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

Xiaokang Fan
UNSW Sydney, Australia

Yulei Sui
UNSW Sydney, Australia

Xiangke Liao
National University of Defense Technology, China

Jingling Xue
UNSW Sydney, Australia

ABSTRACT

We present, VIP, an approach to boosting the precision of **V**irtual call **I**ntegrity **P**rotection 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 vtable 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 tool chain.

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:
Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the Precision of Virtual Call Integrity Protection with Partial Pointer Analysis for C++ . In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 12 pages.
https://doi.org/10.1145/3092703.3092729

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 $2\times - 4\times$ performance slowdown with respect to the native runs [22].

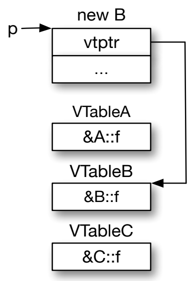
Recent solutions protect virtual call integrity by using a lightweight Control-Flow Integrity (CFI) [6] approach, at either the binary code [15, 36, 47, 51] or source code [21, 33, 46, 50]. In general, source-level CFI provides stronger security guarantees by considering high level C++ semantics, making 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.

ISSTA'17, July 2017, Santa Barbara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092729

```
class A {
  public: virtual void f() {...}
};
class B: public A {
  public: virtual void f() {...}
};
class C {
  public: virtual void f() {...}
};
...
int main() {
  A *p = new B;
  p->f();
}
```

(a) Source code



(b) Object layout

```
vtptr = *p;      //load vtable
vfn = &vtptr[0]; //the 1st vtable entry address
fp = *vfn;       //load virtual function
assert(fp==&A::f || fp==&B::f);
fp(p);           //call virtual function
```

(c) Pseudo LLVM instructions for virtual call `p->f()`

Figure 1: Imprecision of the CHA-based approach, which excludes `C::f`, but includes a spurious function `A::f`.

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., Google's 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 statically, 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, 48, 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 Vɪᴘ, an approach to boosting the precision of **V**irtual call **I**ntegrity **P**rotection 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. Vɪᴘ achieves a comparable low overhead with the state-of-the-art CHA-based solutions (e.g., Vᴛᴠ [46], Clang Vᴄꜰɪ [2] and OVT/Iᴠᴛ [9]), but offers better precision, thus raising the bar against vtable hijacking attacks.

One key technique of Vɪᴘ 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, Vɪᴘ 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, Vɪᴘ 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, Oᴠᴛ/Iᴠᴛ [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 Vɪᴘ 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 Vɪᴘ 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 Vɪᴘ, a new **V**irtual call **I**ntegrity **P**rotection 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 callsites.
- We have implemented Vɪᴘ fully in LLVM-3.8.0 and evaluated it using SPEC programs and Chrome. Statically, Vɪᴘ 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, Vɪᴘ 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 counterfeit 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 callsite 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 $\mathcal{V}$ in the program are separated into two subsets: $\mathcal{O}$ that contains all possible targets, i.e., *abstract objects* or *address-taken variables* of a pointer, and $\mathcal{P}$ that contains all *top-level pointers.*

Table 1: Domains and LLVM IR used by pointer analysis.

Analysis Domains		
f	$\in \mathcal{F}$	Program functions
c, fld	$\in \mathcal{C}$	Constants
t	$\in \mathcal{T}$	Types
p, q, ret	$\in \mathcal{S}$	Stack virtual registers
g	$\in \mathcal{G}$	Global pointer variables
p, q, ret, g	$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	Top-level Pointers
a, a_f, a.fld, a[c]	$\in \mathcal{A}$	Allocation-based normal objects
u_t	$\in \mathcal{U}$	Type-based unknown objects
o	$\in \mathcal{O} = \mathcal{A} \cup \mathcal{U}$	Abstract objects
v	$\in \mathcal{V} = \mathcal{P} \cup \mathcal{O}$	program variables

LLVM IR			
Prog ::=	$\overline{\mathtt{M}}$		PROGRAM
M ::=	$\overline{\mathtt{m}}$		MODULE
m ::=	g | f(p_1,...,p_n){ $\overline{\mathtt{inst}}$; }		GLOBAL|FUNCTION
inst ::=	p = &a		ADDRESSOF
	p = *q		LOAD
	*p = q		STORE
	p = (t) q		CAST
	p_3 = phi(p_1,p_2)		PHI
	p = &(q $\rightarrow$ fld)		FIELD
	p = &q[c]	(constant index)	ARRAY-C
	p = &q[i]	(variable index)	ARRAY-V
	ret = f(p_1,...,p_n)	(direct call)	CALL-D
	ret = fp(p_1,...,p_n)	(indirect call)	CALL-I
	$\mathtt{return}_f$ q		RETURN

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

We distinguish two kinds of abstract objects, $\mathcal{O} = \mathcal{A} \cup \mathcal{U}$. For a complete program, every abstract object a $\in \mathcal{A}$ 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[*] 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 $u_t \in \mathcal{U}$ 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 definitions. 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_1 points to its receiver object. Finally, $\mathtt{return}_f$ q denotes a return instruction in function f.

```
 1: class A {
 2:
 3:   A() { }
 4:   virtual void f() {
 5:     ...
 6:   }
 7: };
 8:
 9: void main() {
10:   A *p = new A;
11:
12:
13:   p->f();
14:
15:
16:
17: }
```
(a) C++ code

```
 1: VTableA = {&aA::f, ...}
 2:
 3: void A::A(A *this) {
 4:   tmp = &VTableA;
 5:   *this = tmp;
 6: }
 7: void A::f(A *this) {...}
 8:
 9: void main() {
10:   p = &a;
11:   A::A(p);
12:
13:   vtptr = *p;
14:   vfn = &vtptr[0];
15:   fp = *vfn;
16:   fp(p);
17: }
```
(b) LLVM IR

Figure 2: C++ code and its corresponding LLVM IR.

