TIPS: Tracking Integer-Pointer Value Flows for C++ Member Function Pointers

CHANGWEI ZOU, UNSW Sydney, Australia
DONGJIE HE, UNSW Sydney, Australia and Chongqing University, China
YULEI SUI, UNSW Sydney, Australia
JINGLING XUE, UNSW Sydney, Australia

C++ is crucial in software development, providing low-level memory control for performance and supporting object-oriented programming to construct modular, reusable code structures. Consequently, tackling pointer analysis for C++ becomes challenging, given the need to address these two fundamental features. A relatively unexplored research area involves the handling of C++ member function pointers. Previous efforts have tended to either disregard this feature or adopt a conservative approach, resulting in unsound or imprecise results.

C++ member function pointers, handling both virtual (via virtual table indexes) and non-virtual functions (through addresses), pose a significant challenge for pointer analysis due to the mix of integers and pointers, often resulting in unsound or imprecise analysis. We introduce Tips, the first pointer analysis that effectively manages both pointers and integers, offering support for C++ member function pointers by tracking their value flows. Our evaluation on Tips demonstrates its accuracy in identifying C++ member function call targets, a task where other tools falter, across fourteen large C++ programs from SPEC CPU, Qt, LLVM, Ninja, and GoogleTest, while maintaining low analysis overhead. In addition, our micro-benchmark suite, complete with ground truth data, allows for precise evaluation of points-to information for C++ member function pointers across various inheritance scenarios, highlighting Tips's precision enhancements.

CCS Concepts: • **Software and its engineering → Compilers**; • **Security and privacy → Software and application security**.

Additional Key Words and Phrases: C++ Member Function Pointers, Pointer Analysis, Integer-Pointer Value Flows

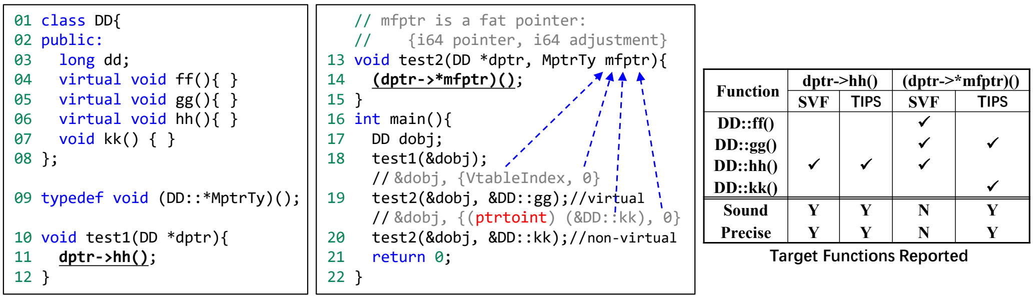
ACM Reference Format:
Changwei Zou, Dongjie He, Yulei Sui, and Jingling Xue. 2024. TIPS: Tracking Integer-Pointer Value Flows for C++ Member Function Pointers. *Proc. ACM Softw. Eng.* 1, FSE, Article 72 (July 2024), 23 pages. https://doi.org/10.1145/3660779

1 INTRODUCTION

Pointers are crucial in system programming languages like C and C++, enabling efficient memory management, data manipulation, and low-level operations for optimized software development. Pointer analysis [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, Yu et al., 2010], a key static analysis technique, aims to over-approximate a pointer's potential targets, supporting diverse applications such as bug detection [Cai et al., 2021, Liu et al., 2016, Livshits and Lam, 2003, Yan et al., 2018], information flow analysis [Arzt et al., 2014, Schubert

Authors' Contact Information: Changwei Zou, UNSW Sydney, Sydney, Australia, changwei.zou@unswalumni.com; Dongjie He, UNSW Sydney, Sydney, Australia and Chongqing University, Chongqing, China, dongjieh@cse.unsw.edu.au; Yulei Sui, UNSW Sydney, Sydney, Australia, y.sui@unsw.edu.au; Jingling Xue, UNSW Sydney, Sydney, Australia, j.xue@unsw.edu.au.

ACM 2994-970X/2024/7-ART72
https://doi.org/10.1145/3660779



```
01 class DD{
02 public:
03   long dd;
04   virtual void ff(){ }
05   virtual void gg(){ }
06   virtual void hh(){ }
07   void kk() { }
08 };

09 typedef void (DD::*MptrTy)();

10 void test1(DD *dptr){
11   dptr->hh();
12 }
```

```
   // mfptr is a fat pointer:
   //    {i64 pointer, i64 adjustment}
13 void test2(DD *dptr, MptrTy mfptr){
14   (dptr->*mfptr)();
15 }
16 int main(){
17   DD dobj;
18   test1(&dobj);
     // &dobj, {VtableIndex, 0}
19   test2(&dobj, &DD::gg);//virtual
     // &dobj, {(ptrtoint) (&DD::kk), 0}
20   test2(&dobj, &DD::kk);//non-virtual
21   return 0;
22 }
```

Function	dptr->hh()		(dptr->*mfptr)()	
	SVF	TIPS	SVF	TIPS
DD::ff()			✓	
DD::gg()			✓	✓
DD::hh()	✓	✓	✓	
DD::kk()				✓
Sound	Y	Y	N	Y
Precise	Y	Y	N	Y

Target Functions Reported

Fig. 1. A motivating example comparing targets reported by the state-of-the-art pointer analysis tool, SVF, and our tool, Tips, for the virtual call `dptr->hh()` and the member function call `(dptr->*mfptr)()`.

et al., 2019, Trabish et al., 2018], symbolic execution [Cadar et al., 2008, Trabish et al., 2018], and compiler optimization [Lattner and Adve, 2004, Lattner et al., 2007, Sui et al., 2013]. However, existing algorithms and frameworks for C and C++ like SVF [Sui and Xue, 2016b, 2024, Sui et al., 2014] and Pinpoint [Shi et al., 2018] typically focus on pointer variables, often overlooking integer variables involved in pointer casting, resulting in unsound or overly conservative analyses.

In this paper, we address the challenge of resolving C++ member function calls, emphasizing the need to track flows between integers and pointers. In C++, indirect calls are categorized into C-style, virtual, and member function calls. C-style calls are conventionally analyzed [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, Yu et al., 2010]. Virtual calls (e.g., line 11 in Figure 1) add a layer of indirection, where the target function of a virtual call is determined by accessing the class's virtual table via the object's virtual table pointer, then using a constant integer offset for a virtual table lookup to identify the specific virtual function.

