Boosterg the Precision of Virtual Call Integrity Protection with Partial Pointer Analysis for C++

Xiaokang Fan  
UNSW Sydney, Australia  
Xiangke Liao  
National University of Defense Technology, China

Yulei Sui  
UNSW Sydney, Australia  
Jingling Xue  
UNSW Sydney, Australia

ABSTRACT

We present, Vip, an approach to boosting the precision of Virtual call Integrity Protection for large-scale real-world C++ programs (e.g., Chrome) by using pointer analysis for the first time. Vip introduces two new techniques: (1) a sound and scalable partial pointer analysis for discovering statically the sets of legitimate targets at virtual callsites from separately compiled C++ modules and (2) a lightweight instrumentation technique for performing (virtual call) integrity checks at runtime. Vip raises the bar against viable hijacking attacks by providing stronger security guarantees than the CHA-based approach with comparable performance overhead.

Vip is implemented in LLVM-3.8.0 and evaluated using SPEC programs and Chrome. Statically, Vip protects virtual calls more effectively than CHA by significantly reducing the sets of legitimate targets permitted at 20.3% of the virtual callsites per program, on average. Dynamically, Vip incurs an average (maximum) instrumentation overhead of 0.7% (3.3%), making it practically deployable as part of a compiler toolchain.

CCS CONCEPTS

- Software and Its Engineering → Automated Static analysis; Object-Oriented Languages; Security and Privacy → Software Security Engineering;

KEYWORDS

Pointer Analysis, CFI, VTable Hijacking Attacks

ACM Reference format:

1 INTRODUCTION

As a low-level object-oriented language, C++ is a primary choice for implementing a wide variety of system software (e.g., web browsers and language runtimes) to achieve both performance and abstraction. Modern compilers, such as LLVM and GCC, implement C++’s dynamic dispatch using virtual tables (vtables), each of which contains the pointers to the virtual functions of a class. A virtual call invoked on an object of a class is dispatched via one extra level of indirection. First, the vtable for the class is retrieved from a vtable pointer (vtptr) stored in the object. Then, the vtable is looked up to determine the virtual function in the class that should be called.

Unlike managed languages such as Java, C++’s low-level vtable representation enables faster dynamic dispatch, but is vulnerable to attacks due to the absence of memory safety. As an increasingly popular attack, vtable hijacking [45, 50] is presently receiving much attention. Despite widely deployed mitigation techniques like stack canaries [11], Data Execution Prevention (DEP) [3] and Address Space Layout Randomization (ASLR) [4], attackers can still hijack vtables [9, 21, 51]. By first exploiting a memory corruption bug (e.g., use-after-free) in the program to overwrite the vtptr of an object, an attacker can redirect the program to a chosen location (e.g., a system call) through a counterfeit or an existing vtable whenever that compromised object calls one of its virtual functions.

A heavyweight approach to virtual call integrity protection is to enforce full memory safety [13, 30–32]. This ensures that no dangling or out-of-bounds pointers can be read or written by the program, thus preventing the attacks in its first step but incurring a 2x – 4x performance slowdown with respect to the native runs [22].

Recent solutions protect virtual call integrity by using a lightweight Control-Flow Integrity (CFI) [6] approach, at either the source code [15, 36, 47, 51] or binary code [21, 33, 46, 50]. In general, source-level CFI provides stronger security guarantees by considering high level C++ semantics, making sophisticated attacks (such as COOP [37]) harder [24, 46, 50]. The enforcement is done by inserting runtime checks before a virtual call to validate the control flow transfers based on a set of statically computed target functions by using Class Hierarchy Analysis (CHA) [9, 12, 46].

Insights. Due to dynamic binding, a pointer to an object declared to have the type of class A (static type) may actually point to an object of type A or one of its subtypes (dynamic type) at runtime. Figure 1(a) gives a virtual call invoked through a pointer p of a static type A, with the related object layout depicted in Figure 1(b). Note that B is a subclass of A but C is not type-related with A and B. As highlighted in red in Figure 1(c), the CHA-based CFI [21, 46] over-approximates the set of dynamic types of the objects pointed to by p as {A, B}, i.e., the set of types in A’s class hierarchy. Thus, A::f and B::f are the two legitimate functions allowed to be called. By excluding C::f, the CHA-based approach is more precise than C++-unaware approaches [6, 21, 46], but still not precise enough, since A::f is spurious but still considered to be legitimate at runtime.
Therefore, the CHA-based approach still provides an attacker a lot of opportunities to launch vtable hijacking attacks by using virtual functions that are considered as legitimate but spurious. The problem is amplified in large C++ applications, e.g., Chrome, which relies heavily on OO features, such as inheritance. In Chrome, the largest class hierarchy has over 9000 classes, and a virtual call may be resolved to have over 2000 virtual functions by CHA. This leads to substantial attack surface and overhead [17]. In general, the more precisely the virtual call targets are resolved by CHA, the more secure the virtual call integrity protection will be, thus raising the bar against sophisticated attacks (e.g., reducing the number of virtual function gadgets [37] for code reuse attacks).

**Challenges.** Accurately resolving virtual calls requires an interprocedural pointer analysis. However, analyzing large-scale C++ programs, such as Chrome, with over 15 MLOC, poses a big challenge to the existing “scalable” pointer analyses that work only for C [18, 19, 26, 39, 40, 48, 49]. C++ Programs tend to have more indirect calls in the form of polymorphic virtual calls. C++ templates and STL promise generic containers and algorithms, but also cause compilers to generate a huge amount of low-level code (e.g., LLVM IR), which significantly complicates pointer analysis further.

Another challenge is that existing pointer analyses [8, 19, 25, 26, 39, 40, 48, 49] are “whole-program” analyses with the assumption that the entire program is compiled into one single module. However, many programs are often composed of separately compiled modules. For example, Chrome has one executable (5.9 MLOC) with 131 dynamically linked libraries (9.3 MLOC). Whole-program pointer analyses are unscalable when applied to such large codebases or unsound when applied to their modules separately. To enforce CFI, the targets at a virtual callsite must also be resolved soundly. Missing any target may result in false protection that forces a program to crash during runtime CFI checks.