Example 1 (LLVM IR). *Figure 2(a) gives a C++ code fragment, including a class definition (lines 1-7), an object creation statement (line 10) and a virtual call (line 13). Figure 2(b) shows its LLVM IR. For class* `A`*, LLVM generates its vtable,* `VTableA` *(an array of constants), to store the function object* $\mathtt{a}_{\mathtt{A::f}}$ *(line 1 in Figure 2(b)). The address of* `VTableA` *is stored into an object of class* `A` *whenever the object is created and initialized via a constructor of* `A` *(lines 3-6 in Figure 2(b)).*

An object creation statement `A * p = new A` *is translated into two instructions, an* AddressOf *that allocates an object* `a` *and a function call* Call-D *that passes* `p` *into* `A`*'s constructor to initialize* `a` *by storing in it a pointer to* `A`*'s vtable. A virtual call* `p->f()` *is translated into four instructions: (1) a* Load *(reading the vtable pointer), (2) an* Array-C *(getting the address of the entry in vtable corresponding to the virtual function to be called), (3) a* Load *(reading the virtual function pointer), and (4) an* Call-I *(making an indirect function call).*

3.2 Partial Pointer Analysis for C++

Section 3.2.1 describes briefly the unsoundness of conventional pointer analysis (CPA) in analyzing incomplete code. Section 3.2.2 introduces our partial pointer analysis (PPA) for a single module with incomplete code. Finally, Section 3.2.3 discusses how to enhance the precision of PPA by using a type-based side-effect summary ($\text{PPA}_{\mathtt{Summary}}$). Figure 3 gives an example that compares all the relevant analyses in analyzing a single module `main.bc`.

3.2.1 CPA's Unsoundness for Incomplete Code. CPA's unsoundness arises when analyzing three incomplete code patterns in a module `M`. (1) A call instruction $\mathtt{ret} = \mathtt{f}(\mathtt{p}_1, \ldots, \mathtt{p}_n)$ in `M` may call a function `f` defined in another module. Any object (together with its fields) and array elements passed indirectly through arguments $\mathtt{p}_1, \cdots, \mathtt{p}_n$ may be modified outside `M`, and `ret` may also point to objects not defined in `M`. (2) A function $\mathtt{f}(\mathtt{p}_1, \ldots, \mathtt{p}_n)$ defined in `M` may be called by a function in another module. `f`'s formal parameters $\mathtt{p}_1, \cdots, \mathtt{p}_n$ may point to incomplete points-to targets. (3) A global variable is used in `M` but defined in another module.

CPA is unsound due to missing pointed-to targets in some pointers, resulting in missing target functions at some callsites in `M`. Thus, CPA is not suited for enforcing CFI, since such false protection will cause the program to crash at runtime checks.

Example 2 (CPA). *Figures 3(a) and (b) list the source files of a program and their LLVM IR. Module* `lib.bc` *is used as a dynamic library when linked with module* `main.bc`*. When applying CPA to analyze* `main.bc` *in Figure 3(b), where the callsite at line 11 invokes* `getC` *defined in* `lib.bc`*, the points-to set of* `c` *is empty as the body of* `getC` *is unavailable. As* `setData` *is also not available for analysis, both* `x->q` *and* `p` *also have an empty points-to set each. Thus, no function is found to be invokable at* `p->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 `escMs(g)` (`escMs(f)`), the set of modules that uses (calls) a global pointer variable `g` (a function `f`).

Main Analysis. PPA analyzes a single module by introducing type-based unknown objects $\mathtt{u}_t \in \mathcal{U}$ to handle the three cases discussed in Section 3.2.1. Each $\mathtt{u}_t$ represents a set of unknown objects whose types are t or subtypes of t. Our analysis aims to infer t by leveraging the existing type information of the pointers in $\mathcal{P}$ and allocation-based objects in $\mathcal{A}$ in the analyzed module in order to filter out spurious target functions that would otherwise be introduced at a virtual callsite by CHA. To improve precision further, $\text{PPA}_{\mathtt{Summary}}$ enhances PPA by applying a lightweight type-based side effect summary of functions across all the modules.

Before delving into the inference rules in Figure 4, we explain the basic idea behind by revisiting the code in Figure 3(a).

Example 3 (PPA). *Figure 3(c) compares the sets of target functions resolved at* `p->f()` *by CHA, CPA, PPA and* $\text{PPA}_{\mathtt{Summary}}$*. CHA assumes soundly that any virtual function* `f()` *in* `A` *or its subclasses may be called. CPA finds unsoundly no targets, as explained in Example 2.*

PPA computes soundly the points-to information in Figure 3(d) by applying our inference rules to Figure 3(b). When analyzing line 11, PPA adds an unknown object $\mathtt{u}_\mathtt{C}$ *to the points-to set of* `c` *to represent an object of type* `C` *returned from* `getC` *(as* `C` *has no subtypes). This object will flow to pointer* `p` *at* `p->f()` *via a store at line 15 and a load at line 17. For an object* `a` *(created as an instance of* `X` *at line 12) passed indirectly via* `x` *to* `setData` *(line 16), PPA adds another unknown object* $\mathtt{u}_\mathtt{B}$ *to the points-to set of* `a.q`*. Here,* $\mathtt{u}_\mathtt{B}$ *indicates that* `a.q` *may be modified by this call to point to an object of type* `B` *or one of its subtypes, deduced from the declaration type* `B*` *of field* `q` *(line 4 of Figure 3(a)). Finally, we find that* `p` *points to objects of type* `B` *and* `C`*. Thus, the spurious target* `A:f` *introduced by CHA is removed.*

$\text{PPA}_{\mathtt{Summary}}$ *further improves the precision of PPA by building a lightweight type summary for lib.bc to indicate that* `setData` *does not modify any object of type* `B*`*. Thus, there is no longer a need to add* $\mathtt{u}_\mathtt{B}$ *to the points-to set of* `a.q`*. 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 PPA and $\text{PPA}_{\mathtt{Summary}}$. In LLVM SSA form, a pointer $\mathtt{p} \in \mathcal{P}$ is defined uniquely to have one single static type at its declaration while an object $\mathtt{a} \in \mathcal{A}$ that is created at an allocation site may have multiple (dynamic) types due to Cast operations applied to the object. For a pointer $\mathtt{p} \in \mathcal{P}$, we use $T(\mathtt{p})$ to denote its pointer type and $\widetilde{T(p)}$ to denote its pointee type. For example, given a pointer declaration `B*` `p`, we have $T(\mathtt{p}) = \mathtt{B*}$ and $\widetilde{T(p)} = \mathtt{B}$. For an allocation-based object $\mathtt{a} \in \mathcal{A}$, $ts(\mathtt{a})$ denotes the set of (dynamic) types that `a` may have.