However, member function calls involving integer variables, exemplified by line 14 in Figure 1, challenge existing pointer analysis algorithms [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, Yu et al., 2010]. These indirect calls, using member function pointers for both virtual and non-virtual functions, are often represented as fat pointers in, e.g., LLVM [LLVM, 2024], merging a function pointer with byte-level object pointer adjustments. This demands the tracking of virtual table indexes for virtual functions and direct addresses for non-virtual functions. However, existing pointer analyses like SVF [Sui and Xue, 2024] struggle with accurately resolving these calls as it does not track integer-pointer flows, highlighted by the blue dotted lines in Figure 1. Thus, typical prior work in the field misses the non-virtual function `DD::kk()` (unsoundness) and wrongly identifies all three virtual functions as potential targets (imprecision).

To address the complexities of analyzing C++ member function pointers, particularly amidst C++'s challenging multiple and virtual inheritance structures, we introduce Tips (**T**racking **I**nteger-**P**ointer value flows), an open-source framework [Zou, 2024] developed in LLVM. Tips elevates pointer analysis in C++ through a field-, flow-, and context-sensitive approach, uniquely tracking both pointer and integer value flows. This is achieved through the modeling of integer constants as the initial points-to information for integer variables and the representation of casts between integers and pointers using the classic Copy rule (as detailed in Section 3.1).

Moreover, Tips adeptly handles byte-level pointer adjustments critical for multiple inheritance by utilizing the Gep_c rule (`p = gep q, c`), leveraging LLVM's `getelementptr` instruction for precise tracking and transformation into flattened field indexes, where `q` represents a pointer and `c` denotes a constant integer. As a result, Tips can precisely identify correct virtual tables for class objects in complex initialization scenarios, including those with multiple virtual table pointers implicitly generated by C++ compilers like LLVM (as described in Section 3.2).

Finally, Tips handles byte-level pointer adjustments manifested as integer variables, especially in member function pointers with virtual inheritance. It uses the $\textsc{Gep}_k$ rule (`p = gep q, k`), with `q` as a pointer and `k` as an integer variable, to refine points-to sets by treating integers as pointers (`PointsTo(k)`). This approach converts $\textsc{Gep}_k$ into several $\textsc{Gep}_c$ instances (`p = gep q,` $\mathtt{c}_i$, where $\mathtt{c}_i$ is a pointed-to element in `PointsTo(k)`), introducing field sensitivity into $\textsc{Gep}_k$ and addressing the imprecision of previous field-insensitive approaches [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, 2024, Yu et al., 2010].

Our evaluation shows that Tips can precisely resolve C++ member function calls missed by current leading solutions in large C++ applications, imposing small overhead. It underscores the significance of C++ member function pointers, including in the C++ STL (Standard Template Library). Additionally, we introduce a micro-benchmark suite that provides a quantitative measure for evaluating pointer information in C++ inheritance cases, demonstrating enhanced precision.

This paper's key contributions include: (1) the first pointer analysis for supporting C++ member function pointers, in all inheritance scenarios, including single, multiple, and virtual inheritance, field-, flow-, and context-sensitively by tracking integer-pointer value flows, (2) a prototyping implementation of Tips in LLVM; and (3) an evaluation of Tips in terms of its precision improvements and analysis overhead using both a micro-benchmark suite and fourteen C++ programs.

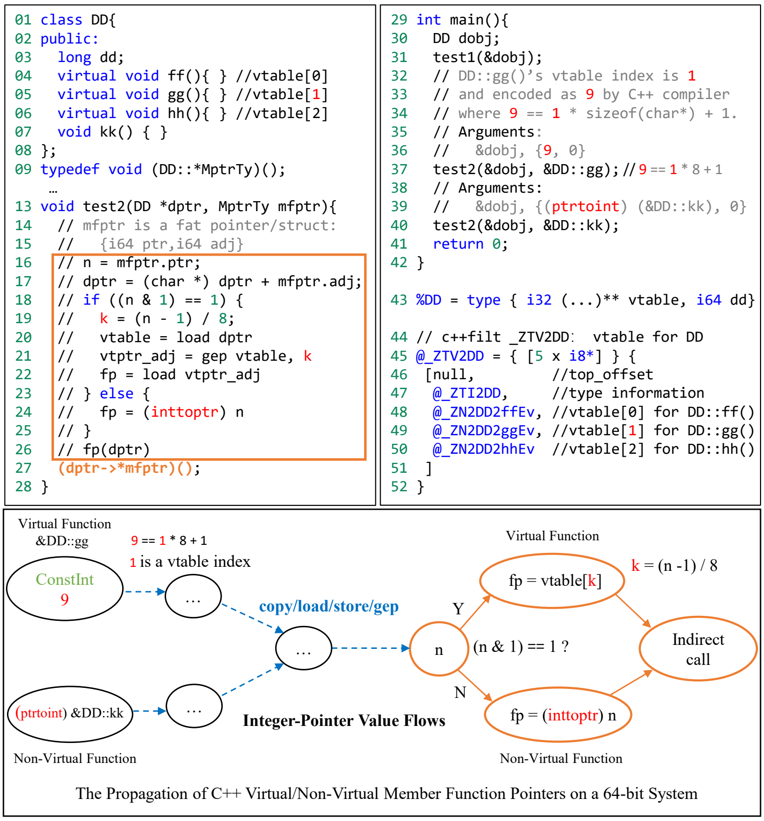
2 WHY MUST WE ANALYZE INTEGERS FOR C++ PROGRAMS?

Revisiting our motivating example, initially given in Figure 1 and now in Figure 2, highlights the crucial role of tracking integer-pointer value flows in analyzing its member function call at line 27. For class `DD`, C++ compilers emit a virtual table (lines 45–52) listing the names of all three virtual functions (lines 48–50) in the order defined in the source code (lines 4–6), with names demangled by `c++filt` (line 44). The LLVM-IR representation for `DD` (line 43) includes two primary data members: the virtual table pointer initialized in `DD`'s constructor and the `dd` data member (line 3).