A final challenge faced is to develop a lightweight instrumentation mechanism to protect virtual calls with negligible overhead.

**Our Solution.** This paper presents VIP, an approach to boosting the precision of Virtual call Integrity Protection for large-scale C++ programs (e.g., Chrome) by introducing a new partial pointer analysis for discovering statically the sets of legitimate targets at virtual callsites and a lightweight instrumentation technique for performing CFI checks at virtual callsites. VIP achieves a comparable low overhead with the state-of-the-art CHA-based solutions (e.g., VTV [46], Clang VCFI [2] and OVT/Ivt [9]), but offers better precision, thus raising the bar against vtable hijacking attacks.

One key technique of VIP is a partial pointer analysis for handling separately compiled modules in a C++ program. Our analysis allows spurious target functions introduced at a virtual callsite by CHA to be significantly reduced. To boost efficiency while achieving soundness, VIP first applies a pre-analysis using a fast CHA [9, 12, 46] by gathering the class hierarchy information from the entire program. Our partial analysis handles a single module by introducing type-based unknown objects $u_t$ wherever the code analyzed is incomplete. Each $u_t$ represents a set of unknown objects whose types are $t$ or subtypes of $t$. The crux of our analysis is to infer $t$ by leveraging the existing type information available in the code, so that the precise type information can be used to filter out the spurious targets introduced by CHA for a virtual callsite. To enhance precision further, VIP also supports cross-module analysis by creating fine-grained unknown objects in selected modules (e.g., Chrome executables), facilitated by building a lightweight type summary of helper modules (e.g., Chrome libraries).

Compared to the CHA-based approach, our partial pointer analysis can reduce not only the set of legitimate targets permitted at a virtual callsite but also the runtime lookup overhead incurred [21, 46]. However, developing an efficient set membership test at runtime becomes non-trivial. A recent work, OVT/Ivt [9], reorders/interleaves the memory layout of the vtables in the same class hierarchy to make their addresses continuous and thus simplifies a set membership test to a cheap range check. However, this technique is inapplicable here. As VIP removes many spurious virtual functions permitted by CHA, an efficient single-range check has been turned into inefficient multiple-range checks. To protect virtual calls with our partial pointer analysis, another key technique of VIP is a simple yet efficient index-based instrumentation, which allows a set membership test at any callsite to be realized efficiently by one single-range check, followed possibly by one bit-wise element test.

In summary, the contributions of this paper are as follows:

- **We present VIP, a new Virtual call Integrity Protection for large-scale real-world C++ programs by combining static pointer analysis with dynamic instrumentation.**
  - We introduce a sound and scalable partial pointer analysis for discovering the sets of legitimate targets at virtual call sites from separately compiled modules.
  - We introduce a fast index-based instrumentation mechanism, which works for any complex class hierarchy (without the need for reordering or interleaving vtables), for performing CFI checks at virtual call sites.

- **We have implemented VIP fully in LLVM-3.8.0 and evaluated it using SPEC programs and Chrome.** Statically, VIP protects virtual calls more effectively than CHA by significantly reducing the sets of legitimate targets permitted at 20.3% of the virtual call sites per program, on average. Dynamically, VIP incurs an average (maximum) instrumentation overhead of 0.7% (3.3%), making it practically deployable as part of a compiler tool chain.
2 BACKGROUND

In this section, we introduce the background on virtual call integrity, including C++ dynamic dispatch, the threat model adopted, vtable hijacking, and CHA-based protection techniques.

C++ Dynamic Dispatch. In C++, a base class (e.g., A) and derived classes (e.g., B) can have virtual functions with the same function signature (i.e., name, argument types and qualifiers). The actual runtime function invoked on an object is decided through dynamic dispatch at runtime, depending on the dynamic type of the receiver object. Modern compilers (e.g., LLVM and GCC) implement dynamic dispatch on an object of a class via a vtable containing the pointers to the virtual functions of the class. For an object of a class, a vtable pointer (vtptr) that points to its vtable is stored in the object. A virtual call on a pointer is dispatched in four steps (Figure 1(c)): (1) obtaining vtptr by dereferencing the pointer to the object, (2) obtaining &vtptr[idx] of the entry in the vtable at a designated offset idx for the target function, (3) loading the address of the function from vtptr[idx], and (4) calling the function.

Threat Model. We consider a common threat model that is consistent with prior work in virtual call integrity protection [46, 50]. The attacker can exploit an existing memory corruption vulnerability in the program to read arbitrary areas of memory and write to all writable memory addresses. The program does not contain self-modifying code, and data execution prevention (DEP) [3] is in place, so the attacker is unable to write into executable memory or inject code for execution. Auxiliary protections for return instructions (e.g., shadow stack) are assumed to be deployed.

Attacks and Defenses. All vtables are placed in the read-only section of an executable, but pointers (vtptr) to them reside in objects that are writable. Vtable hijacking attacks first exploit a memory corruption bug in a program to overwrite a vtptr of an object and then make this vtptr point to a location of the attacker’s choice (e.g., a counterfeited or an existing vtable). For example, in Figure 1(b), by corrupting object new B, the attacker can redirect its vtptr from VTableB to VTableA or VTableC, then p->f() will invoke A::f or C::f, which may contain sensitive calls (e.g., system()).

Existing protection techniques [21, 46, 50] place checks before a virtual call to validate control-flow transfers using CHA [12], which collects the subtyping relations of a hierarchy of classes and computes conservatively a set of legitimate targets for each virtual call site by traversing the class hierarchy. Compared with pointer analysis, CHA is fast but imprecise as it does not need to analyze the implementation of a class, e.g., reason about the flow of a function pointer (from its creation to a use). Thus, in Figure 1, a spurious call to A::f at p->f() is still allowed by CHA.

3 THE VIP APPROACH

We first describe our program representation of a C++ program, then introduce our inclusion-based interprocedural partial pointer analysis for analyzing incomplete programs, and finally, discuss our instrumentation and runtime checking technique to protect virtual calls in terms of the computed points-to information.

3.1 Program Representation

We choose LLVM’s partial SSA form to represent a program by following [8, 19, 25, 48, 49]. As shown in Table 1, the set of all variables $V$ in the program are separated into two subsets: $O$ contains all possible targets, i.e., abstract objects or address-taken variables of a pointer, and $P$ contains all top-level pointers.

