity checks and the VCI integrity enforcement library (libvci).

Figure 2 outlines the workflow of VCI. It operates as follows: first, it statically analyzes the binary and extracts all vtables and constructors. It then reconstructs (partially) class layouts and hierarchies. Then, it identifies all vcalls in the binary. VCI then propagates the identified class types to all vcall sites, using backward slicing and inter-procedural data flow analysis. This produces a set of legitimate target class types and their corresponding vtables for each vcall. When VCI fails to resolve all target class types of a vcall, it utilizes the inferred hierarchies and any known targets for the vcall to construct a set of class types that the vcall may be invoked on. As our experiments show, this is significantly more precise than prior works which either liberally permitted any class type to be used at any vcall site, or any class type where the vcall offset is valid. Specifically, VCI constructs and enforces the mapping: $\mathcal{F}$: $vcall \times class \rightarrow vtable$ by instrumenting checks at each vcall site to test if the vcall target class is one of the valid target class types for that vcall.

4. DESIGN AND IMPLEMENTATION

4.1 Identifying Virtual Tables

To extract vtables, VCI scans the binary for assembly sites that store an address (immediate value) into memory, where the address resides in a read-only memory region, and the words (pointer-size sequence of bytes) at positive offsets of the address are pointers to functions in the code section (see Algorithm A.1). For each such assembly site, VCI starts with an empty vtable, and scans the corresponding memory region, starting at the stored address, one word at a time. Each word is matched against a set of all function addresses in the binary. If a match is found, the word is added to the vtable, otherwise, the algorithm proceeds on to the next assembly site. According to the Itanium ABI, the vtable address referenced by an object's `vptr` (i.e., the entry point of the vtable as seen by the object) is 1) pointer aligned, and 2) points to the beginning of the virtual function pointers array in the vtable. Finally, the algorithm returns the extracted vtable. Note that the algorithm identifies vtables

separately. For example, it will identify two separate vtables for class `C` in Figure 1; the VTT of `C` is populated when VCI reconstructs the class layout.

While VCI may identify false vtables, such as C-style arrays of function pointers (jump tables) stored in a read-only region, the proposed algorithm is sound. It does not miss any real vtable in the binary (no false negatives). This is an important property as missing a legitimate vtable can result in an incorrect policy or misdetection of attacks. Note that C-style jump tables that are misidentified as vtables do not satisfy later stages of the analysis (e.g., no corresponding constructors), and their calling convention does not generally match that of virtual functions (see Sections 4.2 and 4.4). Overall, overestimation of vtables affects only the precision of VCI rather than its soundness, by increasing the number of potential targets of a vcall.

4.2 Identifying Constructors

Each extracted vtable corresponds to one class that declares virtual functions. Each such class will have at least one constructor and one virtual function declaration. To extract constructors, VCI applies Algorithm A.2. It searches the code section for functions that store a pointer to a vtable at the memory location pointed at by `this`, i.e., the first entry in an object's memory. This is done by searching for a function that contains an assembly site that stores an immediate value in memory, where 1) the immediate value matches the address of one of the extracted vtables; 2) the destination expression has zero displacement; and 3) the destination expression is a memory location pointed at by the first argument to the function, e.g., `mov 0x8(%ebp),%eax; movl $0x9b0,(%eax)`. Once identified, the vtable is scanned for occurrence of a pointer to that same function. If a pointer to the function is not found in the vtable, the function is deemed a constructor. Note that C++ does not allow virtual constructors, therefore constructors cannot have entries in the vtable. Similarly, inlined constructors are identified by relaxing the first argument condition. In this case, the store instruction that writes the vtable address in the object's memory is marked as a construction point.

4.3 Inferring Class Layouts and Hierarchies

When a derived class inherits from a base class, the constructor of the derived class calls the base constructor, passing in the `this` pointer of the derived class after applying any necessary pointer adjustments (in case of multiple inheritance). The same semantics are also applied when constructing member objects.

VCI infers class layouts that consist of offsets to polymorphic member objects and base subobjects, and offsets to vtables (the VTT in case of multiple inheritance). The offsets are computed relative to the class `this` ptr. We collectively refer to member objects and base subobjects as subelements. We define each subelement by the tuple: ⟨`cls, offset, dst, deref`⟩, where `cls` is the containing class, `offset` is the subelement's offset from the `this` pointer of `cls`, and `dst` is the corresponding subelement's class. `deref` is a flag indicating whether the subelement has to be dereferenced before accessing, e.g., if a member is a pointer to an object, where in this case the class stores only the subelement's `this` pointer instead of the subelement itself.

Algorithm A.3 outlines the steps taken to infer the layout. For each class, VCI infers the class layout by, first, searching the instructions of the class constructor for assembly call sites that invoke a constructor. Then, for each identified call site, it extracts and analyzes the arguments to the call site to identify the `this` pointer of the subelement's constructor. It then computes the offset of the subelement's `this` pointer to the class `this` pointer. Recall that the `this` pointer points to the address at which the vtable pointer is stored in an object's memory. VCI computes the offset by analyzing the adjustments performed on the `this` pointer before calling the subelement's constructor. For example, `mov 0x8(%ebp),%eax; add 0x4,%eax; mov %eax,(%esp); call sub_ctor()` ; constructs a subelement at offset `0x4` from the `this` pointer of the class. This results in an expression of the form *this + offset*, where *this* is the class `this` pointer, and *offset* is the distance to the subelement's `this` pointer. Finally, VCI checks if the subelement needs to be dereferenced before accessing by checking if the `this` pointer passed to the subelement's constructor is stored in memory after the call to the subelement's constructor.

Similarly, VCI populates the class VTT by identifying the assembly site that stores pointers to vtables, relative to the `this` pointer of the class. For example, `mov 0x8($ebp),%eax ; mov $0x848,(%eax); add $0x8,%eax; mov $0x88c,(%eax );` corresponds to a VTT of two entries `0x848` at offset `0` and `0x88c` at offset `0x8`. Note that the first entry of the VTT is the class' vtable itself.

To reconstruct inheritance relationships between polymorphic classes, VCI needs to differentiate between calls to a base constructor and calls to construct member objects. According to the ABI, in a derived class, its virtual base class' subobjects are constructed before its member objects. In addition, the compiler has to populate the VTT of the class before constructing its member objects. In other words, all calls to constructors that 1) take the derived class' `this` pointer (adjusted) as the top argument on the stack, and 2) occur before storing the vtable address at a zero offset from the `this` pointer in the object's memory, are calls to base constructors. By identifying this pattern in the assembly of constructors, VCI constructs the "is-a" relationship among the identified polymorphic classes. Note that the actual offsets in the VTT in the binary must match the offsets VCI extracted for inherited classes. Additionally, the soundness of the inferred hierarchy follows from the soundness of VCI's vtable identification (no FNs). We do not attempt to construct the full class hierarchy that includes polymorphic and non-polymorphic classes, which is a known hard problem [16, 23, 26]. VCI uses the identified inheritance hierarchy to augment its policy when semantic gaps hinder the identification of all class types that a vcall operates on (see Section 4.6).

4.4 Identifying Virtual Calls

To extract call sites that invoke virtual functions, i.e., vcall sites, VCI scans the binary for indirect call sites that reflect the behavior of virtual function dispatches. That is, the indirect call target is computed by first dereferencing a pointer (the vtable pointer), then adjusting the resulting address to pick an entry of the vtable by adding a non-negative constant offset to it, and finally dereferencing the final adjusted address to retrieve the address of the target function. In addition, the same expression used to dereference the vtable (the `this` pointer) is passed as the first argument (top of stack) to the indirect call. Note that the offset used in a vcall site is not a target of attacks as it is always hardcoded in the assembly as an immediate value or a displacement. However,

attackers can effectively change the offset by modifying the vtable address (`vptr`) referenced by the `this` pointer.