The `typedef` statement at line 9 defines a C++ member function pointer type for class `DD`, facilitating member function calls like `(dptr->*mfptr)()` at line 27 in LLVM. Here, `mfptr` acts as a fat pointer comprising two fields: (1) a function pointer for non-virtual functions or a virtual table index for virtual functions, and (2) a byte-level offset for adjusting the object pointer, such as `dptr`. For the virtual function `DD::gg()`, identified at line 37, its virtual table index (1) is encoded as an odd integer (9) by multiplying with the pointer's size and adding one, with a zero byte offset for adjustments. As a result, the arguments passed to `test2()` become `&dobj` and the fat pointer `{9, 0}`. Non-virtual function addresses, like `&DD::kk` at line 40, are cast to integers using `ptrtoint` and paired with a zero byte offset to form a fat pointer. It is worth noting that `&DD::kk` always results in an even integer due to the linker's 16-byte alignment for function code entries.

The pseudo-code at lines 16–26 outlines the low-level steps required for the high-level member function call `(dptr->*mfptr)()` at line 27. At line 16, we obtain the encoded virtual index or non-virtual function address (cast into an integer) and store it in `n`. Line 17 handles byte-level object pointer adjustment. We distinguish between virtual and non-virtual functions by checking if `n` is odd or even at line 18. If `n` is odd (representing an encoded virtual table index), line 19 decodes it, with `k` now representing the actual virtual table index. Subsequently, we load the virtual table pointer at line 20, adjust it at line 21, and retrieve the corresponding virtual function address from the virtual table at line 22. If `n` is even, it is cast back into a function pointer (containing the non-virtual member function address) at line 24. In both cases, the indirect call is made at line 26.

In the lower section of Figure 2, we highlight in orange the treatment of C++ virtual and non-virtual member function pointers, `&DD:gg` and `&DD:kk`, as integers on a 64-bit system with 8-byte



```
01 class DD{
02 public:
03   long dd;
04   virtual void ff(){ } //vtable[0]
05   virtual void gg(){ } //vtable[1]
06   virtual void hh(){ } //vtable[2]
07   void kk() { }
08 };
09 typedef void (DD::*MptrTy)();
    …
13 void test2(DD *dptr, MptrTy mfptr){
14   // mfptr is a fat pointer/struct:
15   //   {i64 ptr,i64 adj}
16   // n = mfptr.ptr;
17   // dptr = (char *) dptr + mfptr.adj;
18   // if ((n & 1) == 1) {
19   //   k = (n - 1) / 8;
20   //   vtable = load dptr
21   //   vtptr_adj = gep vtable, k
22   //   fp = load vtptr_adj
23   // } else {
24   //   fp = (inttoptr) n
25   // }
26   // fp(dptr)
27   (dptr->*mfptr)();
28 }
```

```
29 int main(){
30   DD dobj;
31   test1(&dobj);
32   // DD::gg()'s vtable index is 1
33   // and encoded as 9 by C++ compiler
34   // where 9 == 1 * sizeof(char*) + 1.
35   // Arguments:
36   //   &dobj, {9, 0}
37   test2(&dobj, &DD::gg); // 9 == 1 * 8 + 1
38   // Arguments:
39   //   &dobj, {(ptrtoint) (&DD::kk), 0}
40   test2(&dobj, &DD::kk);
41   return 0;
42 }

43 %DD = type { i32 (...)** vtable, i64 dd}

44 // c++filt _ZTV2DD:  vtable for DD
45 @_ZTV2DD = { [5 x i8*] } {
46  [null,          //top_offset
47   @_ZTI2DD,      //type information
48   @_ZN2DD2ffEv, //vtable[0] for DD::ff()
49   @_ZN2DD2ggEv, //vtable[1] for DD::gg()
50   @_ZN2DD2hhEv  //vtable[2] for DD::hh()
51  ]
52 }
```

Fig. 2. Understanding the intricacies of statically analyzing C++ member function pointers in the one-class example depicted in Figure 1, which emulates the single inheritance scenario for simplicity.

pointers. This exposes a gap in existing pointer analysis algorithms [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, 2024, Yu et al., 2010], which often overlook integer-pointer value flows, leading to inaccuracies. For example, SVF [Sui and Xue, 2024] tracks only the object pointer &dobj's value flow, missing virtual table indexes and addresses of non-virtual member functions. This results in unsound and imprecise analyses, as evidenced in Figure 1. Notably, because the virtual table index k for DD is an integer variable—not a constant—SVF and similar field-sensitive analyses become field-insensitive for such scenarios. Consequently, SVF incorrectly identifies all virtual functions in DD's virtual table as potential targets for the call at line 27 and misses the non-virtual function &DD::kk at line 40.

3 TIPS: APPROACH

We introduce Tips, a novel approach for resolving C++ member function pointers. Section 3.1 compares Tips with existing frameworks like SVF [Sui and Xue, 2024], outlining fundamental differences. Section 3.2 explores the complexities of C++ compilation, particularly byte-level pointer adjustments (Gep$_c$ in Table 1), and how C++ compilers manage member function calls in multiple inheritance scenarios, addressing common inaccuracies. Section 3.3 details how Tips tracks points-to information for integers and pointers, vital for managing C++ member function pointers in LLVM-IR. Finally, Section 3.4 demonstrates Tips's effectiveness in handling member function

Table 1. The key distinction between prior work (exemplified by SVF) and our work (Tips) in modeling five types of statements in C++ programs, with their inference rules explained in the text (Section 3.1).