In LLVM-IR, top-level pointers in $P = S \cup G$, including stack virtual registers (symbols starting with “%”) and global pointer variables (symbols starting with “@”), are explicit, i.e., directly accessed. Abstract objects in $O$ are implicit, i.e., accessed indirectly at LLVM’s load or store instructions via top-level pointers.

We distinguish two kinds of abstract objects, $O = A \cup U$. For a complete program, every abstract object $a$ in $A$ is identified by its memory allocation (global, stack, heap or function). In our notation, $a_f$ denotes a function object $f$, $a.fld$ represents the subobject of an object $a$ that corresponds to its field fld, and $a[c]$ denotes a subobject that corresponds to an array element of a indexed by a constant $c$. For convenience, we write $a[x]$ to denote any subobject of a (except for a function) at a variable index. For an incomplete program, we use a type-based unknown object $u_f$ in $U$ to represent any object of type $t$ or a subtype of $t$ when $t$ is a class/struct type.

Table 1 also gives LLVM IR. A program comprises a number of modules with each module $M$ consisting of global variables and function definition. The body of function $f$ contains 11 types of instructions. ADDRESSOF $p = \&a$ models an allocation site. LOAD and STORE represent read and write operations of address-taken variables, respectively. CAST denotes a casting instruction, e.g., bitcast. PHI is a standard SSA instruction introduced at a control flow confluence point. To analyze precisely a vtable (an array of function pointers) and vtptr (a field of an object), we model array and field accesses using ARRAY-C, ARRAY-V and FIELD. As in prior work [8, 35], our handling of field- and array-sensitivity is ANSI-compliant [20]. A direct call is represented by CALL-D. An indirect call is represented by CALL-I, where $fp$ is a function pointer. For a C++-style virtual call, $p_f$ points to its receiver object. Finally, returnf $q$ denotes a return instruction in function $f$.

<table>
<thead>
<tr>
<th>Table 1: Domains and LLVM IR used by pointer analysis.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Domains</td>
</tr>
<tr>
<td>$\mathcal{T}$ ∈ $\mathcal{F}$</td>
</tr>
<tr>
<td>$c, \text{fld}$ ∈ $\mathcal{C}$</td>
</tr>
<tr>
<td>$t$ ∈ $\mathcal{F}$</td>
</tr>
<tr>
<td>$p, q, \text{ret}$ ∈ $\mathcal{S}$</td>
</tr>
<tr>
<td>$g$ ∈ $\mathcal{G}$</td>
</tr>
<tr>
<td>$a, a_r, a.fld, a[c] ∈ \mathcal{A}$</td>
</tr>
<tr>
<td>$u_t$ ∈ $\mathcal{U}$</td>
</tr>
<tr>
<td>$o$ ∈ $\mathcal{O} = \mathcal{A} \cup \mathcal{U}$</td>
</tr>
<tr>
<td>$v$ ∈ $V = P \cup O$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LLVM IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Prog} := M$</td>
</tr>
<tr>
<td>$M := \mathbb{M}$</td>
</tr>
<tr>
<td>$m := g</td>
</tr>
<tr>
<td>$\text{inst} := p = &amp;a$</td>
</tr>
<tr>
<td>$p = \ast q$</td>
</tr>
<tr>
<td>$s = q$</td>
</tr>
<tr>
<td>$p = (t) q$</td>
</tr>
<tr>
<td>$p_1 = \text{phi}(p_1, p_2)$</td>
</tr>
<tr>
<td>$p = &amp;(q \rightarrow \text{fld})$</td>
</tr>
<tr>
<td>$p = &amp;[q][c]$</td>
</tr>
<tr>
<td>$\text{ret} = f(p_1, \ldots, p_n)$</td>
</tr>
<tr>
<td>$\text{returnf } q$</td>
</tr>
</tbody>
</table>

331
### 3.2 Partial Pointer Analysis for C++

#### CPA’s Unsoundness for Incomplete Code

CPA’s unsoundness arises when analyzing three incomplete code patterns in a module $\mathcal{M}$. (1) A call instruction $ret = f(p_1, \ldots, p_n)$ in $\mathcal{M}$ may call a function $f$ defined in another module. Any object (together with its fields) and array elements passed indirectly through arguments $p_1, \ldots, p_n$ may be modified outside $\mathcal{M}$, and $ret$ may also point to objects not defined in $\mathcal{M}$. (2) A function $f(p_1, \ldots, p_n)$ defined in $\mathcal{M}$ may be called by a function in another module. $f$’s formal parameters $p_1, \ldots, p_n$ may point to incomplete points-to targets. (3) A global variable is used in $\mathcal{M}$ but defined in another module.

CPA is unsound due to missing pointed-to targets in some pointerto sets (line 13). For an allocation-based object $u$, its points-to set of $u$ points to objects of type $B$ and $C$. Thus, the spurious target $A \times f$ introduced by CPA is removed.

PPA further improves the precision of CPA by building a lightweight type summary for $\text{lib.bc}$ to indicate that $\text{getC}$ does not modify any type of $B^*$. Thus, there is no longer a need to add $u$ to the points-to set of $c$. This way, $p$ is found to point soundly to objects of type $C$ only, as illustrated in Figure 3(e).

#### Inference Rules

Figure 4 gives our inference rules for CPA and CPA$_{\text{Summary}}$. In LLVM SSA form, a pointer $p \in \mathcal{P}$ is defined uniquely to have one single static type at its declaration while an object $c \in \mathcal{A}$ that is created at an allocation site may have multiple (dynamic) types due to cast operations applied to the object. For a pointer $p \in \mathcal{P}$, we use $\mathcal{T}(p)$ to denote its pointer type and $\mathcal{D}(p)$ to denote its pointee type. For example, given a pointer declaration $B^* p$, we have $\mathcal{T}(p) = B^*$ and $\mathcal{D}(p) = B$. For an allocation-based object $a \in \mathcal{A}_{\mathcal{T}}$, $\mathcal{ts}(a)$ denotes the set of (dynamic) types that $a$ may have.