Though a vtable and a jump table (an array of function pointers) share common structure, the semantics for invoking virtual functions are different from those for dispatching functions from a jump table. To dispatch a function pointer from a jump table, the jump table is directly indexed rather than offset and dereferenced. For example, given an index in `%ecx` and a jump table stored at address `0xa034`, the target function from the jump table is invoked by: `call 0xa034(,%ecx,4)`.[2] This dissimilarity in how the indirect call target is computed enables VCI to filter out any spurious jump tables that might have been mislabeled as vtables by Algorithm A.1.

This approach in itself does not yield FPs (false positives, i.e., incorrectly identifying a call as a vcall) for ABI-compliant binaries. However, it might incorrectly identify some specific C constructs as vcalls in mixed C/C++ binaries. Besides that, special compiler rearrangements and non-standard calling conventions that may not be handled by the implementation can result in FNs (false negatives, i.e., missing valid vcalls). We evaluate the identification accuracy of VCI in Section 5.1 and discuss vcall-like C constructs in Section 6.3.

4.5 Class Type Propagation and Pairing

4.5.1 Intra-Procedural Analysis

VCI implements a custom, slicing-based, intra-procedural analysis algorithm (illustrated in Algorithm A.4) to construct intra-procedural bindings between classes and (v)calls. It starts by analyzing the assembly sites at which calls are invoked. For each call site, it extracts a backward slice, starting at the assembly store point of each argument to the call site and ending at the entry point to the procedure. VCI analyzes the slice and decides whether and what classes the argument depends on, i.e., there is data dependence between the call parameter and one or more `this` pointers defined within the same procedure.

Each backward slice is a Program Dependency Graph (PDG) constructed via Value-Set Analysis (VSA) [4]. Nodes in the PDG correspond to program constructs (assignment expressions) and edges correspond to data and control dependencies between the assignments. Since there is no notion of variables at the binary level, VSA extracts variable-like abstractions using the semantics of the instructions. Due to the multi-assignment (multi-source, multi-destination) nature of assembly instructions, the produced slices are often cluttered with irrelevant expressions and dependency paths [36]. To overcome this, VCI analyzes the slice by traversing backwards all paths in the PDG from the exit node (i.e., the call argument) to each entry node. For each path, VCI traces (backwards) the data flow of the argument through memory and registers, till it reaches a construction site (the defining constructor) or an entry node. During this, VCI also maintains a list of all encountered adjustments (via offsets and displacements) that were performed on the parameter expression, in their order of execution. See Figure B.2 for a example snippet and its corresponding PDG generated by VCI.

[2]Code generated by both GCC and Clang with `-O1`, `-O2`, and `-O3`. For `-O0`, GCC and Clang emitted: `mov %ecx,%eax; mov 0xa034(,%eax,4),%eax; call *%eax`, which also does not satisfy the semantics of a vcall.

If such flow exists, then a data dependency is present between the call parameter and that construction site (and its corresponding class). In this case the parameter type is resolved by pairing it with the effective class resulting from the corresponding construction site class after applying any `this` pointer adjustments. If the resolved parameter is the first parameter of the vcall (i.e., the `this` pointer), VCI resolves the vcall target using the resolved parameter's vtable and the vcall offset. It then adds an edge to the CFG between the vcall and the resolved virtual function address.

If there is no data flow, then the path is ignored. Finally, if a flow exists but the defining constructor was not found, that could mean either the definition point is in a different procedure or there is a semantics gap, which we discuss in the following sections.

4.5.2 Inter-Procedural Analysis

VCI performs inter-procedural analysis by recursively propagating the `this` definitions and parameter type information of each procedure down the CFG, through returns and successor call sites. The analysis traces the `this` pointers of both class objects and members as identified in Section 4.3. Specifically, class types are propagated across function boundaries by checking the equivalence of the expressions of the arguments pushed on the stack at the call site (in the caller function) with those loaded from the stack in the preamble of the callee. For example, `%edx` and `0x8(%ebp)` in the following snippet are equivalent: `foo: push %edx ; call bar;` and `bar: mov 0x8(%ebp),%eax`. For returns, class types are recursively propagated across all procedures exit points (function returns) if there is a data dependency between the exit point and the `this` pointers of the objects referenced in the function body.

Vcall Target Resolution. For vcalls, VCI attempts to resolve the vcall target by identifying data flows from the incoming `this` pointer expressions on the stack of the enclosing (parent) procedure, to the arguments of the vcall. Similar to Section 4.5.1, this is done by pairing arguments to incoming (adjusted) class types via reachability analysis over a backward slice starting at each argument to the vcall and ending at the entry of the enclosing procedure. If the first argument (i.e., the `this` pointer the vcall is invoked on) type is successfully resolved, VCI finds the corresponding virtual function address in the corresponding class' vtable, at the offset that appears in the vcall site, and adds an edge to the CFG between the vcall and the virtual function address. The algorithm stops when the CFG stops changing.

Due to semantic gaps (see Section 6), it is possible that VCI fails to resolve all definition points of a vcall's `this` pointer, resulting in potentially missing some valid vcall targets. This divides vcalls into three categories: 1) **fully resolved vcalls**, where all definition points were successfully paired; 2) **partially resolved vcalls**, where some but not all definition points were resolved; and 3) **unresolved vcalls** where all definition points were not paired with any type. In the following section, we discuss how VCI generates and enforces its policy such that it covers all the three cases, yet be as strict as possible.

4.6 Policy Generation and Enforcement

VCI generates the following policy, based on vcall target resolution results:

1) For fully resolved vcalls, all legit targets were success-

fully identified, and only those targets are considered valid.
2) For partially resolved vcalls, find the *common base classes* among the identified targets and *all child* classes that inherit from those common bases (including the identified targets). Assume that all vtable functions at the vcall offset in those classes are valid targets.
3) For unresolved vcalls, i.e., no targets were identified for the vcall, assume that all vtable functions at the same vcall offset are valid targets (the same integrity policy applied by Prakash et al. [30]).

The policy is implemented by constructing the mapping $\mathcal{F}$: $vcall \times class \rightarrow vtable$. VCI stores in the binary a read-only set $\mathcal{C}$ of the extracted class types and their inferred layouts, one class for each nonempty vtable. Each class is assigned a unique ID. Then, before each vcall site v, VCI aggregates a set of IDs $\mathcal{L}_v$ of all the class types that v can be invoked on, based on the three aforementioned policy cases (for fully, partially, and unresolved vcalls). It then injects code in the binary that enforces $\mathcal{F}$ by checking for the following:

1) There is a class c in $\mathcal{C}$ with the same vtable address of the class c' accessed by the vcall.
2) The ID of c belongs to the valid IDs $\mathcal{L}_v$, i.e., $c_{ID} \in \mathcal{L}_v$.
3) The class layout of c' matches the layout of c.

If any of the conditions is not met, the execution is aborted and an alarm is raised. Otherwise, the vcall is dispatched. Note that no policy is enforced for static (direct) call sites, e.g., `call 660`, since they do not pose a threat under our threat model. In addition, checking the class layout is important in order to detect reuse attacks that modify the `vptr` of an object to point to a different vtable than the actual object's vtable, where both vtables are valid for the vcall site. The layout checks validate the information extracted in Section 4.3, i.e., the contents of all involved vtables and offsets of subobjects from each `this` pointer.

In our prototype implementation, the definitions of the policy enforcement procedures are exported in a dynamic library (libvci) that VCI injects into the binary. At runtime, libvci linearly checks the policy conditions on the valid classes set of v, i.e., $\{\mathcal{C}\,[\mathcal{L}_v[i]] \mid i \in 1 \ldots |\mathcal{L}_v|\}$, where $|\mathcal{L}_v| \leq |\mathcal{C}|$. A potential performance improvement is to add a sublinear index, such as using binary search over vtable addresses whenever $\lg |\mathcal{C}| < |\mathcal{L}_v|$, or a read-only hash table that maps vtable addresses to classes. We decided to go with linear constant arrays for simplicity and to avoid unintentionally introducing writable memory or more attack points.