Constraint		Pointer Adjustments	Prior Work		Our Work	
		in LLVM IR code	C-Style	Domains	IR-Style	Domains
AddrOf		NO	p = &obj	p ∈ Pointer	p = alloc obj	p ∈ Pointer
					i = c	i ∈ Integer, c ∈ ConstInt
Copy		NO	p = q	p, q ∈ Pointer	p = q	p, q ∈ Pointer
					i = j	i, j ∈ Integer
					p = inttoptr i	p ∈ Pointer, i ∈ Integer
					i = ptrtoint p	i ∈ Integer, p ∈ Pointer
Store		NO	*q = p	p, q ∈ Pointer	store p, q	p ∈ Pointer ∪ Integer, q ∈ Pointer
Load		NO	p = *q	p, q ∈ Pointer	p = load q	p ∈ Pointer ∪ Integer, q ∈ Pointer
Gep	Gep$_{idx}$	field-index-based	p = &(q->fld)	p ∈ Pointer, q ∈ StructPointer	p = gep q, idx[]	p ∈ Pointer, idx ∈ IntegerArray q ∈ ArrayPointer ∪ StructPointer
			p = &q[k] p = &q[c]	p ∈ Pointer, q ∈ ArrayPointer k ∈ Integer, c ∈ ConstInt		
	Gep$_k$	byte-level	p = q + k	p ∈ SingleValuePointer q ∈ SingleValuePointer k ∈ Integer	p = gep q, k	p, q ∈ SingleValuePointer, k ∈ Integer
	Gep$_c$	byte-level	p = q + c	p ∈ SingleValuePointer q ∈ SingleValuePointer c ∈ ConstInt	p = gep q, c	p, q ∈ SingleValuePointer, c ∈ ConstInt

pointers in the complex setting of C++ virtual inheritance, showing enhanced soundness and precision over SVF.

3.1 Pointer Analysis

There are five key rules for analyzing C++ statements: AddrOf, Copy, Store, Load, and Gep (which stands for the `getelementptr` instruction in LLVM [LLVM, 2024], enhancing field-sensitivity). AddrOf initiates points-to information, with the other rules facilitating its propagation. In Table 1, we present these statements in C-style for SVF [Sui and Xue, 2024] and LLVM-IR for Tips, using LLVM-IR in Tips to effectively track integer-pointer value flows within C++ member functions at the LLVM-IR level. We will later explore their inference rules for both SVF and Tips.

Similar to SVF [Sui and Xue, 2024], Tips is field-, flow-, and context-sensitive. Tips maintains flow-sensitivity (by adhering to control flow) and context-sensitivity (by distinguishing different calling contexts of functions). However, what distinguishes Tips from SVF is its approach to field-sensitive tracking of integer-pointer value flows when analyzing C++ member function calls. In LLVM-IR, Tips tracks pointer adjustments within objects containing multiple fields, involving two types of adjustments: field-index-based and byte-level adjustments. Field-index-based adjustments follow the Gep$_{idx}$ rule, where `q` points to an `AggregateType` object (e.g., struct or array), and `idx` is an `IntegerArray` representing unflattened field indexes. Byte-level adjustments are managed by Gep$_k$ and Gep$_c$, where `q` refers to a `SingleValueType` object (e.g., int and char*), with `k` (an integer variable) for Gep$_k$ and `c` (a constant integer) for Gep$_c$. In C++ programs, ideally, only Gep$_{idx}$ should suffice for field-index-based adjustments. However, byte-level adjustments are common, e.g., in non-standard macros like `container_of()` [Koschel et al., 2023]. Additionally, C++ compilers may generate LLVM instructions requiring byte-level adjustments, e.g., for multiple inheritance using Gep$_c$, as illustrated in Figure 4, an aspect ignored by SVF. Analyzing member function pointers, particularly within the context of virtual inheritance—an area conservatively handled by SVF—necessitates the use of Gep$_k$. Utilizing Gep$_c$ to address multiple inheritance, Tips handles the intricate semantics of initializing C++ virtual table pointers within C++ objects. By treating `k` as a pointer, not an integer variable, Tips achieves field-sensitive handling of Gep$_k$. This involves tracking `k`'s points-to information (`PointsTo(k)`) and converting a single Gep$_k$ application into several Gep$_c$ instances (`p = gep q, c`$_i$ where `c`$_i$ ∈ `PointsTo(k)`). This refinement rectifies the imprecision inherent in SVF [Sui and Xue, 2024], which conservatively approximates this rule with field insensitivity for C++ member function pointers. SVF assumes that k can represent any virtual table index in a class's virtual table (as illustrated in Figure 2).

Let us examine Table 1 to highlight the main differences between SVF [Sui and Xue, 2024] and Tips, while also defining the functionality of each inference rule used.

AddrOf. In Tips, constant integers, such as the one represented by 9 in Figure 2, are treated as constant objects. To achieve this, Tips introduces an additional rule, AddrOf (`i = c`), where `i` is an integer variable and `c` is a constant integer. This rule initializes the points-to information for `i`, ensuring that `c` belongs to the points-to set of `i`, denoted as `PointsTo(i)`.

Copy. Compared to SVF, Tips addresses three additional copy statements. For the assignment `i = j`, Tips transfers the points-to information from one integer variable `j` to another, `i`, ensuring that `PointsTo(j) ⊆ PointsTo(i)`. The remaining two cases, `p = inttoptr i` and `i = ptrtoint p`, handle the LLVM instructions `inttoptr` and `ptrtoint` used for casting between integers and pointers. In the case of `p = inttoptr i`, the points-to information is copied from `i` to `p`, while `i = ptrtoint p` requires transferring the points-to information from `p` to `i`.

Store. The Store rule for handling a store statement "`store p, q`" in previous work [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, 2024, Yu et al., 2010] applies exclusively to pointers. In contrast, in Tips, `p` can be either an integer or a pointer, while `q` is always a pointer. For every object `obj` pointed to by `q`, Tips propagates the points-to information from `p` to `obj`, ensuring that `PointsTo(p) ⊆ PointsTo(obj)`.