Below we examine our inference rules for a module $\mathcal{M}$ in Figure 4. We write $\mathcal{pt}(v)$ to represent the points-to set of a variable $v \in V$. The first 13 rules ($A^*$ inference rules) aim to resolve $\mathcal{ts}(a)$ for an object of $\mathcal{C}$ is unavailable. As $\text{setData}$ is also not available for analysis, both $x \rightarrow q$ and $p$ also have an empty points-to set each. Thus, no function is found to be invokable at $p \rightarrow f()$. So CPA is unsound.

#### 3.2.2 Partial Pointer Analysis (PPA)

PPA is described below.

### Pre-analysis

To enable PPA to achieve soundness and scalability for a single module, we first apply a lightweight pre-analysis consisting of two phases: (1) A CHA phase [9, 12] that obtains the subclasses of every class across all modules, together with the prototype declarations for all the virtual functions in a class, and (2) an escape analysis that computes $\mathcal{escMs}(g)$ ($\mathcal{escMs}(f)$), the set of modules that uses (calls) a global pointer variable $g$ (a function $f$).
Boosting the Precision of Virtual Call Integrity
Protection with Partial Pointer Analysis for C++

Analysis | Target Function(s) | Soundness
---------|---------------------|---------
CPA      | [A-f, B-f, C-f]     | ✓       |
PPA      | [B-f, C-f]          | ✓       |
PPA_{summary}  | [C-f]               | ✓       |

(c) Target functions at virtual call p->f()

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Points-to Set</th>
<th>Source Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(x)</td>
<td>[a]</td>
<td>[A-ALLOC]</td>
<td>11</td>
</tr>
<tr>
<td>p(inp)</td>
<td>[a]</td>
<td>[A-STORE]</td>
<td>14</td>
</tr>
<tr>
<td>p(a,q)</td>
<td>[uc, un]</td>
<td>[U-CALL-D]</td>
<td>16</td>
</tr>
<tr>
<td>p(np)</td>
<td>[uc, un]</td>
<td>[U-LOAD]</td>
<td>17</td>
</tr>
<tr>
<td>p(vtptr)</td>
<td>[uoid(A)']</td>
<td>[U-LOAD]</td>
<td>18</td>
</tr>
<tr>
<td>p(vfn)</td>
<td>[uoid(A)']</td>
<td>[U-ARRAY]</td>
<td>19</td>
</tr>
<tr>
<td>p(t1)</td>
<td>[uoid(A)']</td>
<td>[U-LOAD]</td>
<td>20</td>
</tr>
</tbody>
</table>

Resolve(p1) = [B-f, C-f] [U-LOAD]

(d) Points-to sets inferred by PPA

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Points-to Set</th>
<th>Source Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(x)</td>
<td>[a]</td>
<td>[A-STORE]</td>
<td>15</td>
</tr>
<tr>
<td>p(np)</td>
<td>[uc, un]</td>
<td>[A-STORE]</td>
<td></td>
</tr>
</tbody>
</table>

Resolve(p1) = [B-f, C-f] [U-LOAD]

(e) Points-to sets inferred by PPA_{summary}

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Points-to Set</th>
<th>Source Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(x)</td>
<td>[a]</td>
<td>[A-STORE]</td>
<td>15</td>
</tr>
<tr>
<td>p(np)</td>
<td>[uc, un]</td>
<td>[A-STORE]</td>
<td></td>
</tr>
</tbody>
</table>

Resolve(p1) = [B-f, C-f] [U-LOAD]

Figure 3: An example illustrating partial pointer analysis with two modules main.bc and lib.b. The analysis will focus only on main.bc with the function bodies of getc and setdata in lib.b being unavailable to main.bc.

Figure 4: Partial pointer analysis for analyzing a module M without and with side-effect summary(f) of a function f (Figure 6).

allocation-based object a ∈ A while the last 11 rules (U-* inference rules) introduce unknown objects in U with their types inferred from the types of pointers in P and allocation-based objects in A.

(1) A-* inference rules. [A-GLOBAL], [A-STACK] and [A-Heap] handle global, stack and heap object allocations, respectively. For a global or stack object, its initial type is read off from its allocation site. For a heap object, its initial type is void and thus ignored; its concrete types will be deduced at later CAST instructions. In addition to propagating the points-to information, [A-CAST] adds the casting type T to ts(a). [A-Function] creates an object for a function f whose address is taken by pointer p for making indirect calls. [A-Phi] propagates the points-to targets o ∈ A ∪ U between pointers. [A-Load] and [A-Store] handle loads and stores for address-taken objects in A. [A-Field], [A-Array-C] and [A-Array-V] realize field- and array-sensitivity. For p & q → Fld or p = &q[,], where q points to a base object a ∈ A, p is made to point to the sub-object a.Fld or a[.]. In LLVM IR, a high-level statement p->f = in

(a) C++ source files

(b) Two modules in the form of LLVM IR

(c) Target functions at virtual call p->f()

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Points-to Set</th>
<th>Source Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(x)</td>
<td>[a]</td>
<td>[A-ALLOC]</td>
<td>11</td>
</tr>
<tr>
<td>p(inp)</td>
<td>[a]</td>
<td>[A-STORE]</td>
<td>14</td>
</tr>
<tr>
<td>p(a,q)</td>
<td>[uc, un]</td>
<td>[U-CALL-D]</td>
<td>16</td>
</tr>
<tr>
<td>p(np)</td>
<td>[uc, un]</td>
<td>[U-LOAD]</td>
<td>17</td>
</tr>
<tr>
<td>p(vtptr)</td>
<td>[uoid(A)']</td>
<td>[U-LOAD]</td>
<td>18</td>
</tr>
<tr>
<td>p(vfn)</td>
<td>[uoid(A)']</td>
<td>[U-ARRAY]</td>
<td>19</td>
</tr>
<tr>
<td>p(t1)</td>
<td>[uoid(A)']</td>
<td>[U-LOAD]</td>
<td>20</td>
</tr>
</tbody>
</table>

Resolve(p1) = [B-f, C-f] [U-LOAD]

(d) Points-to sets inferred by PPA

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Points-to Set</th>
<th>Source Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(x)</td>
<td>[a]</td>
<td>[A-STORE]</td>
<td>15</td>
</tr>
<tr>
<td>p(np)</td>
<td>[uc, un]</td>
<td>[A-STORE]</td>
<td></td>
</tr>
</tbody>
</table>