5. EVALUATION

In our evaluation of VCI, we answer the following:

1) How accurately can VCI identify vtables and vcalls? Section 5.1.
2) How precise and effective is the policy enforced by VCI, compared to both binary and source-based state-of-the-art C++ defenses? Section 5.2.
3) How much runtime overhead do binaries protected by VCI incur? Section 5.3.

All experiments were conducted with GCC 4.8.2 on Ubuntu 14.04.1, running on 2.5GHz Intel Core i7 with 16GB RAM. The results are reported for `-m32` and `-O2` optimization, but we observed similar results at other optimization levels.

5.1 Identification Accuracy

We compiled the C++ SPEC CPU2006 benchmarks and the C++ Firefox modules[3] with debug and symbol information, then counted the number of nonempty vtables by parsing the output of the `objdump -Ct` command, which demangles and dumps the symbol table entries of a binary. That count is used as the ground truth. We then compiled the same programs without debug and symbol information, processed them by VCI, counted the number of extracted vtables and compared to the ground truth. Here, FNs (missing a vtable) are not desired, while FPs are acceptable since the policy is enforced at vcall sites rather than the vtables themselves. In other words, falsely identified vtables will not result in FPs at runtime, but in lower precision during the identification of vcall targets.

We also report the count of vcalls in each binary, and compare that to the ground truth from GCC VTV. VTV inserts checks at each vcall site in the binary to validate its vtable. We compiled each of the test programs with and without VTV, and matched the call sites that contained VTV checks against the vcall sites identified by VCI. Note that, unlike vtables, falsely identified vcalls may result in runtime crashes. Thus, FPs in terms of vcalls are undesired, or else the enforced policy would be unsound. On the other hand, FNs (missed vcalls) do not sway the soundness of the policy, rather they reduce its precision.

Table 1 shows the breakdown of our analysis. VCI did not miss any legitimate vtable in the binaries, achieving zero FNs. It incorrectly identified some memory blocks as vtables in five out of the 13 binaries, resulting in FPs between 0.04% and 3.33%. In terms of vcalls, VCI did not report any FPs, but it had some FNs (missed vcalls) between 0.32% and 2.18%. These results indicate that VCI shall be sound, but not perfectly precise (not complete) due to the missed vcalls and the overestimated vtables. We quantify the precision of VCI in the following section.

5.2 Security Effectiveness

5.2.1 Policy Precision

Quantifying the effectiveness of a defense system is a difficult task. Recent work by Zhang et al. [45] introduced the Average Indirect-Target Reduction (AIR) metric as a quantitative measure of the security introduced by a defense. We understand that the AIR metric has been questioned by the community [8], primarily since it does not quantify the usefulness of the remaining targets from the attacker's perspective. However, for the sake of comparison with similar defenses, we use an AIR-based metric to evaluate VCI. We concede that a better evaluation metric is needed, albeit outside the scope of this work. Developing a conclusive security metric is a very challenging task, especially when dealing with whole functions as in the case of VCI. To give conclusive results, we also compare the precision of VCI to that of GCC VTV, the state-of-the-art source-based vtable defense.

In the context of VCI, we are only interested in defending vcalls, which are forward-edge control transfers. Therefore, we only compute the average number of vcall targets over all vcalls. We protected the C++ SPEC CPU2006 benchmarks and the C++ Firefox modules, and computed the average

[3]For the non-C++ FireFox modules, VCI did not identify any vtables in the binaries and aborted the analysis without modification to the binaries.

Table 1: Analysis result of the C++ SPEC CPU2006 benchmarks (top) and the C++ Firefox modules (bottom), including the analysis time in seconds, number of identified vtables and vcalls, and the identification accuracy.

Program	#Insns	Analysis Time (sec.)	Identified Vtables				Identified Vcalls			
			Ground Truth	VCI	%FP	%FN	Ground Truth	VCI	%FP	%FN
444.namd	91k	22.8	4	4	0.00%	0	2	2	0	0.00%
447.dealII	789k	193.2	732	736	0.55%	0	936	916	0	2.18%
450.soplex	110k	128.4	30	30	0.00%	0	522	511	0	2.15%
453.povray	256k	143.4	30	31	3.33%	0	129	127	0	1.57%
471.omnetpp	166k	207.5	114	114	0.00%	0	710	706	0	0.57%
473.astar*	12k	0.2	1	1	0.00%	0	0	0	-	-
483.xalancbmk	1m	329.6	962	971	0.94%	0	9201	9134	0	0.73%
liblgbllibs.so	9k	12.3	8	8	0.00%	0	63	63	0	0.00%
libmozgnome.so	53k	134.7	24	24	0.00%	0	209	206	0	1.46%
libmozjs.so	2m	510.4	1230	1244	1.14%	0	3796	3784	0	0.32%
libxul.so	27m	2973.6	14465	14471	0.04%	0	79703	79315	0	0.49%
libzmq.so	122k	108.2	67	67	0.00%	0	135	133	0	1.50%
updater	31k	15.3	8	8	0.00%	0	8	8	0	0.00%

* We manually inspected 473.astar and found that it contained 4 indirect calls, none of which were vcalls.

number of targets per vcall. Then, we computed the precision of the policy as the percent reduction in the average number of vcall targets, compared to the source-based defense GCC VTV (perceived as the ground truth) as well as the two policies that appeared in prior studies:

1) "AnyV," permit the vcall target to be *any* function in any vtable (e.g., [27, 43]); and
2) "SameOff," permit only functions at the *same* vtable *offset* as the vcall site, in *any* vtable (e.g., [17, 30]).

The higher the reduction the more precise the enforced policy. We further assumed that the solutions that enforced any of those two policies had perfect knowledge of the vtables in the binaries. Since exploits do not only target vcalls, and for the sake of completeness, we also report the reduction in attack surface on indirect calls (icalls). This is computed as the percentage of vcalls (protected by VCI) to the total number of icalls in the analyzed binaries.

Table 2 shows the breakdown of the results per program. The results show that VCI achieved significantly higher precision that prior solutions. For some programs, it limited the vcall target to one or two functions on average (e.g., 444.namd, Firefox liblgllibs.so and updater). In comparison to the source-based VTV, VCI achieved the highest precision amongst other policies, with 100% precision in some cases, and greater than 60% precision for the majority of the programs. Compared to solutions that apply the AnyV policy, VCI achieved 87% to 99% reduction in the vcall targets. This is more pronounced in programs with large numbers of vcalls. For example, in Firefox libxul.so, VCI limited each vcall to only 1035 targets on average, while AnyV allowed 71069 targets per vcall. Compared to SameOff policies, VCI achieved 48% to 89% reduction. For the same libxul.so, a SameOff policy would permit 9692 targets per vcall, while VCI reduced that by more than 89%.

5.2.2 Real-World Exploits

We experimented with three publicly-available use-after-free vtable exploits for Mozilla Firefox: CVE-2011-0065, CVE-2013-0753, and CVE-2013-1690. All three vulnerabilities reside in libxul. CVE-2011-0065 exploits a use-after-free vulnerability in Firefox 3.6.16 where the mChannel pointer associated with an Element object can be used after being freed, via the OnChannelRedirect function of the nsIChannelEventSink class. CVE-2013-0753 exploits a vulnerability in Firefox verions prior to 17.0.2, where an object of type Element is used after being freed inside the the serializeToStream function of the nsDocumentEncoder class. CVE-2013-1690 exploits a vulnerability in Firefox 17.0.6 where a DocumentViewerImpl object is used after being freed, when triggered via a specially crafted web page using the onReadyStateChange event and the Window.stop API. This was the vulnerability exploited in the wild in 2013 to target Tor Browser users.