Below we examine our inference rules for a module `M` in Figure 4. We write $pt(v)$ to represent the points-to set of a variable $v \in \mathcal{V}$. The first 13 rules (*A-* inference rules*) aim to resolve $ts(\mathtt{a})$ for an

lib.h
```
1: class A {public: virtual void f();};
2: class B: public A {public: virtual void f();};
3: class C: public B {public: virtual void f();};
4: class X {public: B *q; int data;};
5: void setData(X *x, int d);
6: C *getC();
```

lib.cpp
```
7: #include "lib.h"
8: void setData(X *x, int d) {
9:     x->data = d;
10: }
11: C *getC() {return new C;}
```

main.cpp
```
12: #include "lib.h"
13: void main() {
14:     C *c = getC();
15:     X *x = new X;      //abstract object a
16:     x->q = c;
17:     setData(x, 1);
18:     A *p = x->q;
19:     p->f();
20: }
```

(a) C++ source files

lib.bc
```
1: void setData(X *x, int d) {
2:     tmp1 = &(x->data);
3:     *tmp1 = d;
4: }
5: C *getC() {
6:     tmp2 = &c';
7:     C::C(tmp2);
8:     return tmp2;
9: }
```

main.bc
```
10: void main( ) {
11:     c = getC();
12:     x = &a;
13:     X::X(x);
14:     tmp = &(x->q);
15:     *tmp = c;
16:     setData(x, 1);
17:     p = *tmp;
18:     vtptr = *p;
19:     vfn = &vtptr[0];
20:     fp = *vfn;
21:     fp(p);
22: }
```

(b) Two modules in the form of LLVM IR (with four constructors' IR omitted for brevity)

Analysis	Target Functions	Soundness
CHA	**{A::f, B::f, C::f}**	✓
CPA	∅	✗
PPA	**{B::f, C::f}**	✓
PPA$_{\text{Summary}}$	**{C::f}**	✓

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

Pointer	Points-to Set	Statement	Rule
pt(c)	$=\{u_C\}$	11	[U-CALL-D]
pt(x)	$=\{a\}$	12	[A-ALLOC]
pt(tmp)	$=\{a.q\}$	14	[A-FIELD]
pt(a.q)	$=\{u_C, u_B\}$	15 16	[A-STORE] [U-CALL-D]
pt(p)	$=\{u_C, u_B\}$	17	[A-LOAD]
pt(vtptr)	$=\{u_{\text{void (A*)*}}\}$	18	[U-LOAD]
pt(vfn)	$=\{u_{\text{void (A*)*}}\}$	19	[U-ARRAY]
pt(fp)	$=\{u_{\text{void (A*)}}\}$	20	[U-LOAD]
Resolve(fp(p)) = {B::f, C::f}			[U-CALL-I]

(d) Points-to sets inferred by PPA

Pointer	Points-to Set	Statement	Rule
...	(Same as in Figure 3(d))		
pt(a.q)	$=\{u_C\}$	15 16	[A-STORE] [U-CALL-D]
pt(p)	$=\{u_C\}$	17	[A-LOAD]
...	(Same as in Figure 3(d))		
Resolve(fp(p)) = {C::f}			[U-CALL-I]

(e) Points-to sets inferred by PPA$_{\text{Summary}}$

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

$$\text{[A-GLOBAL]}\ \frac{g = \&a \quad g \in \mathcal{G} \quad t = T(g)}{\{a\} \subseteq pt(g) \quad \{\widetilde{t}\} \subseteq ts(a)} \qquad \text{[A-HEAP]}\ \frac{p = \&a \quad a \text{ is a heap obj}}{\{a\} \subseteq pt(p) \quad ts(a) = \emptyset} \qquad \text{[A-STACK]}\ \frac{p = \&a \quad a \text{ is a stack obj} \quad t = T(p)}{\{a\} \subseteq pt(p) \quad \{\widetilde{t}\} \subseteq ts(a)}$$

$$\text{[A-PHI]}\ \frac{p_3 = \text{phi}(p_1,\ p_2)}{pt(p_1) \subseteq pt(p_3) \quad pt(p_2) \subseteq pt(p_3)} \qquad \text{[A-CAST]}\ \frac{p = (t)\ q \quad a \in pt(q) \quad a \in \mathcal{A}}{pt(q) \subseteq pt(p) \quad \{\widetilde{t}\} \subseteq ts(a)} \qquad \text{[A-FUNCTION]}\ \frac{p = \&a_f \quad f(p_1, \dots, p_n) \quad t = T(p)}{\{a_f\} \subseteq pt(p) \quad \{\widetilde{t}\} \subseteq ts(a_f)}$$

$$\text{[A-LOAD]}\ \frac{p = *q \quad a \in pt(q) \quad a \in \mathcal{A}}{pt(a) \subseteq pt(p)} \qquad \text{[A-STORE]}\ \frac{*p = q \quad a \in pt(p) \quad a \in \mathcal{A}}{pt(q) \subseteq pt(a)} \qquad \text{[A-FIELD]}\ \frac{p = \&(q \rightarrow \text{fld}) \quad a \in pt(q) \quad a \in \mathcal{A} \quad t = T(p)}{\{a.\text{fld}\} \subseteq pt(p) \quad \{\widetilde{t}\} \subseteq ts(a.\text{fld})}$$

$$\text{[A-ARRAY-C]}\ \frac{p = \&q[c] \quad a \in pt(q) \quad a \in \mathcal{A} \quad t = T(p)}{\{a[c]\} \subseteq pt(p) \quad \{\widetilde{t}\} \subseteq ts(a[c])} \qquad \text{[A-ARRAY-V]}\ \frac{p = \&q[i] \quad a \in pt(q) \quad a \in \mathcal{A} \quad t = T(p)}{\{a[*]\} \subseteq pt(p) \quad \{\widetilde{t}\} \subseteq ts(a[*])}$$