Resolve(p1) = [B-f, C-f] [U-LOAD]

(e) Points-to sets inferred by PPA_{summary}

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Points-to Set</th>
<th>Source Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(x)</td>
<td>[a]</td>
<td>[A-STORE]</td>
<td>15</td>
</tr>
<tr>
<td>p(np)</td>
<td>[uc, un]</td>
<td>[A-STORE]</td>
<td></td>
</tr>
</tbody>
</table>

Resolve(p1) = [B-f, C-f] [U-LOAD]

Figure 4: Partial pointer analysis for analyzing a module M without and with side-effect summary(f) of a function f (Figure 6).
As shown in Figure 5, this auxil-

ty function, which is used in [A-CALL-D] and [U-CALL-I], resolves $fp(p1, \ldots, pn)$ to a target function $f(q1, \ldots, qn)$ by distinguishing two kinds of indirect calls (in LLVM IR), as discussed below. We use the clang++ front-end to annotate and identify a virtual callsite.

When generating low-level LLVM IR. In C++, the first parameter of a virtual function represents the hidden "this" pointer variable.

(I) C++ Virtual Calls. $f(q1, \ldots, qn)$ is a target function if $C1 \land (C2 \lor C3 \lor C4)$ holds, where $C1$ requires $f$ to be a valid candidate function according to the C++ specification for virtual calls \cite{C++Spec} and $C2 \lor C3 \lor C4$ requires $fp$ to satisfy some conditions related to the points-to information at this callsite. If $C2$ holds, then $fp$ points to $f$ explicitly. Consider $C3$, where $fp$ points to an unknown function, i.e., $u_1 \in p(fp)$. If $p_1$ points to a receiver object $a$ of type $t$, then $f$ is a target function, where $f \in vfnst(f)$ is a virtual function in $t$’s vtable, provided that $C$ also holds. $C4$ is similar to $C3$ except that $C4$ handles the case when $p_1$ points to an unknown receiver object of type $u_1$.

(II) C-Style Indirect Calls. These include both the calls made indirectly via C-style function pointers and C++-style member function pointers (which are rarely used and thus handled conservatively). Here, $f(q1, \ldots, qn)$ is a target function if $C5 \land (C2 \lor C6)$ holds, where $C5$ requires $fp$ and $f$ to be type-compatible according to the C99 standard (ISO/IEC 9899:1999). $C2$ is the same as above. If $C6$ is applicable, then $f \in F$ is any function that satisfies $C5$ under the C/C++ semantics.

Example 4 (PPA). By applying PPA to the LLVM IR in Figure 3(b), we give the points-to sets obtained and the corresponding inference rules applied in Figure 3(d). For the indirect call, i.e., $C++$ virtual call, at line 21 in Figure 3(b), $p$ is found to point to $u_0$ and $u_1$. By applying [U-CALL-I], where $C1 \land C4$ holds, the set of target functions is resolved to be $[B : f, C : f]$. Compared with CHA (Figure 3(c)), PPA has successfully removed its spurious target $A : f$.

3.2.3 Type-based Side-Effect Summaries (PPA\text{Summarize}) We make [U-GLOBAL], [U-CALL-D] and [U-CALL-I] in Figure 4 more precise, by refining their $ts(a)$ to $ts(a) \cap \underline{\text{summary}(f)}$ by using a side-effect type summary for every relevant function $f$. As a result, unknown objects with fewer types will be generated.

Figure 6 gives the three rules for computing $\text{summary}(f)$ for every function $f$ in all the modules in a program. We collect the types of objects that may be modified at a store in $f$ (S-STORE). We also recursively collect such types interprocedurally at a direct (S-CALL-D) and indirect call (S-CALL-I) in $f$.

Example 5 (PPA\text{Summary}). With type-based summaries, we have refined the points-to sets from Figure 3(d) to Figure 3(e). As $\text{summary(setData)} = \{ \text{int} \}$, no object of type $B*$ is ever modified at the direct call to setData at line 16 in Figure 3(b). Thus, when applying [U-CALL-D] to this call, a.q will no longer point to $u_0$, since a.f passed into setData via pointer x is not modified in setData. Therefore, $C : f$ is the only target resolved at line 21 (Figure 3(c)).
3.3 Pointer-Analysis-Guided Instrumentation

We present our new lightweight instrumentation technique to protect indirect calls in terms of the computed points-to information.

Prior Runtime Checks. We use an example in Figure 7 to illustrate why existing solutions [9, 17, 21, 33, 46, 50] are inefficient in implementing our fine-grained CFI protection (made possible by pointer analysis). Figure 7(a) gives a C++ code fragment with a virtual call p->f(), where p is of type A. The class hierarchy containing the set of all implementations of f, denoted $T_f$, is given in Figure 7(b).

Suppose $p^f_T = \{ C::f, F::f \}$ is the set of target functions found at p->f() by pointer analysis. To enforce CFI for p->f(), we can insert runtime checks for either the vtables accessed [9, 17, 46] (Figure 7(c)) or the virtual functions accessed [21, 33, 50] (Figure 7(d)), against $T_f^p$, a set of statically determined legitimate targets. However, such a set-membership test is impractically expensive for large programs, as $T_f^p$ can be large. For example, Chrome has large and deep class hierarchies consisting of thousands of classes. $T_f^p$ contains an average of 24.8 targets with the largest reaching 2013.

To accelerate membership testing, a recent CHA-based approach for enforcing CFI, called OvrIVT [9], reorders/interleaves the memory layout of the vtables in the same class hierarchy to make their addresses continuous. For the class hierarchy in Figure 7(b), &VTableA – &VTableE will be continuous in memory. Thus, the set membership test in Figure 7(c) can be simplified to a fast single range test:

```
assert(vtptr == &VTableA & &vtptr <= &VTableE);
```

Unfortunately, the set of legitimate targets allowed is $T_f = \{ A::f, C::f, D::f, E::f, F::f \}$ rather than $T_f^p = \{ C::f, F::f \}$. To avoid spurious targets such as A::f, D::f and E::f introduced by CHA but excluded by our approach, a single range check would have to be split into multiple range checks, which are costly. Finally, re-ordering/interleaving vtables is complex and error prone, especially in the presence of multiple and virtual inheritance.