We downloaded vulnerable Firefox versions, protected the relevant C++ modules with VCI and tested the protected browser against exploits from Metasploit. Though some Metasploit modules for the aforementioned vulnerabilities supported only Windows, the HTML payloads that trigger the vulnerabilities are cross-platform. The only platform-specific part is the actual payload (ROP in all 3 exploits) that is executed after the vulnerability is exploited.

VCI identified and protected the vcalls targetted by the exploits, rendering the three exploits inoperable. All three exploits resembled a vtable injection attack. We could not find any publicly-available vtable reuse attacks. In the following, we discuss how VCI mitigates and hardens binaries against COOP attacks.

5.2.3 Impact of VCI on COOP

Schuster et al. [11, 34] introduced Counterfeit Object-Oriented Programming (COOP), a novel vtable reuse attack against C++ programs. In a COOP attack, the attacker injects a counterfeit (attacker controlled) object that repurposes existing virtual functions in the binary. The counterfeit object is specially crafted such that benign vulnerable constructs in the binary execute attacker picked virtual functions. The gadgets in a COOP attack are calls to virtual functions (vfgadgets). By chaining multiple vfgadgets via counterfeit objects, the attacker can achieve arbitrary code execution.

A COOP attack requires a memory corruption bug that enables injection of attacker-controlled objects. Besides that, it has two key requirements for a successful exploit: 1) the ability to target *unrelated* virtual functions from the *same* vcall site; and 2) the ability to flow data between the vfgadgets. The vfgadgets are dispatched via two types of initial vfgadgets: the main-loop gadget (ML-G), and the recursive

Table 2: VCI policy coverage and average target reduction in the analyzed programs. AnyV refers to solutions that allow any target as long as it is in a valid (read-only) vtable. SameOff refers to solutions that allow targets that are in a valid vtable and at the same offset of the vcall site. VTV represents the source-based ground truth.

Program	Avg. #targets per vcall				%Precision w.r.t VTV			%Reduction	
	AnyV	SameOff	VCI	VTV	AnyV	SameOff	VCI	vs. AnyV	vs. SameOff
444.namd	8	4	1	1	12.50%	25.00%	100.00%	87.50%	75.00%
447.dealII	3775	289	49	18	0.48%	6.23%	36.73%	98.70%	83.04%
450.soplex	725	17	8	7	0.97%	41.18%	87.50%	98.90%	52.94%
453.povray	384	41	13	8	2.08%	19.51%	61.54%	96.61%	68.29%
471.omnetpp	2361	72	37	21	0.89%	29.17%	56.76%	98.43%	48.61%
483.xalancbmk	11345	515	85	33	0.29%	6.41%	38.82%	99.25%	83.50%
liblgbllibs.so	61	5	2	2	3.28%	40.00%	100.00%	96.72%	60.00%
libmozgnome.so	149	20	9	6	4.03%	30.00%	66.67%	93.96%	55.00%
libmozjs.so	23840	657	91	27	0.11%	4.11%	29.67%	99.62%	86.15%
libxul.so	71069	9692	1035	63	0.09%	0.65%	6.09%	98.54%	89.32%
libzmq.so	979	42	13	9	0.92%	21.43%	69.23%	98.67%	69.05%
updater	44	7	2	2	4.55%	28.57%	100.00%	95.45%	71.43%

gadget (REC-G). The ML-G gadget represents a linear dispatch using a loop that iterates over a list of objects (counterfeit) and calls some virtual function of each object. The REC-G gadget corresponds to a recursive dispatch using two consecutive vcalls on different objects, where the first vcall dispatches one vfgadget and the second vcall recurses back into a REC-G.

In a COOP attack, data is passed between vfgadgets either explicitly or implicitly. In *explicit* data flows, the attacker picks vfgadgets that pass data via object fields or vcall arguments. In *implicit* data flows, data is passed via unused argument registers by chaining vfgadgets that take different numbers of arguments. Note that this is specific to architectures that pass arguments in registers by default, such as x86_64. Explicit data flow via object fields is achieved by overlapping objects, in memory, of *different* classes such that one vfgadget writes to some object field, then another vfgadget reads from the same field. On x86_32, this also requires an initial vfgadget that passes the same field to the dispatched vfgadgets, so that they can read or write to it. In the following, we discuss how VCI abrogates the attacker's ability to satisfy the COOP requirements.

Table 3 shows VCI's vcall target resolution coverage results (summary statistic are in Table G.1). VCI fully and partially resolved 58% plus 26% of all vcall targets, on average (geometric). Unresolved targets ranged from 0% to 29%, with an average of 14%. While the percentage of unresolved calls is not particularly low for some of the test programs, the percentage of fully and partially resolved targets outweighed that of unresolved targets in all programs.

For both fully and partially resolved vcalls, VCI guarantees that all targets of a vcall are at the same vtable offset and under the same class hierarchy. The targets in this case correspond to function polymorphs (redefinitions) of some virtual function in the hierarchy, therefore, all taking the same number of arguments. This prevents *implicit* data flows in COOP. This also means that the targets are functionally related, since they are part of the same hierarchy, and unlikely to exhibit useful semantics for the attacker.

For *explicit* data flows, VCI checks the layout of a vcall invoking object and all its polymorphic subobjects against the statically inferred layouts. The checks assert that the contents of all involved vtables and offsets of subobjects from each `this` pointer are valid. This means that the attackers cannot overlap objects unless the objects classes have a sufficient number of non-polymorphic subobjects arranged in a way that allows overlapping without disrupting the layout checks. We argue that this significantly complicates the attacks. Additionally, even if such classes were available, VCI guarantees the integrity of vtable contents which consequently prevents the invocation of desired system calls via counterfeit vtables. This limits the attackers to only invoking system calls via vfgadgets that invoke an attacker-controlled indirect call (C-style function pointer), which are "rare in practice" [34] and outside the scope of this work.

Table 3: Percentage of fully, partially, and unresolved vcalls of the C++ SPEC CPU2006 benchmarks (top) and the C++ Firefox modules (bottom).

Program	Identified Vcalls	%Resolved Vcalls		
		Fully	Partially	Unres.
444.namd	2	100%	0%	0%
447.dealII	916	37%	54%	09%
450.soplex	511	63%	12%	25%
453.povray	127	51%	23%	26%
471.omnetpp	706	47%	29%	24%
483.xalancbmk	9134	56%	28%	16%
liblgbllibs.so	63	79%	21%	0%
libmozgnome.so	206	72%	22%	06%
libmozjs.so	3784	60%	28%	12%
libxul.so	79315	32%	39%	29%
libzmq.so	133	32%	52%	16%
updater	8	75%	13%	12%
	geomean:	58%	26%	14%

If VCI fails to resolve the targets of some vcall (14% of all vcalls in Table 3, on average), it resorts to the SameOff policy for that particular vcall. This might enable an attacker to deploy a COOP attack by attacking and utilizing only the unresolved vcalls. Note that the same policy against counterfeit object overlapping is still in effect for unresolved vcalls. While attackers might be able to workaround those constraints, at least in theory, this setup is still significantly constrained compared to unprotected binaries. The reduction in attack surface is essential to heighten the cost of building a functional exploit. Complementary solutions that depend on reference and argument counts (e.g., [39, 42, 45]) can be selectively applied at unresolved vcalls sites to further shrink the possibility of data flows (see Section 6).