$$\text{[A-CALL-D]}\ \frac{\begin{array}{c} \text{ret} = f(p_1, \dots, p_n) \in M \\ f(q_1, \dots, q_n) \in M' \quad M = M' \quad \text{return}_f\ q \end{array}}{\forall i \in \{1, \dots, n\} : pt(p_i) \subseteq pt(q_i) \quad pt(q) \subseteq pt(\text{ret})} \qquad \text{[A-CALL-I]}\ \frac{\begin{array}{c} \text{ret} = fp(p_1, \dots, p_n) \in M \ \ f \in \textbf{Resolve}(fp(p_1, \dots, p_n)) \\ f(q_1, \dots, q_n) \in M' \quad M = M' \quad \text{return}_f\ q \end{array}}{\forall i \in \{1, \dots, n\} : pt(p_i) \subseteq pt(q_i) \quad pt(q) \subseteq pt(\text{ret})}$$

$$\text{[U-FUNCTION]}\ \frac{f(p_1, \dots, p_n) \in M \quad M' \in \text{escMs}(f) \quad M \neq M'}{\forall i \in \{1, \dots, n\} \quad t = T(p_i) : \{u_{\widetilde{t}}\} \subseteq pt(p_i)} \qquad \text{[U-FIELD]}\ \frac{\begin{array}{c} p = \&(q \rightarrow \text{fld}) \quad t = T(p) \\ u_- \in pt(q) \quad u_- \in \mathcal{U} \end{array}}{\{u'_{\widetilde{t}}\} \subseteq pt(p)} \qquad \text{[U-ARRAY]}\ \frac{\begin{array}{c} p = \&q[\_] \quad t = T(p) \\ u_- \in pt(q) \quad u_- \in \mathcal{U} \end{array}}{\{u'_{\widetilde{t}}\} \subseteq pt(p)} \qquad \text{[U-PTG]}\ \frac{\begin{array}{c} a \in pt(p) \\ a \in \mathcal{A} \end{array}}{\{a\} \subseteq ptg(p)}$$

$$\text{[U-LOAD]}\ \frac{\begin{array}{c} p = *q \quad t = T(p) \\ u_- \in pt(q) \quad u_- \in \mathcal{U} \end{array}}{\{u'_{\widetilde{t}}\} \subseteq pt(p)} \qquad \text{[U-CAST-D]}\ \frac{\begin{array}{c} p = (t')\ q \quad u_t \in pt(q) \\ u_t \in \mathcal{U} \quad \widetilde{t'} <: t \end{array}}{\{u_{\widetilde{t'}}\} \subseteq pt(p)} \qquad \text{[U-CAST-U]}\ \frac{\begin{array}{c} p = (t')\ q \quad u_t \in pt(q) \\ u_t \in \mathcal{U} \quad t <: \widetilde{t'} \end{array}}{\{u_t\} \subseteq pt(p)} \qquad \text{[U-PTG-REC]}\ \frac{a \in ptg(p) \quad a \in \mathcal{A}}{pt(a) \cup \{a.\text{fld}, a[*]\} \subseteq ptg(p)}$$

$$\text{[U-GLOBAL]}\ \frac{\begin{array}{c} g \in M \ \ g \in \mathcal{G} \ \ M' \in \text{escMs}(g) \ \ M \neq M' \\ a \in ptg(g) \quad f \in \mathcal{F} \quad t \in ts(a) \boxed{\cap\, summary(f)} \end{array}}{\{u_{\widetilde{t}}\} \subseteq pt(a)} \qquad \text{[U-CALL-D]}\ \frac{\begin{array}{c} \text{ret} = f(p_1, \dots, p_n) \in M \ \ f \in M' \ \ M \neq M' \\ t = T(\text{ret}) \ \ i \in \{1, \dots, n\} \ \ a \in ptg(p_i) \ \ t' \in ts(a) \boxed{\cap\, summary(f)} \end{array}}{\{u_{\widetilde{t}}\} \subseteq pt(\text{ret}) \quad \{u'_{\widetilde{t'}}\} \subseteq pt(a)}$$

$$\text{[U-CALL-I]}\ \frac{\text{ret} = fp(p_1, \dots, p_n) \in M \ \ f \in \textbf{Resolve}(fp(p_1, \dots, p_n)) \ \ f \in M' \ \ M \neq M' \ \ t = T(\text{ret}) \ \ i \in \{1, \dots, n\} \ \ a \in ptg(p_i) \ \ t' \in ts(a) \boxed{\cap\, summary(f)}}{\{u_{\widetilde{t}}\} \subseteq pt(\text{ret}) \quad \{u'_{\widetilde{t'}}\} \subseteq pt(a)}$$

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

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

(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 $\widetilde{t}$ 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 $o \in \mathcal{A} \cup \mathcal{U}$ between pointers. [A-LOAD] and [A-STORE] handle loads and stores for address-taken objects in $\mathcal{A}$. [A-FIELD], [A-ARRAY-C] and [A-ARRAY-V] realize field- and array-sensitivity. For $p = \&(q \rightarrow \text{fld})$ or $p = \&q[\_]$, where q points to a base object $a \in \mathcal{A}$, p is made to point to the subobject a.fld or a[_]. In LLVM IR, a high-level statement p->f=q in

$$\textbf{Resolve}(\mathtt{fp}(\mathtt{p}_1,\dots,\mathtt{p}_n)) = \begin{cases} \{\mathtt{f}(\mathtt{q}_1,\dots,\mathtt{q}_n) \in \mathcal{VF} \mid \textbf{C1} \wedge (\textbf{C2} \vee \textbf{C3} \vee \textbf{C4})\} & \text{Virtual Call} \\ \{\mathtt{f}(\mathtt{q}_1,\dots,\mathtt{q}_n) \in \mathcal{F} \mid \textbf{C5} \wedge (\textbf{C2} \vee \textbf{C6})\} & \text{Otherwise} \end{cases}$$

C1: $\widetilde{T(\mathtt{q}_1)} <: \widetilde{T(\mathtt{p}_1)} \wedge \forall \mathtt{i} \in \{2,\dots,\mathtt{n}\} : T(\mathtt{p_i}) = T(\mathtt{q_i})$

C2: $\mathtt{a_f} \in pt(\mathtt{fp})$

C3: $\mathtt{u}_{\_} \in pt(\mathtt{fp}) \wedge \mathtt{a} \in pt(\mathtt{p}_1) \wedge t \in ts(\mathtt{a}) \wedge \mathtt{f}(\mathtt{q}_1,\dots,\mathtt{q}_n) \in vfns(t)$

C4: $\mathtt{u}_{\_} \in pt(\mathtt{fp}) \wedge \mathtt{u}_t \in pt(\mathtt{p}_1) \wedge t' <: t \wedge \mathtt{f}(\mathtt{q}_1,\dots,\mathtt{q}_n) \in vfns(t')$

C5: $\forall \mathtt{i} \in \{1,\dots,\mathtt{n}\} : T(\mathtt{p_i}) \cong T(\mathtt{q_i})$ **[Type Compatibility $\cong$ by C99]**

C6: $\mathtt{u}_{\_} \in pt(\mathtt{fp}) \wedge \mathtt{f} \in \mathcal{F}$

Figure 5: Resolving indirect calls, where $\mathcal{VF} \subseteq \mathcal{F}$ (Table 1) contains all virtual functions. For a type t, *vfns(t)* denotes the set of virtual functions in its vtable. Recall that for a pointer $r \in \mathcal{P}$, $T(r)$ ($\widetilde{T(r)}$) denotes its pointer (pointee) type and <: denotes the standard subtyping relation.