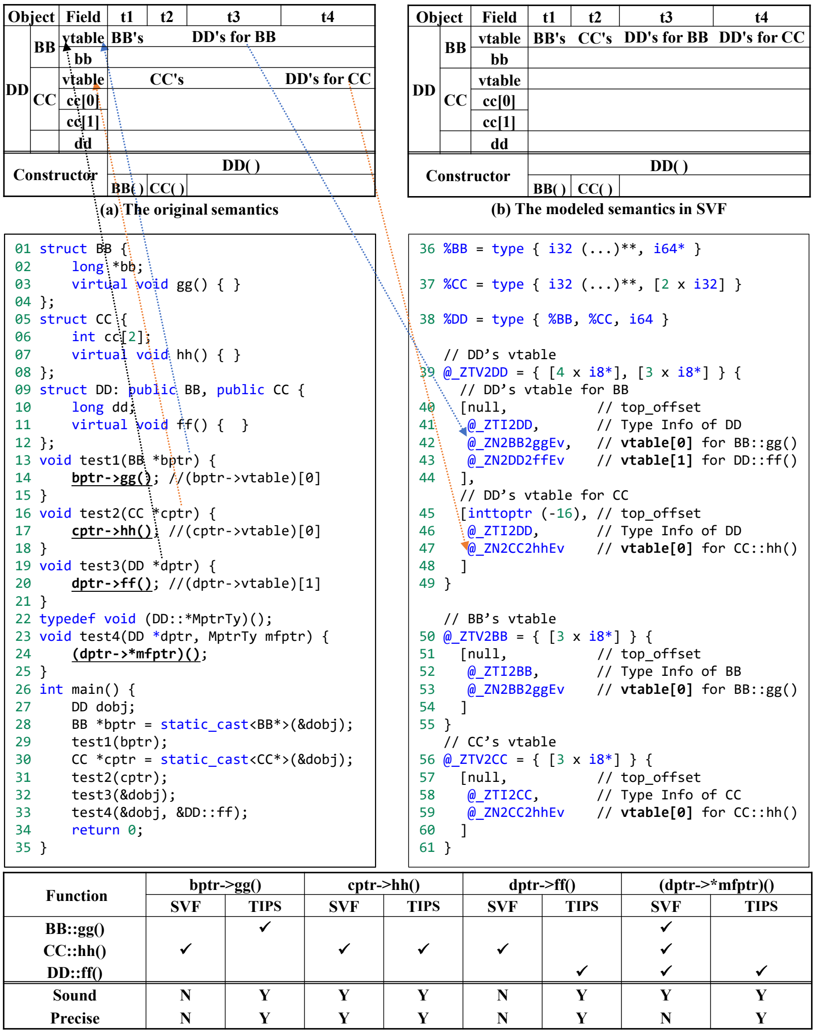
Load. Load statements, just like store statements, have been traditionally restricted to pointers in prior research [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, 2024, Yu et al., 2010]. However, Tips broadens this by allowing `p` in `p = load q` to be an integer or a pointer, with `q` being a pointer. For each `obj` that `q` points to, Tips updates `p`'s points-to set to include `obj`'s points-to information, maintaining `PointsTo(obj) ⊆ PointsTo(p)`.

Gep. Both Tips and SVF handle field-index-based field accesses similarly, using Gep$_{idx}$. However, when it comes to byte-level field accesses, SVF is field-insensitive in Gep$_k$ (assuming that `k` ranges over all possible field offsets) and field-sensitive only in Gep$_c$. In contrast, Tips maintains field sensitivity in both cases. Tips consistently tracks byte offsets along with flattened field indexes field-sensitively (as shown in Figure 5). To enable field-sensitive analysis for cases like `vtable[k]` in Figure 2, where `k` is an integer variable, Tips maintains points-to information for `k` in the Gep$_k$ rule. In addition, in the Gep$_c$ rule, Tips handles byte-level pointer adjustments, often necessary for C++ multiple inheritance, where `q` is a char pointer and `c` is a constant integer. Here, Tips tracks byte-level offsets and converts them into flattened field indexes for field-sensitive analysis.

Table 2. Unflattened/flattened field indexes and byte offsets in the layered object model.

Object/Sub-Object		Field	Unflattened Field Index in an Object				Flattened Field Index in DD	Byte Offset in DD
			DD	BB	CC	Array		
DD	BB	vtable	0	0			0	0
		bb		1			1	8
	CC	vtable	1		0		2	16
		cc[0]			1	0	3	24
		cc[1]				1	4	28
		dd	2				5	32

The field indexes utilized in Gep$_{idx}$ are in an unflattened form. C++ adopts a distinct object model compared to Java, where a sub-object can be embedded within its containing object in C++, while in Java, only the reference/pointer to the sub-object is contained within the containing object.



Object		Field	t1	t2	t3	t4
DD	BB	vtable	BB's	DD's for BB		
		bb				
	CC	vtable	CC's			DD's for CC
		cc[0]				
		cc[1]				
		dd				
Constructor			DD( )			
			BB( )	CC( )		

(a) The original semantics

Object		Field	t1	t2	t3	t4
DD	BB	vtable	BB's	CC's	DD's for BB	DD's for CC
		bb				
	CC	vtable				
		cc[0]				
		cc[1]				
		dd				
Constructor			DD( )			
			BB( )	CC( )		

(b) The modeled semantics in SVF

```
01 struct BB {
02     long *bb;
03     virtual void gg() { }
04 };
05 struct CC {
06     int cc[2];
07     virtual void hh() { }
08 };
09 struct DD: public BB, public CC {
10     long dd;
11     virtual void ff() {  }
12 };
13 void test1(BB *bptr) {
14     bptr->gg(); //(bptr->vtable)[0]
15 }
16 void test2(CC *cptr) {
17     cptr->hh(); //(cptr->vtable)[0]
18 }
19 void test3(DD *dptr) {
20     dptr->ff(); //(dptr->vtable)[1]
21 }
22 typedef void (DD::*MptrTy)();
23 void test4(DD *dptr, MptrTy mfptr) {
24     (dptr->*mfptr)();
25 }
26 int main() {
27     DD dobj;
28     BB *bptr = static_cast<BB*>(&dobj);
29     test1(bptr);
30     CC *cptr = static_cast<CC*>(&dobj);
31     test2(cptr);
32     test3(&dobj);
33     test4(&dobj, &DD::ff);
34     return 0;
35 }
```

```
36 %BB = type { i32 (...)**, i64* }

37 %CC = type { i32 (...)**, [2 x i32] }

38 %DD = type { %BB, %CC, i64 }

     // DD’s vtable
39 @_ZTV2DD = { [4 x i8*], [3 x i8*] } {
     // DD’s vtable for BB
40   [null,           // top_offset
41    @_ZTI2DD,       // Type Info of DD
42    @_ZN2BB2ggEv,   // vtable[0] for BB::gg()
43    @_ZN2DD2ffEv    // vtable[1] for DD::ff()
44   ],
     // DD’s vtable for CC
45   [inttoptr (-16), // top_offset
46    @_ZTI2DD,       // Type Info of DD
47    @_ZN2CC2hhEv    // vtable[0] for CC::hh()
48   ]
49 }

     // BB’s vtable
50 @_ZTV2BB = { [3 x i8*] } {
51   [null,           // top_offset
52    @_ZTI2BB,       // Type Info of BB
53    @_ZN2BB2ggEv    // vtable[0] for BB::gg()
54   ]
55 }
     // CC’s vtable
56 @_ZTV2CC = { [3 x i8*] } {
57   [null,           // top_offset
58    @_ZTI2CC,       // Type Info of CC
59    @_ZN2CC2hhEv    // vtable[0] for CC::hh()
60   ]
61 }
```

Function	bptr->gg()		cptr->hh()		dptr->ff()		(dptr->*mfptr)()	
	SVF	TIPS	SVF	TIPS	SVF	TIPS	SVF	TIPS
BB::gg()		✓					✓	
CC::hh()	✓		✓	✓	✓		✓	
DD::ff()						✓	✓	✓
Sound	N	Y	Y	Y	N	Y	Y	Y
Precise	N	Y	Y	Y	N	Y	N	Y

(c) Target functions resolved by SVF and TIPS

Fig. 3. Target functions resolved more precisely by TIPS than SVF in multiple inheritance.