Fast Index-based Runtime Checks. We introduce a simple yet efficient technique for protecting indirect calls based on the computed points-to information, without requiring vtable re-layout. For every indirect call protected, this technique requires only one single range check, and possibly, one more vector membership test.

Below we introduce our index-based technique by applying it to protect p->f() in Figure 7(a), as illustrated in Figure 8:

1. Assigning IDs. Let $C_f$ be the class hierarchy, which is always a DAG in C++, that contains all the possible implementations of f, denoted $T_f$, in the program. We assume that $C_f$ has a single source (as its root) and will consider more complex class hierarchies in Section 3.3.2. We assign a unique ID to every function in $T_f$. To facilitate a word-based optimization (OneWordOpt) discussed in Section 3.3.1, we will number the functions in $T_f$ continuously, starting from 0, by traversing $C_f$ in DFS. In our example, $T_f = \{ A::f, A::f, C::f, D::f, E::f, F::f, F::f \}$ (as assumed earlier). By traversing the class hierarchy shown in Figure 7(b) in DFS, we obtain the ID assignment given in Figure 8(a).

2. Assigning Prefix Data. For every function in $T_f$, its unique ID (represented by a 64-bit word) is inserted at the start of the function as its prefix data [6].

3. Constructing Valid Vectors. For p->f(), where p is of type A, we construct a valid vector, implemented as an array of chars, to tag the set of legitimate targets, $p^f_T$, found by pointer analysis. Let $id_{\text{min}}$ and $id_{\text{max}}$ be their smallest and largest IDs, respectively. We create a valid vector vvec[f][0 : id_{\text{max}} - id_{\text{min}}] and set vvec[f][i] to 1 if $T_f^p$ contains a function with ID, f.id, such that f.id - $id_{\text{min}} = i$, and 0 otherwise. In our example, $T_f^p = \{ C::f, F::f \}$. Thus, its valid vector is initialized as shown in Figure 8(b).

4. Inserting Runtime Checks. For every virtual call site p->f(), we insert the runtime checks, as shown in Figure 8(c), to allow only the targets in $T_f^p$ to be called. This involves one range check (line 5), and possibly, one more vector membership test (line 7).

In Figure 8(c), the two tests in lines 5 and 7 are implemented with 8 low-level LLVM instructions: 2 subtractions, 2 loads, 1 pointer arithmetic, and 3 comparisons. The checking overhead at a call site is independent of the callee protected and class hierarchies used.
Valid vector | Binary value | Decimal value
---|---|---
| 0...01001 | 9

(a) Word encoding for p->f()

1: vtptr = *sp;
2: vf = &vtptr[0];
3: fp = *vf;
4: fid = *vf-0x08; //load function id
5: assert(id < fid <= idmax); //range check
6: assert((9 >= (idmax-fid)) & 1 == 1) //validity check
7: call fp(p);

(b) Runtime checks for p->f()

Figure 9: OneWordOpt: replacing a valid vector by a 64-bit word for a callsite with 64 or fewer legitimate targets.

3.3.1 Optimizations. Two optimizations are described below.

MemTSTOpt. An index-based check involves three comparisons, as mentioned above. If an indirect callsite has no more than three legitimate targets, a traditional set membership test (as illustrated in Figure 7(d)) is adopted. For a single-target callsite, the target can be devirtualized, with its runtime checking avoided altogether. As pointer analysis becomes more and more precise, more and more virtual callsites will benefit from this optimization.

OneWordOpt. If a valid vector has 64 elements or fewer, we will use a 64-bit word instead of an array of chars so that we can replace a vector element test in line 7 of Figure 8(c) with a more efficient bitwise operation. Figure 9 demonstrates this optimization for checking the same callsite p->f() illustrated in Figure 8. In Figure 9(a), the binary pattern 0...01001 (corresponding to a decimal value of 9) that is stored in a valid vector in Figure 8(b) is now stored in a 64-bit word. The unused higher bits are assigned 0. In Figure 9(b), a bitwise check in line 6 is performed instead.

3.3.2 Handling Complex Class Hierarchies. Our instrumentation scheme works well for any complex class hierarchies, including multiple inheritance (Figure 10(a)) and virtual diamond inheritance (Figure 10(b)), which result in non-tree-like class hierarchies.

Figure 10: Complex class hierarchies in C++, with the IDs assigned for a function assumed to be defined in each class.

In C++, every class hierarchy must be a DAG. We can assume that every class hierarchy has a single source node, i.e., a root, by adding a dummy class to connect with the original source nodes. Let \( C_f \) be a single rooted class hierarchy containing all the definitions of a virtual function \( f \). We can assign unique IDs to these functions continuously by traversing \( C_f \) in DFS (i.e., preorder for trees).

Our index-based instrumentation scheme is correct, as revealed by the runtime checks in Figure 8, as long as our ID assignment is unique. However, assigning these IDs in DFS ensures that the definitions of \( f \) in the same tree-like sub-class hierarchy of \( C_f \) always receive continuous IDs, thereby maximizing the chances for OneWordOpt in Figure 9 to be applied. For complex class hierarchies in Figure 10, continuous IDs occur also for Figure 10(b) but not for Figure 10(a), assuming that every class contains a definition of \( f \). In practice, our ID assignment is effective. For Chrome, non-continuous IDs occur only in 371 out of 66609 indirect callsites.

4 EVALUATION

Our objective is to demonstrate that Vip can significantly improve the precision of virtual call integrity protection by using pointer analysis compared to the state-of-the-art CHA-based approach, thereby raising the bar against control-flow hijacking attacks.

We evaluate Vip using all SPEC2000/2006 C++ programs except 444.namd and 473.astar (because 444.namd has no virtual calls and 473.astar has one virtual call with one target resolved by both CHA and pointer analysis) and Google’s Chrome browser (with over 15 MLOC) based on the open-source project Chromium (version 54.0.2798.0). Their characteristics are listed in Table 2.