C++ is decomposed into `tmp = &(p->f)` and `*tmp = q`. Similarly, `p[_]=q` in C++ is decomposed into `tmp = &p[_]` and `*tmp = q`.

Passing arguments into and receiving values from a callee invoked directly (indirectly) at a callsite are handled by [A-CALL-D] ([A-CALL-I]). Both the call and return instructions are assumed to be in the same module ($\mathtt{M} = \mathtt{M}'$). In [A-CALL-I], the target functions are found by $\mathtt{Resolve}(\mathtt{fp}(\mathtt{p}_1,\dots,\mathtt{p}_n))$, as explained shortly below.

(2) U- inference rules.* [U-FIELD] and [U-ARRAY] adds an unknown object $\mathtt{u}'_{\tilde{t}}$, where $t = T(\mathtt{p})$, to p's points-to set if q points to an unknown object $\mathtt{u}_{\_} \in \mathcal{U}$. For type casting, the type of an unknown object is narrowed for a downcast ([U-CAST-D]) but not an upcast ([U-CAST-U]), where <: denotes the subtyping relation.

[U-LOAD] reasons about reading from an unknown object at a LOAD `p = *q`. If q points to $\mathtt{u}_{\_}$, a new unknown object $\mathtt{u}_{\tilde{t}}$ is created, where $t = T(\mathtt{p})$, and added into p's points-to set. To trade precision for efficiency, we do not have a corresponding rule [U-STORE] for handling writes into an unknown object. This is sound as [U-LOAD] has over-approximated the effects of such writes.

If a global pointer $\mathtt{g} \in \mathcal{G}$ that is defined in module M escapes to another module M′ by our pre-analysis, [U-GLOBAL] assumes that any object $\mathtt{a} \in \mathcal{A}$ pointed to by g recursively and its subobjects are modified in M′. We use a points-to graph $ptg(\mathtt{g})$ to capture the escaped objects via g ([U-PTG] and [U-PTG-REC]). Every object $\mathtt{a} \in ptg(\mathtt{g})$ may point to an object $\mathtt{u}_{\tilde{t}}$, where $t \in ts(\mathtt{a})$. We will use a side-effect summary to filter spurious types thus introduced (Section 3.2.3).

[U-FUNCTION] handles the case when a function f is defined in M but called in another module M′. Conservatively, its parameter $\mathtt{p_i}$ may point to an unknown object $\mathtt{u}_{\tilde{t}}$ created in M′, where $t = T(\mathtt{p_i})$.

[U-CALL-D] analyzes a direct call $\mathtt{ret} = \mathtt{f}(\mathtt{p}_1,\dots,\mathtt{p}_n)$ in M with f defined in another module M′. Every object $\mathtt{a} \in ptg(\mathtt{p_i})$ passed indirectly through $\mathtt{p_i}$ to M′ may be modified in M′. Similarly, `ret` may point to an unknown object $u_{\tilde{t}}$, where $t = T(\mathtt{ret})$. [U-CALL-I] analyzes an indirect call $\mathtt{ret} = \mathtt{fp}(\mathtt{p}_1,\dots,\mathtt{p}_n)$ in M, where `fp` may point to a function defined in another module M′. [U-CALL-I] behaves identically as [U-CALL-D] except that the target functions invoked at the callsite are found by $\mathtt{Resolve}(\mathtt{fp}(\mathtt{p}_1,\dots,\mathtt{p}_n))$.

(3) $\mathtt{Resolve}(\mathtt{fp}(\mathtt{p}_1,\dots,\mathtt{p}_n))$. As shown in Figure 5, this auxiliary function, which is used in [A-CALL-I] and [U-CALL-I], resolves $\mathtt{fp}(\mathtt{p}_1,\dots,\mathtt{p}_n)$ to a target function $\mathtt{f}(\mathtt{q}_1,\dots,\mathtt{q}_n)$ 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.** $\mathtt{f}(\mathtt{q}_1,\dots,\mathtt{q}_n)$ is a target function if **C1** ∧ (**C2** ∨ **C3** ∨ **C4**) holds, where **C1** requires f to be a valid candidate function according to the C++ specification for virtual calls [1] and **C2** ∨ **C3** ∨ **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., $\mathtt{u}_{\_} \in pt(\mathtt{fp})$. If $\mathtt{p}_1$ points to a receiver object a of type t, then f is a target function, where $\mathtt{f} \in vfns(\mathtt{t})$ is a virtual function in t's vtable, provided that C1 also holds. **C4** is similar to **C3** except that **C4** handles the case when $\mathtt{p}_1$ points to an unknown receiver object of type $\mathtt{u}_t$.