5.3 Performance Overhead

We benchmarked the runtime overhead of binaries protected by VCI using 1) the C++ SPEC CPU2006 benchmarks, and 2) the three industry standard browser speed benchmarks: JetStream, Kraken and Octane. The results are tabulated in Table 4. Overall, VCI incurred low overhead ranging from 2.01% to 10.69% on `namd`, `dealII`, `soplex`, and `povray`. `omnetpp` and `xalancbmk` incurred higher overhead (21.11% and 34.80%), which we believe is a side effect of alignment changes in the modified binary, as witnessed by other studies [17, 22, 37]. On browser benchmarks, VCI incurred very low overhead, ranging from 1% to 7%. Overall, VCI incurred a total average (geometric) of 7.79%. The time it took VCI to analyze each binary is tabulated separately in Table 1. We emphasize that we made no attempts to optimize the performance of VCI's analysis or policy enforcement (see Section 4.6). The overhead incurred by VCI aligns with the state-of-the-art vtable defenses (10% − 18.7% [30], 0.6% − 103%[17], 2% − 30%[22], 8% − 19.2% [37]).

Table 4: Performance overhead of VCI on the C++ SPEC CPU2006 benchmarks and three industry standard browser speed benchmarks on Firefox (median of 3 runs).

Benchmark	orig.	new	overhead
444.namd	739 s	818 s	10.69%
447.dealII	1813 s	1994 s	9.98%
450.soplex	565 s	613 s	8.50%
453.povray	399 s	407 s	2.01%
471.omnetpp	612 s	825 s	34.80%
483.xalancbmk	1047 s	1268 s	21.11%
JetStream	146.64 pt	135.81 pt	7.34%
Kraken	1332.7 ms	1358.5 ms	1.94%
Octance	27328 pt	25819 pt	5.52%
		geomean:	7.79%

6. DISCUSSION AND IMPROVEMENTS

In this section, we discuss some limitations and improvements of VCI. More technical aspects and complementary policies are discussed in Appendices E, F and H.

6.1 Position-Independent Code (PIC)

VCI supports position-independent code (PIC), including executable and shared libraries. For instance, the Firefox modules used in our experiments were all PIC. To support PIC, VCI first analyzes the binary by searching for a memory section with a `data.rel` prefix, which is the prefix used to denote relocatable data regions in binaries. If any such section is identified, VCI extracts all program counter thunks (PC thunks) in the binary. A PC thunk is a function generated by the compiler to load the current PC into a specific register when called, which allows memory accesses as an offset from the PC. VCI identifies PC thunks by searching for two-instruction functions that move the stack pointer to a register and immediately return, e.g., the function `get_pc_thunk.cx: mov (%esp),ecx; ret;` returns the PC into the `ecx` register when called. Recall that the `call` instruction pushes the address of the immediately proceeding instruction on the stack, and global data is accessed via an offset relative to the PC in PIC. Once PC thunks are identified, the analysis proceeds as normal, with the only exception that the PC value returned by PC thunks, and the PC offset, are taken into consideration when computing vtable addresses during the extraction of vtables and constructors.

6.2 Heterogeneous Containers

VCI, like any static analysis solution, has limited visibility into the semantics of the analyzed programs. Despite that VCI extracts significantly more semantics than prior solutions, there are cases where the analysis fails to identify all the class types used by a vcall. The most common case is objects stored in a heterogeneous container, e.g., a container of base pointers. Even though VCI performs alias analysis to some extent during type propagation, the analysis is conservative and cannot trace through containers logic. For example, without function names, it is not possible to determine whether a call adds or perhaps removes elements from some C++ container.

One possible approach to narrow this gap is to learn and cluster patterns of generated assembly code for common containers (e.g., the standard C++ containers). Then, identify those patterns in the assembly of analyzed programs to map out the semantics of the containers and their functions. Identifying the functions is only the first step. In addition to that, the reference to the container must be traced through procedures in order to maintain the class types that the container stores. This becomes even more complicated with nested containers. Overall, precisely bridging such semantic gaps using only static analysis remains an open, very challenging, problem.

6.3 Virtual-dispatch-like C Calls

While we have not faced any false positives during our evaluation of VCI, it is possible that some non-virtual calls resemble the behavior of a C++ vcall dispatch. For example, VCI will incorrectly idenfity the following call as a vcall: `a->b->foo(a)`, where `a` and `b` are pointers to plain C structs, and `foo` is a function pointer. It will also fail to find any constructor that defines the `this` pointer since the C struct types `a` and `b` will not have vtables. As a result, VCI will err in favor of security by limiting the target of `foo(.)` to any virtual function at the same offset of `foo` in `b`.

In vfGuard [30], the authors proposed a potential solution to this problem by looking for compiler-specific patterns in the assembly code. The authors argued that compilers tend to dispatch vcalls and nested C struct function pointers differently. However, based on our experimentation with the GCC compiler, there is no specific pattern that is used over the other. The authors of T-VIP [17] suggested recording the actual indirect call targets using a dynamic profiling pass that executes benign test cases that (optimally) cover all indirect calls. Then, filter out misidentified vcalls if a recorded target is not in a vtable.[4] However, the main challenge is in coming up with a *conclusive* benign input set that does not result in erroneous elimination and PFs at runtime. To the best of our knowledge, this remains an open research problem.

7. RELATED WORK

ASLR. Address Space Layout Randomization (ASLR) [3] is perhaps the most deployed defense against code-reuse at-

[4]The same approach could be utilized in augmenting the SameOff and AnyV policies by filtering out vcall targets that are never called by the benign inputs.

tacks. Actual deployments, however, are far from perfect, and it has been shown that various ASLR deployments can be bypassed (e.g., [31, 35]). Crane et al. [10, 11], proposed randomization based defenses resilient to memory disclosure attacks. The two approaches utilized the newly introduced execute-only memory pages via the Extended Page Tables (EPT) virtualization technology in Intel processors since the Nehalem microarchitecture. Both solutions require hardware support, kernel and compiler changes, and source recompilation. While randomization increases the attack cost by increasing the attacker's uncertainty, it only provides probabilistic guarantees.

Control Flow Integrity. Abadi et al. [1] introduced Control Flow Integrity (CFI), which prevents control flows not intended by the original program. The idea is to extract a Control Flow Graph (CFG) from the program and enforce the CFG at runtime. Unfortunately, CFI is not widely adopted in practice, because of two main hurdles: 1) building a complete CFG is a very challenging task, especially without access to source code or debug symbols; and 2) the overhead incurred by ideal CFI is rather large. Recent approaches [42, 45] attempted to address those issues by enforcing coarse-grained CFI. However, it has been shown [8, 9, 11, 13, 18, 34] that code reuse is still possible with such loose notions of CFI in place. Recently, PathArmor [38] showed that context-sensitive CFI can be enforced with little overhead using recent hardware features. However, it lacked forward-edge context sensitivity which made COOP attacks still possible. TypeArmor [39] enforced a generic binary-level policy based on the number of produced and consumed function arguments. As demonstrated by our analysis in Appendix H, such policy is imprecise compared to semantic-aware policies. Nevertheless, generic CFI solutions are complementary to our work, where we only focus on protecting the integrity of vcalls.

Compiler Solutions. Recent versions of the GCC compiler support a new vtable verification (VTV) [37] feature, which inserts checks before each vcall that asserts that the vtable pointer is valid for the invoker object type. Shrinkwrap [19] enhanced this by enforcing object-call pairing for each vcall in the program, as well as fixing a number of corner cases that were discovered in the implementation of VTV. Similary, SafeDispatch [22] extended LLVM to support a similar policy to VTV. Also for LLVM, VTrust [44] proposed a hash-based technique to verify the integrity of vcalls. For the MSVC compiler, VT-Guard [27] proposed a defense that inserted a secret cookie into each vtable and checked if the cookie is valid before each vcall. While this makes it harder for an attacker to inject a valid vtable, it falls short against memory disclosure attacks that can leak the cookie value.