To illustrate the contrast between unflattened and flattened indexes, let us consider a DD object created at line 27 in Figure 3, with its layered object model outlined in Table 2. The DD object contains three elements (line 38): a BB sub-object, a CC sub-object, and an integer. Their unflattened indexes range from 0 to 2. Similarly, the BB sub-object (line 36) has two of its own elements (a virtual table pointer and bb), and the CC sub-object (line 37) also possesses two elements (a virtual table pointer and cc[2]), where the array cc takes up two elements (cc[0] and cc[1]). To facilitate field-index-based field-sensitive accesses within the containing DD object, all these unflattened indexes must be transformed into flattened indexes. In total, there are six flattened indexes, ranging from 0 to 5. The corresponding byte offsets for these flattened indexes are also listed in Table 2. On a 64-bit system, a BB object (line 36) occupies 16 bytes. If the size of an int is 4, then the size of a CC object (line 37) is also 16 bytes. Consequently, the size of a DD object is 40 bytes.

Given that GEP$_c$ introduces byte-level offsets, TIPS naturally incorporates a recursive algorithm to convert a byte offset within an object into its corresponding flattened index.

Table 3 outlines the principal rules Tips employs for managing integer-pointer value flows. The rules [PtrToInt] and [IntToPtr] are pivotal for tracing non-virtual member function pointers, [ConstToInt] specifically addresses the tracking of virtual member function pointers, while [IntToInt] is versatile, applicable to both non-virtual and virtual function pointers alike. Crucially, the Gep$_k$ undergoes decomposition into multiple Gep$_c$ instances, a strategic move to achieve field-sensitivity, thus significantly refining the precision in resolving member function calls.

Table 3. Tips's key rules for tracking integer-pointer value flows.

$$[\textsc{IntToPtr}]\ \frac{\text{p = i, i} \in \text{Integer, p} \in \text{Pointer}}{\text{PointsTo(i)} \subseteq \text{PointsTo(p)}} \quad [\textsc{PtrToInt}]\ \frac{\text{i = p, i} \in \text{Integer, p} \in \text{Pointer}}{\text{PointsTo(p)} \subseteq \text{PointsTo(i)}} \quad [\textsc{IntToInt}]\ \frac{\text{i = j, i} \in \text{Integer, j} \in \text{Integer}}{\text{PointsTo(j)} \subseteq \text{PointsTo(i)}}$$

$$[\textsc{ConstToInt}]\ \frac{\text{i = c, i} \in \text{Integer, c} \in \text{ConstInt}}{\text{c} \in \text{PointsTo(i)}} \quad [\textsc{Gep}_k\ ]\ \frac{\begin{array}{c}\text{p = gep vtable, k}\\ \text{p} \in \text{Pointer, vtable} \in \text{Pointer, k} \in \text{Integer, c} \in \text{PointsTo(k), c} \in \text{ConstInt}\end{array}}{\text{p = gep vtable, c}}$$

3.2 Byte-Level Pointer Adjustments in Multiple Inheritance

C++'s multiple inheritance allows classes to inherit from several base classes, complicating object memory layouts and posing challenges for pointer analysis, particularly during base-to-derived (or reverse) type casts that require Gep$_c$-guided pointer adjustments. Existing frameworks like SVF fail to account for these adjustments. SVF, despite its flow-sensitive analysis demonstrated in Figure 3, incorrectly identifies `CC::hh()` calls at lines 14 and 20, besides the correct identification at line 17. Below we explore the semantics of C++ multiple inheritance, how disregarding byte-level pointer adjustments affects call resolution precision, and Tips's strategy for addressing these inaccuracies.

As shown in Figure 3, class `DD` inherits from two base classes: `BB` and `CC`, resulting in a `DD` object containing two virtual table pointers (line 38). Object `dobj` is created at line 27 and initialized using constructors for `BB`, `CC`, and `DD`. Figure 3(a) summarizes the initialization sequence of its two virtual table pointers when `dobj` is created via `DD`'s constructor `DD()`. At `t1`, the constructor `BB()` is called to initialize the `BB` sub-object, setting its virtual table pointer to `BB`'s virtual table (line 53). Then, at `t2`, the underlying object pointer is adjusted to the `CC` sub-object, and the constructor `CC()` initializes it, setting its virtual table pointer to `CC`'s virtual table (line 59). Once both sub-objects are ready, `DD()` proceeds to initialize its own data. At `t3`, the virtual table pointer in the `BB` sub-object is adjusted to point to `DD`'s virtual table for `BB` (line 42). Finally, at `t4`, the virtual table pointer in the `CC` sub-object is adjusted to point to `DD`'s virtual table for `CC` (line 47).

Upon returning to the `main()` function, the `DD` object represented by `dobj` undergoes explicit casting into a `BB` object at line 28 and a `CC` object at line 30. Since the `BB` sub-object of `dobj` begins at byte offset 0, no pointer adjustment is required at line 28. However, `dobj`'s `CC` sub-object is located at byte offset 16, prompting LLVM to implicitly generate a Gep$_c$ instruction to adjust the object pointer, enabling `cptr` at line 30 to point to the `CC` sub-object. Concerning the parameters of `test1()`, `test2()`, and `test3()`, both `bptr` (line 13) and `dptr` (line 19) point to byte offset 0 of the `DD` object, while `cptr` (line 16) points to the `CC` sub-object. Thus, the three virtual calls at lines 14, 17, and 20 are anticipated to indirectly invoke `BB::gg()`, `CC::hh()`, and `DD::ff()`, respectively.