(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, $\mathtt{f}(\mathtt{q}_1,\dots,\mathtt{q}_n)$ is a target function if **C5** ∧ (**C2** ∨ **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 $\mathtt{f} \in \mathcal{F}$ 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* $\mathtt{u_B}$ *and* $\mathtt{u_C}$. *By applying* [U-CALL-I], *where* **C1** ∧ **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*$_{\mathtt{Summary}}$*).* We make [U-GLOBAL], [U-CALL-D] and [U-CALL-I] in Figure 4 more precise, by refining their $ts(\mathtt{a})$ to $ts(\mathtt{a}) \cap \boxed{summary(\mathtt{f})}$ by using a side-effect type summary for every relevant function f. As a result, unknown objects with fewer types will be generated.

$$[\texttt{S-STORE}]\frac{\mathtt{*p = q \in f} \quad t = T(\mathtt{p})}{\{\,\tilde{t}\,\} \subseteq summary(\mathtt{f})} \qquad [\texttt{S-CALL-D}]\frac{\mathtt{ret} = \mathtt{f}'(\mathtt{p}_1,\dots,\mathtt{p}_n) \in \mathtt{f}}{summary(\mathtt{f}') \subseteq summary(\mathtt{f})}$$

$$[\texttt{S-CALL-I}]\frac{\mathtt{ret} = \mathtt{fp}(\mathtt{p}_1,\dots,\mathtt{p}_n) \in \mathtt{f} \quad \mathtt{f}'(\mathtt{q}_1,\dots,\mathtt{q}_n) \quad \forall \mathtt{i} \in \{1,\dots,n\}: T(\mathtt{q_i}) \cong T(\mathtt{p_i})}{summary(\mathtt{f}') \subseteq summary(\mathtt{f})}$$

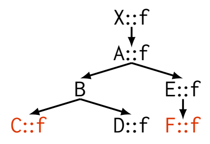
Figure 6: Type-based summarization for a function f.

Figure 6 gives the three rules for computing *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$_{\mathtt{Summary}}$). *With type-based summaries, we have refined the points-to sets from Figure 3(d) to Figure 3(e). As summary(setData) = {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* $\mathtt{U_B}$, *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)).*

```
void main() {
  A *p;
  ...
  p->f();
}
```

(a) Source code



(b) Class hierarchy

```
vtptr = *p;
assert(vtptr==&VTableC || vtptr==&VTableF);
vfn = &(vtptr[0]);
fp = *vfn;
call fp(p);
```

(c) Runtime checks (for vtables)

```
vtptr = *p;
vfn = &(vtptr[0]);
fp = *vfn;
assert(fp==&C::f || fp==&F::f);
call fp(p);
```

(d) Runtime checks (for virtual functions)

Figure 7: Protecting indirect calls with membership tests.

```
Candidate functions for f:  X::f A::f C::f D::f E::f F::f
IDs:                          0    1    2    3    4    5
```

(a) ID assignment for function f

```
Candidate functions for p->f():                C::f D::f E::f F::f
IDs:                                             2    3    4    5
Valid Vector (idmin = 2, idmax = 5):             1    0    0    1
```

(b) Valid vector construction for p->f()

```
1: vtptr = *p;
2: vfn = &(vtptr[0]);
3: fp = *vfn;
4: fid = *(fp-0x08);            //load function id
5: assert(idmin ≤fid≤ idmax);   //range check
6: flag_addr = &vec[fid-idmin]  //load valid bit
7: assert(*flag_addr==1);       //validity check
8: call fp(p);
```

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

Figure 8: Protecting indirect calls with an index-based instrumentation, illustrated for the code in Figure 7(a).

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 $\mathbb{T}_f$ = {`X::f`,`A::f`,`C::f`,`D::f`,`E::f`,`F::f`}, is given in Figure 7(b).

Suppose $\mathbb{T}_f^p$ = {`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 $\mathbb{T}_f^p$, a set of statically determined legitimate targets. However, such a set-membership test is impractically expensive for large programs, as $\mathbb{T}_f^p$ can be large. For example, Chrome has large and deep class hierarchies consisting of thousands of classes. $\mathbb{T}_f^p$ 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 Ovt/Ivt [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 – &VTableF 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 <= &VTableF);

Unfortunately, the set of legitimate targets allowed is $\mathbb{T}_f$ = { `A::f`, `C::f`, `D::f`, `E::f`, `F::f`} rather than $\mathbb{T}_f^p$ = { `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 singe range check would have to be split into multiple range checks, which are costly. Finally, reordering/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 $\mathbb{C}_f$ be the class hierarchy, which is always a DAG in C++, that contains all the possible implementations of f, denoted $\mathbb{T}_f$, in the program. We assume that $\mathbb{C}_f$ 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 $\mathbb{T}_f$. To facilitate a word-based optimization (OneWordOpt) discussed in Section 3.3.1, we will number the functions in $\mathbb{T}_f$ continuously, starting from 0, by traversing $\mathbb{C}_f$ in DFS. In our example, $\mathbb{T}_f$ = {`X::f`,`A::f`,`C::f`,`D::f`,`E::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 $\mathbb{T}_f$, 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, $\mathbb{T}_f^p$, found by pointer analysis. Let $\mathtt{id}_{\min}$ and $\mathtt{id}_{\max}$ be their smallest and largest IDs, respectively. We create a valid vector $\mathtt{vec}_\mathtt{f}^\mathtt{A}[0 : \mathtt{id}_{\max} - \mathtt{id}_{\min}]$ and set $\mathtt{vec}_\mathtt{f}^\mathtt{A}[i]$ to 1 if $\mathbb{T}_f^p$ contains a function with ID, fid, such that $fid - \mathtt{id}_{\min} = i$, and 0 otherwise. In our example, $\mathbb{T}_f^p$ = {`C::f`,`F::f`}. Thus, its valid vector is initialized as shown in Figure 8(b).

(4) **Inserting Runtime Checks.** For every virtual callsite `p->f()`, we insert the runtime checks, as shown in Figure 8(c), to allow only the targets in $\mathbb{T}_f^p$ 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 callsite is independent of the callsite protected and class hierarchies used.