Vip protects virtual calls more effectively than CHA by significantly reducing the sets of legitimate target functions permitted at 20.3% of the virtual callsites per program, on average. Vip incurs an average (maximum) instrumentation overhead of 0.7% (3.3%), making it deployable as part of a compiler tool chain.

4.1 Implementation

Vip is implemented on top of SVF [41] in LLVM-3.8.0. During the pre-analysis, CHA is applied to the entire program to obtain the class hierarchy information. Our pointer analysis is field- and array-sensitive. Each field of a struct is treated as a separate object. Elements of an array are distinguished when accessed using constant indexes. Positive weight cycles (PWCs) that arise from processing fields and arrays are collapsed [35]. Distinct allocation sites (i.e., \text{AddressOf} instructions) are modeled by distinct objects [19, 25].

Our inclusion-based partial pointer analysis uses a wave propagation solver for constraint resolution, with PWCs detected by Nuutila’s algorithm [34]. To improve precision, library functions are summarized according to Figure 6. The call graph of a program module is built on the fly by \text{[A-CALL-I]} and \text{[U-CALL-I]} (Figure 4). Points-to sets are represented using LLVM’s \text{SparseBitVectors}.

For our fast index-based instrumentation, we first assign unique IDs to the virtual functions with the same function signature based on the pre-computed class hierarchy. We then insert the runtime checks at every virtual callsite according to the set of legitimate targets found by pointer analysis. All valid vectors are placed in the read-only section to protect them from being modified by attackers.

4.2 Experiment Setup and C++ Applications

Experiment Setup. All our experiments were conducted on a platform consisting of a 3.70GHz Xeon(R) E5-1620 v2 CPU with

<table>
<thead>
<tr>
<th>Program</th>
<th>KLOC</th>
<th>#Stmt</th>
<th>#Ptrs</th>
<th>#Objs</th>
<th>#CallSite</th>
</tr>
</thead>
<tbody>
<tr>
<td>dealll</td>
<td>199</td>
<td>577482</td>
<td>530249</td>
<td>77894</td>
<td>94284</td>
</tr>
<tr>
<td>eon</td>
<td>41</td>
<td>65218</td>
<td>63385</td>
<td>15855</td>
<td>14033</td>
</tr>
<tr>
<td>omnetpp</td>
<td>48</td>
<td>95961</td>
<td>108349</td>
<td>8592</td>
<td>20601</td>
</tr>
<tr>
<td>povray</td>
<td>155</td>
<td>126484</td>
<td>189138</td>
<td>8195</td>
<td>15226</td>
</tr>
<tr>
<td>soplex</td>
<td>41</td>
<td>54190</td>
<td>69287</td>
<td>4191</td>
<td>9878</td>
</tr>
<tr>
<td>xalan</td>
<td>553</td>
<td>744971</td>
<td>703675</td>
<td>73973</td>
<td>106090</td>
</tr>
<tr>
<td>Chrome</td>
<td>15224</td>
<td>20362892</td>
<td>34173201</td>
<td>1685228</td>
<td>3566682</td>
</tr>
<tr>
<td>Total</td>
<td>16261</td>
<td>22027198</td>
<td>35837264</td>
<td>1873928</td>
<td>3826794</td>
</tr>
</tbody>
</table>

Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue

Table 2: Program characteristics.
128 GB memory, running Ubuntu Linux 14.04. The individual C++ files of a program are compiled under -O2 into bitcode files by the clang++ front-end and then the files within the same program module are merged together using the LLVM Gold Plugin at link time (-lto). CHA, as a pre-analysis of Vip, collects the class hierarchy information from all modules. Then we perform pointer analysis and instrumenta‌tion on each module. Finally, clang++ is used to generate the final executable using the instrumented bitcode files.

**C++ Applications.** We use Chrome including the whole browser and its supporting libraries in the Chromium code base. Chrome contains one executable (5.9MLOC) and 131 shared libraries (9.3MLOC), totaling 15.2 MLOC with 132 modules compiled by clang++. Its largest class hierarchy has 9519 classes. Based on CHA, some virtual callsites can invoke over 2000 target functions. The executable and shared libraries interact closely with 22952 functions. As shown in Table 3, Column 2 gives the number of callsites where CHA, as a pre-analysis of Vip, collects the class hierarchy information from all modules. Then we perform pointer analysis and instrumentation on each module. Finally, clang++ is used to generate the final executable using the instrumented bitcode files.

**Table 3: Instrumentation statistics of Vip.**

<table>
<thead>
<tr>
<th>Program</th>
<th>#Callsites</th>
<th>#Callsites with different checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>dealll</td>
<td>902</td>
<td>1 2 3 4 64</td>
</tr>
<tr>
<td>eon</td>
<td>91</td>
<td>1 2 3 4 64</td>
</tr>
<tr>
<td>omnetpp</td>
<td>686</td>
<td>1 2 3 4 64</td>
</tr>
<tr>
<td>povray</td>
<td>122</td>
<td>1 2 3 4 64</td>
</tr>
<tr>
<td>soplex</td>
<td>553</td>
<td>1 2 3 4 64</td>
</tr>
<tr>
<td>xalan</td>
<td>9342</td>
<td>1 2 3 4 64</td>
</tr>
<tr>
<td>Chrome</td>
<td>66609</td>
<td>1 2 3 4 64</td>
</tr>
<tr>
<td>Total</td>
<td>78305</td>
<td>1 2 3 4 64</td>
</tr>
</tbody>
</table>

**Table 4: Target functions resolved by Vip and CHA.**

<table>
<thead>
<tr>
<th>Program</th>
<th>#Callsite</th>
<th>Analysis time</th>
</tr>
</thead>
<tbody>
<tr>
<td>dealll</td>
<td>368</td>
<td>1.3secs 77.9secs</td>
</tr>
<tr>
<td>eon</td>
<td>43</td>
<td>0.1secs 3.9secs</td>
</tr>
<tr>
<td>omnetpp</td>
<td>657</td>
<td>0.1secs 21.1secs</td>
</tr>
<tr>
<td>povray</td>
<td>112</td>
<td>0.1secs 53.7secs</td>
</tr>
<tr>
<td>soplex</td>
<td>549</td>
<td>0.1secs 4.3secs</td>
</tr>
<tr>
<td>xalan</td>
<td>7445</td>
<td>2.2secs 4780.0secs</td>
</tr>
<tr>
<td>Chrome</td>
<td>53994</td>
<td>2188.0secs 6.1hrs</td>
</tr>
<tr>
<td>Total</td>
<td>63368</td>
<td>2192.0secs 7.4hrs</td>
</tr>
</tbody>
</table>

where Vip has the same precision as CHA, while Column 3 gives the number of callsites where Vip is more precise. Columns 4 – 6 show the number of these callsites where Vip can further reduce the number of targets to one, two and three, respectively.