However, existing pointer analysis techniques [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, Yu et al., 2010], including SVF [Sui and Xue, 2024], do not precisely model this complex semantics. For example, SVF's approximation is illustrated in Figure 3(b). SVF neglects Gep$_c$, resulting in no pointer adjustments being made. Consequently, only the virtual table pointer of the `BB` sub-object is updated from `t1` to `t4`. Furthermore, due to SVF's flow-sensitive nature with strong updates [Sui and Xue, 2016a], only the virtual table for `CC` remains after `t4`, with the previous three being killed. Thus, SVF reports that `CC::hh()` is called at line 17

```
1 // container to sub-object
2 %p2 = bitcast %DD* dptr to i8*
3 %p3 = getelementptr i8* %p2, i64 16
4 cptr = bitcast i8* %p3 to %CC*

5 // sub-object to container
6 %p5 = bitcast %CC* cptr to i8*
7 %p6 = getelementptr i8* %p5, i64 -16
8 %p7 = bitcast i8* %p6 to %DD*
```

Pointer	Object	GEPc		PointsTo	Rule
		Byte Offset	Flattened Field Index		
dptr	dobj			(dobj, 0, 0)	ADDROF
%p2	dobj			(dobj, 0, 0)	COPY
%p3	**dobj**	**16**	**2**	**(dobj, 16, 2)**	**GEPc**
cptr	dobj			(dobj, 16, 2)	COPY
%p5	dobj			(dobj, 16, 2)	COPY
%p6	**dobj**	**-16**	**-2**	**(dobj, 0, 0)**	**GEPc**
%p7	dobj			(dobj, 0, 0)	COPY

Fig. 4. TIPS's pointer analysis for byte-level pointer adjustments in C++ multiple inheritance.

(`cptr->hh()`) correctly, as expected, but also at line 14 (`bptr->gg()`) and line 20 (`dptr->ff()`) in Figure 3 erroneously. On the other hand, if SVF is configured to run in flow-insensitively, all four virtual tables in Figure 3(b) are conflated, resulting in sound but imprecise results.

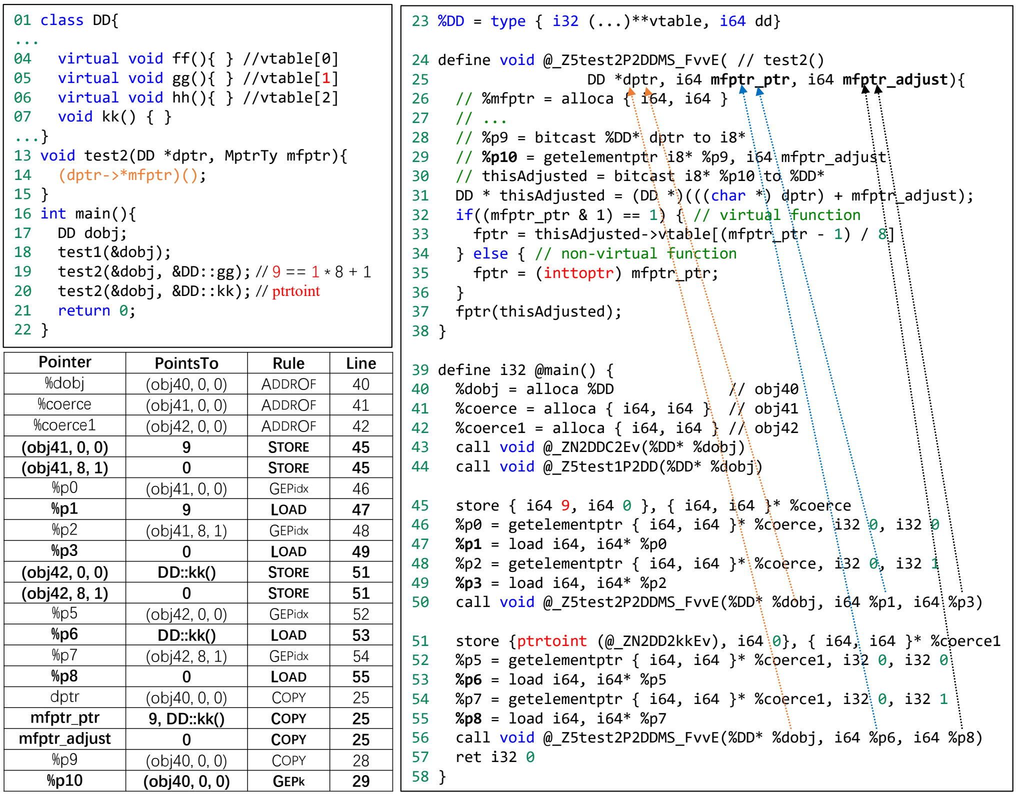
The LLVM IR instructions used to perform byte-level pointer adjustments for casting a pointer from a `DD` object to a `CC` object, and vice versa, are detailed in Figure 4. These instructions consist of two `getelementptr` instructions (lines 3 and 7), which are modeled by GEP$_c$ (Table 1), and four `bitcast` instructions (lines 2, 4, 6, and 8) used for type casting, which are modeled by the COPY rule. For instance, let us assume that `dptr` (line 2) is initially set to point to a `DD` object, as defined in Table 2. To cast it into a pointer to a `CC` object, the byte offset of 16 at line 3 is converted into a flattened index of 2. This means that `%p3` now points to the field represented as a tuple `(dobj, 16, 2)`, specifically, the virtual table pointer (referred to as `vtable`) within the `CC` sub-object (as specified in Table 2). Conversely, when dealing with the byte offset of −16 at line 7, it needs to be converted into a flattened index of −2. As a result, `%p6` points to the field `(dobj,` $16-16$`,` $2-2$`)`, which simplifies to `(dobj, 0, 0)`. When applying GEP$_{idx}$ in the classic field-index-based pointer analysis (Table 1), all such converted field indexes are accumulated throughout the pointer analysis process. Similarly, TIPS also handles the conversion of byte-level pointer adjustments within the C-style macro `container_of()` [Koschel et al., 2023] into flattened field indexes. For the sake of brevity, we have omitted a detailed discussion on the workings of `container_of()`.