	Binary value	Decimal value
`Valid vector`	0...01001	9

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

```
1: vtptr = *p;
2: vfn = &vtptr[0];
3: fp = *vfn;
4: fid = *(fp-0x08);                 //load function id
5: assert(id_min ≤fid≤ id_max);      //range check
6: assert((9>>(id_max-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 $0 \cdots 01001$ (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.

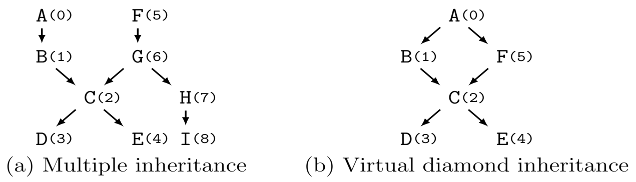


Figure 10: Complex class hierarchies in C++, with the IDs assigned for a function assumed to be defined in each class.

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 $\mathbb{C}_f$ 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 $\mathbb{C}_f$ 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 $\mathbb{C}_f$ 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.

Table 2: Program characteristics.

Program	KLOC	#Stmt	#Ptrs	#Objs	#CallSite
dealII	199	577482	530249	77894	94284
eon	41	65218	63385	15855	14033
omnetpp	48	95961	108349	8592	20601
povray	155	126484	189118	8195	15226
soplex	41	54190	69287	4191	9878
xalan	553	744971	703675	73973	106090
Chrome	15224	20362892	34173201	1685228	3566682
Total	16261	22027198	35837264	1873928	3826794

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., 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 `[A-CALL-I]` and `[U-CALL-I]` (Figure 4). Points-to sets are represented using LLVM's 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

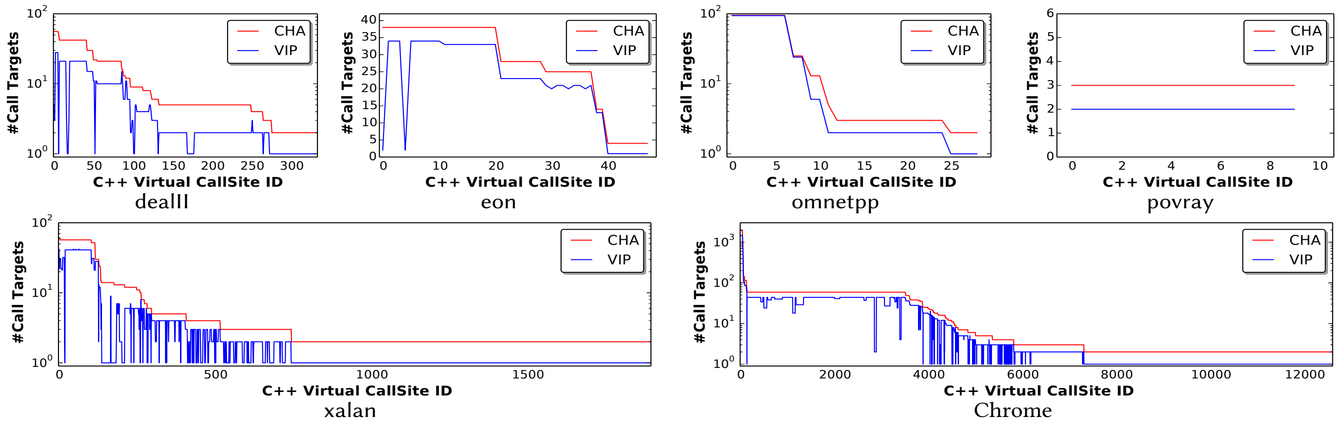


Figure 11: Precision comparison at callsites where VIP and resolves fewer targets than CHA.

Table 3: Instrumentation statistics of VIP.

Program	#CallSites	#Callsites with different checks				
		1	2	3	4~64	> 64
dealII	902	306	157	59	378	2
eon	91	31	3	0	57	0
omnetpp	686	465	83	26	105	7
povray	122	95	20	7	0	0
soplex	553	420	91	14	28	0
xalan	9342	4500	1637	659	2469	77
Chrome	66609	37874	9630	2050	15882	1173
Total	78305	43691	11621	2815	18919	1259

Table 4: Target functions resolved by VIP and CHA.

Program	#Callsite					Analysis time	
	VIP =CHA	VIP <CHA	More Found by VIP			CHA	VIP
			1	2	3		
dealII	568	334	77	131	14	1.3secs	77.9secs
eon	43	48	8	2	0	0.1secs	3.9secs
omnetpp	657	29	4	14	0	0.1secs	21.1secs
povray	112	10	0	10	0	0.1secs	53.7secs
soplex	549	4	4	0	0	0.1secs	4.7secs
xalan	7445	1897	1423	208	66	2.2secs	4780.0secs
Chrome	53994	12615	5468	1507	771	2188.0secs	6.1hrs
Total	63368	14937	6984	1872	851	2192.0secs	7.4hrs

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 (*-flto*). 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 defined in the executable being called from the shared libraries, and 150360 callsites in the executable calling functions defined in the libraries. As far as we know, VIP is the first sound pointer analysis that scales to multi-MLOC C++ programs, and also the first partial analysis that works at this scale. We also use six SPEC programs to compare against the CHA-based approach for both precision and performance. Every SPEC program consists of a single module. Thus, our partial pointer analysis becomes a full analysis.

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 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 dealII, 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.

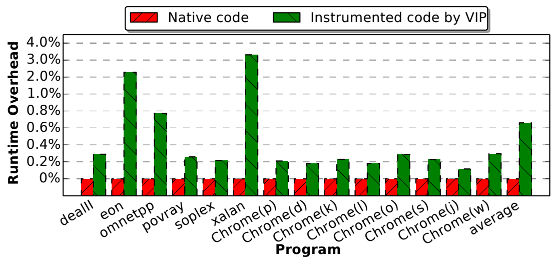


Figure 12: Runtime overhead of VIP against the native run.

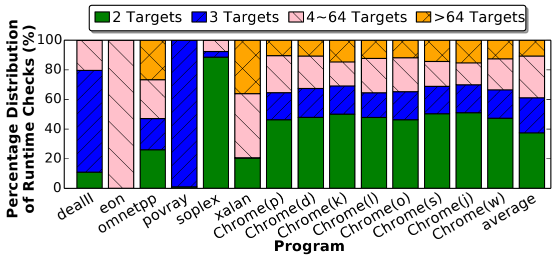


Figure 13: Distribution of runtime checks inserted by VIP.

Analysis Times. These are given in the last two columns in Table 4 for CHA and VIP. It takes 6.1 hours for VIP to analyze the largest program (Chrome). The only SPEC program that takes more than one hour to finish is xalan as it has an order of magnitude more virtual callsites than the other SPEC programs (Table 3). VIP can analyze each remaining program within two minutes.

Runtime Overhead. We measure the performance of the SPEC programs using their reference inputs. For Chrome, we use eight industry browser benchmarks, peacekeeper [14], dromaeo [28], kraken [29], litebrite [27], octane [16], sunspider [7], jetstream [10] and wirple-bmark [5], with the first six being also used in [9, 17, 50].

Figure 12 compares the runtime overheads of VIP with the native runs. Figure 13 shows the percentages of different kinds of runtime checks over the total. For Chrome, there are eight bars, one for each browser benchmark used, identified by the first letter of its name. The overheads for eon (2.2%) and xalan (3.3%) are over 1% due to their relatively high percentages of unoptimized or partially optimized checks (by MemTSTOpt or OneWordOpt). For Chrome, the overheads are relatively small, because the number of lightweight checks (for ≤ 3 targets) is around 60% of all checks executed.

VIP incurs comparable overheads as OVT/IVT [9]. For the five common programs used, omnetpp, povray, soplex, xalan and Chrome (evaluated under kraken, octane and sunspider), VIP has an average overhead of 0.76%, compared to 1.17% reported in their paper.

Space Overhead. Due to our index-based instrumentation, there are slight increases in terms of LLVM IR instructions used, ranging from to 0.3% (povray) to 6.5% (xalan) with an average of 2.5%. The space overheads (due to the use of valid vectors and 64-bit words) range from 0 KB (povray) to 263.4KB (Chrome) with an average of 42KB, which are consistent with the results reported in Table 3.

A Proof-of-Concept Attack. Consider Figure 1, representing a real scenario for Chrome. For `p->f()`, we use an attack to corrupt the vtable pointer in an object pointed to by `p` by exploiting a use-after-free or buffer overflow vulnerability. Then `p->f()` 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 callsites, 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