Figure 11 gives the details about the callsites where Vip resolves fewer targets than CHA. For each program, the x-axis represents its callsites and the y-axis represents the number of targets at a callsite resolved by Vip and CHA. To save space, soplex is omitted as its four more precise callsites are all reduced to one target (Table 4). Vip resolves fewer targets than CHA at 14937 out of all 78305 callsites in all programs by removing 9.8% of the (spurious) targets resolved by CHA. On average, Vip is more precise than CHA at 20.3% of the callsites per program. Over 50% of the callsites in dealll, omnetpp, povray, soplex, xalan and Chrome have fewer than four targets. This clearly shows that our pointer analysis can significantly increase the precision of CFI and provide lots of opportunities for the two optimizations in Section 3.3.1 to boost the performance of Vip.

**Static Instrumentation.** As shown in Table 3, Column 2 gives the number of callsites for each program. Columns 3 – 5 give the number of callsites where MemTSTOpt is applied (Section 3.3.1). In particular, the single-target callsites (with over 30% in all programs and over 50% in four of the seven programs) require no runtime checks at all. Column 6 gives the number of callsites where OneWordOpt (Section 3.3.1) is applied. Column 7 shows the number of callsites with unoptimized instrumentation, requiring the most time-consuming runtime checks. However, those callsites only occupy a very small percentage of the total callsites (less than 2% for every program). These results demonstrate the significant improvements made by Vip in reducing spurious target functions.

**4.3 Precision and Performance**

This section evaluates the precision and performance of Vip using large-scale applications, including SPEC programs and Chrome.

**Precision of pointer analysis.** Table 4 compares the precision of Vip against CHA in terms of the number of target functions resolved at all virtual callsites. Column 2 gives the number of callsites and resolves fewer targets than CHA. For each program, the x-axis represents its callsites and the y-axis represents the number of targets at a callsite resolved by Vip and CHA. To save space, soplex is omitted as its four more precise callsites are all reduced to one target (Table 4). Vip resolves fewer targets than CHA at 14937 out of all 78305 callsites in all programs by removing 9.8% of the (spurious) targets resolved by CHA. On average, Vip is more precise than CHA at 20.3% of the callsites per program. Over 50% of the callsites in dealll, omnetpp, povray, soplex, xalan and Chrome have fewer than four targets. This clearly shows that our pointer analysis can significantly increase the precision of CFI and provide lots of opportunities for the two optimizations in Section 3.3.1 to boost the performance of Vip.
The overheads for Table 4 for CHA and Vip (evaluated under checks over the total. For are slight increases in terms of LLVM IR instructions used, ranging A::f an illegitimate function p->f() after-free or bufer overlow vulnerability. Then the vtable pointer in an object pointed to by p will invoke an illegitimate function A::f that contains a sensitive method e.g., system(). This attack is prevented by Vip but not by CHA.

5 RELATED WORK

Virtual Call Integrity Protection. Vtable hijacking attacks can be mitigated in source- or binary-level programs. Binary-level defenses [15, 36, 51] can be deployed to production code where source code is not available. However, as compiler optimizations cannot be easily applied to binaries, binary-level instrumentation is often more expensive than source-level instrumentation. In addition, due to the lack of high-level C++ semantics, binary-level defenses cannot protect C++ virtual calls precisely, making them vulnerable to COOP-style attacks [37].

Closest to our work are source-level vtable protection and forward-edge CFI techniques via compiler transformations. Source-level defenses give stronger security guarantees than binary-level ones, making vtable hijacking harder. Vtv [46] and SAFEDispatch [21] are two early CHA-based techniques using membership tests. Vtv [46] incurs an overhead of 4.1% on SPEC programs. SAFEDispatch requires profile-guided optimizations to achieve an overhead of 2.1%. Later, Ovt/Ivt [9] simplifies membership tests into range checks by reordering/interleaving vtables to make their addresses continuous. However, this requires significant modifications to the memory layout of vtables, which can be complex and error prone, especially in the presence of multiple and virtual inheritance [9]. VTrust [50], which is a recent CHA-based approach, checks a virtual function using a signature comparison, by trading precision for efficiency.

Vip enforces a much more precise call graph than the CHA-based approach by using pointer analysis and a fast index-based instrumentation technique. Thus, the bar is raised against vtable hijacking by providing stronger security guarantees.

Pointer Analysis. Most of the existing pointer analyses [19, 25, 26, 38, 39, 42–44, 48, 49] for C only model a subset of the modern intermediate languages (e.g., LLVM IR). A few approaches [8, 23] provide comprehensive support for analyzing both C and its OO incarnation, C++. Furthermore, existing whole-program pointer analyses [8, 19, 25, 26, 48] assume that the entire program is compiled into a single program module (e.g., linked into one LLVM bc). However many real-world programs are often composed of separately compiled modules, such as Chrome. Applying previous pointer analyses to resolve virtual calls in a single program module with incomplete code leads to unsound results, which makes CFI unusable in practice due to false protection. This paper introduces a new partial pointer analysis to support both C and C++. We enforce the pointer analysis results for virtual call integrity protection as a major client, showing the benefits of our partial pointer analysis.

6 CONCLUSION

This paper introduces Vip, a new approach to boosting the precision of CFI protection for large-scale multi-MLOC real-world C++ programs. Vip introduces a new partial pointer analysis and a new lightweight instrumentation technique to enforce CFI at virtual call sites, thereby raising the bar against vtable hijacking attacks.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and Mr. Mohamad Barbar for providing a proof-of-concept attack in the artifact. The work is supported by ARC Grants (DP150102109 and DE170101081) and a CSC scholarship.
REFERENCES