Recently, Bounov et al. [6] proposed an LLVM extension that reorders vtables such that integrity policies can test for vtable membership in constant time. In general, compiler-based solutions have the maximum visibility into the source code, allowing them to enforce stronger policies than ours. Nevertheless, they require access to the source code and recompilation of all linked modules, which may not be feasible in practice. Other solutions, such as CETS [28] and Dangnull [25], attempted to eliminate dangling pointers altogether by tracing object pointers and nullifying them upon deletion. Unfortunately, sound and complete tracing of pointers is NP-Hard [24], especially with pointer aliasing and multithreading constructs available in all modern programming languages. Additionally, there are various ways to mount vtable attacks besides using a dangling pointer, such as buffer overflow, format string, and type confusion attacks. That said, eliminating dangling pointers is complementary to our work and resembles a strong layer of defense against various memory corruption attacks.

Binary Solutions. Multiple binary solutions were proposed to defend against vtable attacks. T-VIP [17] used static analysis to identify and extract vtables and vcall sites. At runtime, it checked at each vcall site that the referenced vtable is read-only and the vcall offset is in the vtable. Similarly, RECALL [14] identified unsafe casting in MSVC binaries by matching the layouts of objects that reach vcall sites. Both solutions worked on an intermediate binary representation obtained by lifting the x86 assembly to a static single assignment (SSA) form. However, as the authors explained, this is not error-free.

VTint [43] relocated vtables to a read-only memory section, and checked before every vcall that the referenced vtable is read-only. VTint incurred low overhead, but at the same time it suffered from poor identification accuracy. For instance, VTint identified only 115 vtables and 200 vcalls for 447.dealII, whereas VCI identified about 7 times as many. Similarly, vfGuard [30] used static analysis to reconstruct the set of all possible targets for each vcall site, given the vcall offset, and instrumented the binary to check for membership. Unfortunately, it was assessed that such policies are not precise enough to stand against COOP attacks [11, 34].

On a different defense front, solutions were proposed to detect memory corruption and access bugs. Valgrind [29], AddressSanitizer [32], and Undangle [7] are a few examples of dynamic memory monitoring systems that help detect memory access errors, including use-after-free. However, the overhead is prohibitive for practical deployment as a security solution (25x runtime overhead). DieHard [5] provided a probabilistic memory integrity guarantee by randomizing and expanding the heap. While it incurred much less overhead than full-blown dynamic memory monitoring, it required at least double the heap size for each program it protects, which is not feasible in practice. More recently, VTPin [33] introduced a simple and novel solution by directly managing deallocations, and preventing reuse of deleted objects by repointing their `vtptr` to a *safe* vtable. For that purpose, however, it required hooking the `free` and `malloc` calls, the presence of RTTI in the binary, as well as catching segfaults that may result from probing unmapped memory.

Complementary to our work is C++ reverse engineering efforts. In Smartdec [16], the authors proposed a system to reconstruct C++ class hierarchies from RTTI. Similarly, Objdigger [23] extracted objects and member functions of classes from compiled MSVC binaries. While decompilation is very valuable for many security problems, VCI is more tuned for vcall integrity as it focuses on only recovering the C++ semantics that impose restrictions on vcall targets.

8. CONCLUSION

This paper presented VCI, a system to generate and enforce a strict CFI policy against vtable attacks in COTS C++ binaries. VCI statically reconstructs various C++ semantics from the binaries, without needing debug symbols or type information, making it applicable to any C++ application. VCI defeats vtable injection attacks and signif-

icantly reduces the attack surface for vtable reuse attacks. As demonstrated by our experiments, VCI significantly improves upon the state-of-the-art, defeats real-world exploits, and incurs low overhead.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This material is based on work supported by the National Science Foundation (NSF) under grant no. SATC 1421747, and by the National Institute of Standards and Technology (NIST) under grant no. 60NANB16D285. Opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, NIST, or the US Government.

References

[1] M. Abadi et al. Control-flow integrity. In *Computer and Communications Security*, 2005.

[2] J. Afek and A. Sharabani. Smashing the pointer for fun and profit. *Black Hat USA*, 2007.

[3] Aslr. https://pax.grsecurity.net.

[4] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you eXecute. *ACM Trans. on Prog. Lang. and Syst.*, 32(6), 2010.

[5] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Programming Language Design and Implementation*, 2006.

[6] D. Bounov, R. G. Kıcı, and S. Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Network and Distributed System Security*, 2016.

[7] J. Caballero et al. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis*, 2012.

[8] N. Carlini et al. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, pages 161–176, 2015.

[9] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security*, 2014.

[10] S. Crane et al. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Computer and Communications Security*, 2015.

[11] S. Crane et al. Readactor: Practical code randomization resilient to memory disclosure. In *Symposium on Security & Privacy*, 2015.

[12] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with javascript. *WOOT*, 8, 2008.

[13] L. Davi et al. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[14] D. Dewey and J. T. Giffin. Static detection of C++ vtable escape vulnerabilities in binary code. In *Network and Distributed System Security*, 2012.

[15] Dyninst API. http://www.dyninst.org/dyninst.

[16] A. Fokin et al. SmartDec: Approaching C++ decompilation. In *Working Conference on Reverse Engineering*, 2011.

[17] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference*, 2014.

[18] E. Goktas et al. Out of control: Overcoming control-flow integrity. In *Symposium on Security & Privacy*, 2014.

[19] I. Haller et al. ShrinkWrap: Vtable protection without loose ends. In *Annual Computer Security Applications Conference*, 2015.

[20] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, 2005.

[21] Itanium c++ abi. http://mentorembedded.github.io/cxx-abi/abi.html.

[22] D. Jang, Z. Tatlock, and S. Lerner. Safedispatch: Securing C++ virtual calls from memory corruption attacks. In *Network and Distributed System Security*, 2014.

[23] W. Jin et al. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In *Program Protection and Reverse Engineering Workshop*, 2014.

[24] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Principles of programming languages*, 1991.

[25] B. Lee et al. Preventing use-after-free with dangling pointers nullification. In *Network and Distributed System Security*, 2015.

[26] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Annual Information Security Symposium*. Purdue University, 2010.

[27] M. R. Miller and K. D. Johnson. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities.

[28] S. Nagarakatte et al. CETS: Compiler enforced temporal safety for C. In *International Symposium on Memory Management*, 2010.

[29] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, 2007.

[30] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Network and Distributed System Security*, 2015.

[31] G. F. Roglia et al. Surgically returning to randomized libc. In *Annual Computer Security Applications Conference*, 2009.

[32] Address sanitizer. https://github.com/google/sanitizers.

[33] P. Sarbinowski et al. Vtpin: practical vtable hijacking protection for binaries. In *Annual Conference on Computer Security Applications*. ACM, 2016.

[34] F. Schuster et al. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Symposium on Security & Privacy*, 2015.

[35] J. Seibert, H. Okhravi, and E. Soderstrom. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Computer and Communications Security*, 2014.

[36] V. Srinivasan and T. Reps. Slicing machine code. Technical Report TR1824, UW-Madison, 2015.

[37] C. Tice et al. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security*, 2014.

[38] van der Veen et al. Practical context-sensitive cfi. In *Computer and Communications Security*, pages 927–940. ACM, 2015.

[39] van der Veen et al. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Symposium on Security & Privacy*, 2016.

[40] W. R. Williams, X. Meng, B. Welton, and B. P. Miller. Dyninst and MRNet: Foundational infrastructure for parallel tools. *9th Parallel Tools Workshop*, 2015.

[41] P. R. Wilson. Uniprocessor garbage collection techniques. In *Memory Management*. Springer, 1992.

[42] C. Zhang et al. Practical control flow integrity and randomization for binary executables. In *Symposium on Security & Privacy*, 2013.

[43] C. Zhang et al. VTint: Defending virtual function tables' integrity. In *Network and Distributed System Security*, 2015.

[44] C. Zhang et al. VTrust: Regaining trust on virtual calls. In *Network and Distributed System Security*, 2016.