[1] *C++ specification.* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf.

[2] *Clang: Control flow integrity design documentation.* http://clang.llvm.org/docs/ControlFlowIntegrityDesign.html.

[3] *MicroSoft data execution prevention.* https://support.microsoft.com/en-au/kb/875352.

[4] *PaX.* https://pax.grsecurity.net/docs/aslr.txt.

[5] *Wirple BMark HTML5 3D benchmark.* https://www.wirple.com/bmark/.

[6] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *CCS '05*. ACM, 340–353.

[7] Apple. *Sunspider 1.0.2 javascript benchmark suite.* Apple. https://webkit.org/perf/sunspider/sunspider.html.

[8] George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive Points-to analysis for C and C++. In *SAS '16*. 84–104.

[9] Dimitar Bounov, R Kici, and Sorin Lerner. 2016. Protecting C++ dynamic dispatch through vtable interleaving. In *NDSS '16*.

[10] BrowserBench. *JetStream JavaScript performance test suite.* BrowserBench. http://browserbench.org/JetStream/.

[11] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.. In *USENIX Security '98*. 63–78.

[12] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95*. Springer, 77–101.

[13] Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo MK Martin, and Steve Zdancewic. 2013. Ironclad C++: A library-augmented type-safe subset of C++. In *OOPSLA '13*. 287–304.

[14] FutureMark. *Peacekeeper: HTML5 browser speed test.* FutureMark. http://peacekeeper.futuremark.com/.

[15] Robert Gawlik and Thorsten Holz. 2014. Towards automated integrity protection of C++ virtual function tables in binary programs. In *ACSAC '14*. ACM, 396–405.

[16] Google. *Octane JavaScript benchmark suite.* Google. https://developers.google.com/octane/.

[17] Istvan Haller, Enes Göktas, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. Shrinkwrap: Vtable protection without loose ends. In *ACSAC '15*. ACM, 341–350.

[18] Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*. ACM, 290–299.

[19] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *CGO '11*. 289–298.

[20] ISO90. 1990. *ISO/IEC. international standard ISO/IEC 9899, programming languages - C.*

[21] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: securing C++ virtual calls from memory corruption attacks. In *NDSS '14*.

[22] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-pointer integrity. In *OSDI '14*. 147–163.

[23] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI '07*. New York, NY, USA, 278–289.

[24] Julian Lettner, Benjamin Kollenda, Andrei Homescu, Per Larsen, Felix Schuster, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Michael Franz. 2016. Subversive-C: Abusing and protecting dynamic message dispatch. In *USENIX ATC '16*. 209–221.

[25] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *POPL '11*. 3–16.

[26] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE '11*. ACM, 343–353.

[27] Microsoft. *LiteBrite: HTML, CSS and JavaScript Performance Benchmark.* Microsoft. https://testdrive-archive.azurewebsites.net/Performance/LiteBrite/.

[28] Mozilla. *Dromaeo JavaScript performance test suite.* Mozilla. http://dromaeo.com/.

[29] Mozilla. *Kraken 1.1 Javascript benchmark suite.* Mozilla. http://krakenbenchmark.mozilla.org/.

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI '09*. ACM, 245–258.

[31] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *ISMM '10*, Vol. 45. ACM, 31–40.

[32] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. In *TOPLAS '05*. ACM, 477–526.

[33] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *CCS '15*. ACM, 914–926.

[34] Esko Nuutila and Eljas Soisalon-Soininen. 1994. On finding the strongly connected components in a directed graph. *Inform. Process. Lett.* 49, 1 (1994), 9–14.

[35] D.J. Pearce, P.H.J. Kelly, and C. Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems* 30, 1 (2007), 106–148.

[36] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *NDSS '15*.

[37] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *S&P '15*. IEEE, 745–762.

[38] Yulei Sui, Xiaokang Fan, Hao Zhou, and Jingling Xue. 2016. Loop-oriented array- and field-sensitive pointer analysis for Automatic SIMD Vectorization. In *LCTES '16*. ACM, 41–51.

[39] Yulei Sui, Yue Li, and Jingling Xue. 2013. Query-directed adaptive heap cloning for optimizing compilers. In *CGO '13*. 1–11.

[40] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-Flow Refinement. In *FSE '16*. 460–473.

[41] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. https://github.com/unsw-corg/SVF. In *CC '16*. 265–266.

[42] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*. 254–264.

[43] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transacstions on Software Engineering* 40, 2 (2014), 107–122.

[44] Yulei Sui, Sen Ye, Jing Xue, and Pen-Chung Yew. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA. In *APLAS '11*. 155–171.

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

[46] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security '14*. 941–955.

[47] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating

advanced code-reuse attacks at the binary level. In *S&P '16*. 934–953.

[48] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-based selective flow-sensitive pointer analysis. In *SAS '14*. Springer, 319–336.

[49] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*. ACM, 218–229.

[50] Chao Zhang, Scott A Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. 2016. VTrust: Regaining trust on virtual calls. In *NDSS '16*.

[51] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting virtual function tables' integrity.. In *NDSS '15*.