[45] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *USENIX Security*, 2013.

APPENDIX

A. MAIN ALGORITHMS USED BY VCI

Algorithms A.1 to A.4 summarize the main algorithms used by VCI.

Algorithm A.1: Scan and Extract Vtables

```
input : Rgns: set of memory regions from the binary
        Funcs: set of functions from the binary
output: Vtables: set of virtual function tables
1  foreach func ∈ Funcs do
2      foreach insn ∈ getInstructions(func) do
3          if writesMemory(insn) then
4              src ← getSrcExpr(insn)
5              dst ← getDstExpr(insn)
6              if isDefined(src) then
7                  rgn ← getRegion(src)
8                  if readonly(rgn) then
9                      vt ← extractVtable(rgn, src)
10                     Vtables ← Vtables ∪ vt
11 return Vtables
12 Procedure extractVtable(Funcs, rgn, offset)
13     vt ← ∅
14     i ← 0
15     foreach wd ∈ rgn starting at offset do
16         if wd ∈ Funcs then
17             vt ← vt ∪⟨i, wd⟩
18             i ← i + 1
19         else
20             break
21     return vt
```

Algorithm A.2: Identify and Extract Constructors

```
input : Funcs: set of functions from the binary
        Vtables: set of virtual function tables
output: Ctors: set of constructors
1  foreach func ∈ Funcs do
2      foreach insn ∈ getInstructions(func) do
3          if writesMemory(insn) then
4              src ← getSrcExpr(insn)
5              dst ← getDstExpr(insn)
6              if isDefined(src) and getDisp(dst) = 0 and
               firstArg(dst) then
7                  vt ← Vtables[src]
8                  if vt ≠ ∅ then
9                      if getOffset(func) ∉ vt then
10                         // ctor cannot be in vt
11                         Ctors ← Ctors ∪ func
```

B. EXAMPLE SNIPPET AND POLICY

Figure B.1 shows an example C++ program, its corresponding assembly dump, and the policy semantics injected by VCI at the vcall site. The corresponding filtered PDG generated by VCI is shown in Figure B.2.

Algorithm A.3: Reconstruct Class Layout

```
input : cls: initial class layout
        Ctors: set of constructors
output: cls: populated class layout
1  offset ← 0
2  foreach insn ∈ getInstructions(cls.ctor) do
3      if isCall(insn) then
4          dst ← getCallTarget(insn)
5          if dst ∈ Ctors then
6              mThis ← findThis(dst)
7              offset ← calcOffset(cls.this, mThis)
8              deref ← storesThis(cls, mThis)
9              addToLayout(cls, offset, dst, deref)
10 return cls
```

Algorithm A.4: Intra-procedural Type-Vcall Pairing

```
input : Ctors: set of constructors
        Funcs: set of functions from the binary
result : Pairing information between classes and vcalls
1  foreach func ∈ Funcs do
2      foreach call ∈ func do
3          foreach param ∈ findParams(call) do
4              slice ← backwardSlice(param)
5              foreach entryNode ∈ slice do
6                  def, Adjs ← reaches(entryNode, exitNode)
7                  if def ≠ ∅ then
8                      cls ← resolve(def, Adjs, func, Ctors)
9                      pair(cls, param, call, Adjs)
10                 else
11                     pair(∅, param, call, Adjs)
```

C. ABI DEPENDENCY

The C++ Application Binary Interface (ABI) sets the interface between program modules and the execution environment at the assembly level. It defines things such as the memory layout of objects, details of how virtual functions are invoked, and the behavior of the linking stage. The most adopted C++ ABI is the Itanium ABI [21], which is the focus of this work. The Itanium ABI is used by all Linux compilers. Alternatively, the MSVC compiler on Windows uses the MSVC ABI which was internally developed by Microsoft. The two ABIs mainly differ in their choice of calling conventions and the layouts of vtables in memory. Nevertheless, the approach discussed in this paper can also be applied to MSVC C++ binaries by adjusting the algorithms to accommodate the rules of MSVC.

D. WHY NOT DEPEND ON RTTI?

C++ supports dynamic type reporting, i.e., identifying and checking the *actual* type of an object at runtime (as opposed to at compile-time). This is enabled by what the ABI calls "Runtime Type Information" (RTTI). RTTI enables the program to dynamically identify and cast objects at runtime, via the `typeid` and the `dynamic_cast` operators, respectively. For each polymorphic class, an RTTI record is added to the class layout in memory, and a pointer to that

```
1 int main() {
2   int x; cin >> x;
3   Base *ptr = nullptr;
4   if (x == 1) ptr = new A();
5   else ptr = new B();
6   ptr->foo(); //vcall
7 }
```

(a) Example C++ program with a virtual call.

```
77d:  push   %ebp
...
79b:  movl   $0x0, 0x1c(%esp)    ; ptr = nullptr
...
; compare x
7a7:  cmp    $0x1, %eax          ; x == 1 ?
7aa:  jne    7ce <main+0x51>
; if x == 0:
7ac:  movl   $0x4, (%esp)
7b3:  call   660                 ; operator new()
7b8:  mov    %eax, %ebx
7ba:  movl   $0x0, (%ebx)
7c0:  mov    %ebx, (%esp)
7c3:  call   8aa                 ; constructor of A
7c8:  mov    %ebx, 0x1c(%esp)    ; this ptr of A
7cc:  jmp    7ee <main+0x71>
; if x == 1:
7ce:  movl   $0x4, (%esp)
7d5:  call   660                 ; operator new()
7da:  mov    %eax, %ebx
7dc:  movl   $0x0, (%ebx)
7e2:  mov    %ebx, (%esp)
7e5:  call   8c6                 ; constructor of B
7ea:  mov    %ebx, 0x1c(%esp)    ; this ptr of B
; vcall site
7ee:  mov    0x1c(%esp), %eax    ; %eax = this ptr
7f2:  mov    (%eax), %eax        ; %eax = vptr
7f4:  mov    (%eax), %eax        ; %eax = vptr[0]
7f6:  mov    0x1c(%esp), %edx
7fa:  mov    %edx, (%esp)        ; push this ptr
7fd:  call   *%eax               ; invoke vptr[0]
...
```

(b) Assembly dump of (a).

```
clsz ← {A,B};
ptr ← (%esp);
assert(∃ cls ∈ clsz
       ∧ vtable(ptr) == cls.vtable
       ∧ layout(ptr) == cls.layout);
```

(c) Injected policy checks before the vcall at `7fd`.

Figure B.1: (a) Example C++ program, (b) it's assembly dump, and (c) the policy injected by VCI. `clsz` is the statically constructed set of valid classes at the vcall site. `ptr` refers to the pointer at address `0x1c(%esp)`.

record is included at a negative offset in the vtable. The RTTI record contains several structures that describe the class type and its bases.

The structural details of RTTI records can be very useful in reconstructing the polymorphic class hierarchy. However, RTTI is not required if the program uses an RTTI operator in a way that the compiler can infer at compile-time. For example, a dynamic up cast to an unambiguous base can be replaced by a static (compile-time) cast by the compiler, hence not requiring RTTI. All major compilers support RTTI as an optional feature that can be enabled or disabled. Some compilers, such as Clang/LLVM, use alternative implementations of RTTI via C++ templates (`dyn_cast<>, isa<>` in Clang). Additionally, RTTI is typically stripped from COTS binaries or is not present to begin with. For instance, the

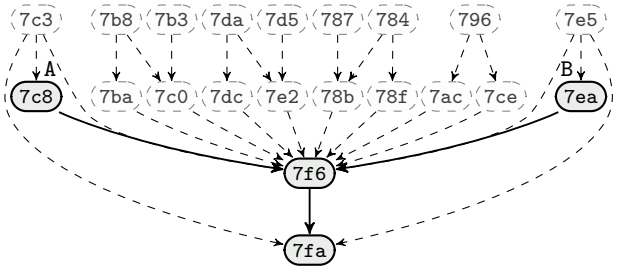

Figure B.2: Extracted PDG backward slice starting at the `this` pointer argument at offset `7fa` in Figure B.1. For simplicity, nodes only show the corresponding instruction offsets. Dashed nodes and edges correspond to irrelevant dependences that were filtered out by VCI.

C++ Firefox modules on Ubuntu that we used in our experiments had no RTTI by default. Finally, the details of the RTTI record are compiler-specific, and the ABI does not mandate an implementation standard for compilers to follow. Therefore, we opted against depending on RTTI in VCI.

E. DESTRUCTORS CORNER CASES

The Itanium ABI defines three different types of destructors: 1) base destructor, which destroys the object itself, data members, and non-virtual base subobjects; 2) complete destructor, additionally destroys virtual base subobjects; and 3) deleting destructor, which in addition to performing a complete destruction, calls operator `delete` to free the object's memory. Since base destructors do not call non-virtual bases, they do not reference any vtable and therefore are always ignored by VCI. Deleting destructors are also ignored since they call complete destructors and do not reference vtables. Complete destructors, on the other hand, have to call the virtual destructors of base classes. Therefore, they access the vtable of the object and its subobjects, in a somewhat similar behavior to constructors. Algorithm A.2 implicitly assumes that all complete destructors are virtual. While that is true most of the time, there are a few exceptions to this rule.

For instance, the C++11 ABI added a `final` specifier that can be applied to classes. A class that is marked final cannot be inherited from (C++11 Clause 9.3). A final class can have a non-virtual complete destructor even though it defines or inherits virtual functions. This would cause VCI to incorrectly identify those destructors as constructors. However, the first thing a complete destructor does is store the vtable address of its class in the object's memory. This is done *before* calling base destructors, if any. Therefore, VCI will not identify any base classes when analyzing the destructor site, compared to analyzing a constructor, when extracting inheritance relationships. VCI utilizes this disagreement in the identified "is-a" relationship to filter out non-virtual complete destructors (if any).

F. CROSS-MODULE POLYMORPHISM

A C++ binary can use or inherit a class that is defined in a different module (shared library). In this case, space for the vtable of the shared class is reserved in the `.bss` section of the binary, but the contents of the vtable are not

present until after the dynamic linker populates the `.bss` section. Similarly, in the case of cross-module inheritance, the derived class vtable may contain pointers to the PLT (Procedure Linkage Table), where the actual addresses of the base functions are to be determined at runtime by the dynamic linker. In both cases, VCI applies the SameOff policy since it has limited visibility into the shared vtables and the virtual function bodies. This gap can be narrowed via cross-module inter-procedural analysis, and a runtime stage, similar to VTV, that adjusts the policy as modules are loaded and the contents of the vtables become available. We leave this extension for future work.

G. VCALL RESOLUTION STATISTICS

Table G.1 provides summary statistics of VCI's vcall target resolution results of the programs used in our experiments. The statistics represent the number of vcall targets per vcall site for each of the three policy cases.

Table G.1: Vcall target resolution statistics.

Program	Fully			Partially			Unres.		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
444.namd	1	1	1	0	0	0	0	0	0
447.deal	1	31	18	2	76	43	91	335	207
450.soplex	1	7	5	2	14	9	16	30	17
453.povray	1	11	7	4	22	13	24	30	26
471.omnetpp	1	28	19	2	51	39	49	82	68
483.xalanch	1	49	31	3	112	81	213	705	283
libgbllibs.so	1	3	1	2	5	4	0	0	0
libmozgnome.so	1	12	8	2	18	11	18	24	19
libmozjs.so	1	62	33	8	192	83	201	917	402
libxul.so	1	208	74	12	987	646	1104	10021	2619
libzmq.so	1	9	4	4	23	11	30	54	37
updater	1	1	1	3	3	3	7	7	7

H. COMPARISON TO RELATED POLICIES

H.1 Reference Counts

It is possible to further strengthen the policies enforced by VCI via means of reference counting [41]. For instance, a vcall can never be invoked on a class type that has zero referenced instances. While this may result in additional reduction in the attack surface, the reference counters are vulnerable to memory corruption attacks since they have to reside in writable memory. Thus, VCI does not use reference counters.

H.2 Calling Convention

Though VCI handles the `stdcall` convention by default, developers could set specific calling conventions, such as `thiscall` and `fastcall`, for some virtual functions. This results in discrepancies in how arguments are passed to vcalls: `stdcall` passes arguments on the stack, `thiscall` passes only the `this` pointer in `ecx`, while `fastcall` passes the first two arguments in `ecx` and `edx`. By identifying the calling convention at each vcall site, it is possible to filter out target virtual functions that do not adhere to the same calling convention. Care must be taken to precisely distinguish overlapping conventions, such as `thiscall` and `fastcall`. This policy was applied by Prakash et al. [30], but it yielded minimal precision improvements (<1%).

H.3 Call Arity

In C++, polymorphs of a function *must have* the same parameters type list (C++14 Clause 10.3.2). This implies that they must also have the same arity, i.e., accept the same number of arguments. Therefore, it seems plausible to use the number of arguments passed to a vcall site to filter out potential target virtual functions that cannot accept that number of arguments. However, *exact* argument matching will be unsound, since at the binary level, only *consumed* parameters rather than *accepted* arguments are present. Additionally, as per the ABI, the `this` pointer is passed to class member functions regardless of whether the functions consume the `this` pointer or not. This discrepancy in the number of passed (prepared) arguments and the number of consumed parameters makes such policies unsound, as legitimate targets may be incorrectly eliminated if function polymorphs consume (use) a different number of arguments.

As a result, exact matching has to be relaxed by allowing *compatible* arguments, i.e., icall sites that prepare N arguments can target functions that consume *less than or equal* to N arguments. This policy was recently applied by TypeArmor, by van der Veen et al. [39], to protect indirect calls in both C and C++ binaries. TypeArmor, however, does *not* take the C++ semantics into consideration. Though the compatible arguments policy is sound, it is *less* precise than semantic-aware policies, as noted by the authors.

To evaluate how imprecise this policy is compared to VCI, we parsed the assembly dump of libxul.so, and counted the number of prepared arguments at each icall site as well as the number of accepted arguments by each function.[5] We then computed the number of compatible target functions per icall site, grouped by the number of prepared arguments. Figure H.1 depicts the results. For the sake of this argument, assume the best case scenario where any vcall site prepares only one argument (the `this` pointer). That means there are 188k compatible targets per vcall (functions that accept one or zero arguments). This is approximately 188× more targets per vcall than VCI. Even if, hypothetically speaking, policy refinements applied by TypeArmor would reduce that by 90%, there would still be 18k targets per vcall, about 18× less precise than VCI. Hence, we conclude that generic policies based on call arity cannot replace C++ semantic-aware policies. This, of course, does not nullify the fact that layering multiple policies helps reduce the attack surface and is essential for complete protection at the binary level.

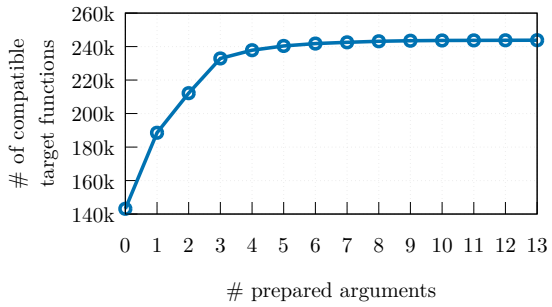

Figure H.1: Number of compatible target functions in libxul.so for a given number of prepared arguments at icall sites, under TypeArmor's [39] policy.

[5]The reported counts are underestimates as we ignored unused and variable length arguments.