When member function pointers are involved at line 33 in Figure 3, GEP$_k$ is applied to analyze the member function call at line 24, adjusting object and virtual table pointers for proper analysis. However, SVF's field-insensitivity in this context, as discussed earlier, causes it to report all three virtual functions from the two virtual tables in the `DD` class as potential targets for the member function call (Figure 3(c)). While regressing to field-insensitivity is sound, it yields imprecise results, due to the imprecision in modeling the semantics of object initialization, as depicted in Figure 3(b). In contrast, TIPS's precise tracking of byte-level pointer adjustments in both GEP$_c$ and GEP$_k$ allows it to accurately identify `DD::ff()` as the sole target for the member function call (Figure 3(c)).

3.3 Analyzing C++ Member Function Pointers at the LLVM-IR Level

Let us explore the handling of C++ member function pointers at the LLVM-IR level, as illustrated in Figure 5, by utilizing our earlier motivating example from Figure 1. The LLVM-generated IR instructions are presented at lines 24–58. It is worth noting that while `test2()` (lines 13-15) in the source code has two parameters, its LLVM IR-level counterpart (lines 24-38) has three parameters. The member function pointer `mfptr` (line 13) is a fat pointer, which is dissected into two distinct integer parameters: `mfptr_ptr`, representing either a virtual table index or a non-virtual function address, and `mfptr_adjust`, accounting for object pointer adjustments (line 25).

In the `main()` function, the `alloca` instruction (`line 40`) is modeled by `AddrOf` (Table 1). It creates a virtual register, `%dobj`, pointing to a locally allocated `DD` object. At the source code level, `dobj` (`line 17`) is a local object. To obtain its address, `&dobj` (`lines 18-20`) corresponds to the virtual register `%dobj` in LLVM-IR. The `DD` object pointed to by `%dobj` is then initialized by



```
01 class DD{
...
04   virtual void ff(){ } //vtable[0]
05   virtual void gg(){ } //vtable[1]
06   virtual void hh(){ } //vtable[2]
07   void kk() { }
...}
13 void test2(DD *dptr, MptrTy mfptr){
14   (dptr->*mfptr)();
15 }
16 int main(){
17   DD dobj;
18   test1(&dobj);
19   test2(&dobj, &DD::gg); // 9 == 1 * 8 + 1
20   test2(&dobj, &DD::kk); // ptrtoint
21   return 0;
22 }
```

Pointer	PointsTo	Rule	Line
%dobj	(obj40, 0, 0)	AddrOf	40
%coerce	(obj41, 0, 0)	AddrOf	41
%coerce1	(obj42, 0, 0)	AddrOf	42
(obj41, 0, 0)	**9**	**Store**	**45**
(obj41, 8, 1)	**0**	**Store**	**45**
%p0	(obj41, 0, 0)	GEPidx	46
%p1	**9**	**Load**	**47**
%p2	(obj41, 8, 1)	GEPidx	48
%p3	**0**	**Load**	**49**
(obj42, 0, 0)	**DD::kk()**	**Store**	**51**
(obj42, 8, 1)	**0**	**Store**	**51**
%p5	(obj42, 0, 0)	GEPidx	52
%p6	**DD::kk()**	**Load**	**53**
%p7	(obj42, 8, 1)	GEPidx	54
%p8	**0**	**Load**	**55**
dptr	(obj40, 0, 0)	Copy	25
mfptr_ptr	**9, DD::kk()**	**Copy**	**25**
mfptr_adjust	**0**	**Copy**	**25**
%p9	(obj40, 0, 0)	Copy	28
%p10	**(obj40, 0, 0)**	**GEPk**	**29**

```
23 %DD = type { i32 (...)**vtable, i64 dd}

24 define void @_Z5test2P2DDMS_FvvE( // test2()
25                   DD *dptr, i64 mfptr_ptr, i64 mfptr_adjust){
26   // %mfptr = alloca { i64, i64 }
27   // ...
28   // %p9 = bitcast %DD* dptr to i8*
29   // %p10 = getelementptr i8* %p9, i64 mfptr_adjust
30   // thisAdjusted = bitcast i8* %p10 to %DD*
31   DD * thisAdjusted = (DD *)(((char *) dptr) + mfptr_adjust);
32   if((mfptr_ptr & 1) == 1) { // virtual function
33     fptr = thisAdjusted->vtable[(mfptr_ptr - 1) / 8]
34   } else { // non-virtual function
35     fptr = (inttoptr) mfptr_ptr;
36   }
37   fptr(thisAdjusted);
38 }

39 define i32 @main() {
40   %dobj = alloca %DD              // obj40
41   %coerce = alloca { i64, i64 }  // obj41
42   %coerce1 = alloca { i64, i64 } // obj42
43   call void @_ZN2DDC2Ev(%DD* %dobj)
44   call void @_Z5test1P2DD(%DD* %dobj)

45   store { i64 9, i64 0 }, { i64, i64 }* %coerce
46   %p0 = getelementptr { i64, i64 }* %coerce, i32 0, i32 0
47   %p1 = load i64, i64* %p0
48   %p2 = getelementptr { i64, i64 }* %coerce, i32 0, i32 1
49   %p3 = load i64, i64* %p2
50   call void @_Z5test2P2DDMS_FvvE(%DD* %dobj, i64 %p1, i64 %p3)

51   store {ptrtoint (@_ZN2DD2kkEv), i64 0}, { i64, i64 }* %coerce1
52   %p5 = getelementptr { i64, i64 }* %coerce1, i32 0, i32 0
53   %p6 = load i64, i64* %p5
54   %p7 = getelementptr { i64, i64 }* %coerce1, i32 0, i32 1
55   %p8 = load i64, i64* %p7
56   call void @_Z5test2P2DDMS_FvvE(%DD* %dobj, i64 %p6, i64 %p8)
57   ret i32 0
58 }
```

Fig. 5. Applying Tips to analyze the member function pointers at the LLVM-IR level in our example (Figure 1).

DD's constructor at line 43. At lines 19–20, two member function pointers, &DD::gg and &DD::kk, generate two local objects, coerce and coerce1 (lines 41–42) for storage. coerce is initialized (line 45) with 9 (the encoded virtual table index for DD::gg()) and 0 (a byte offset for the object pointer adjustment). At line 51, _ZN2DD2kkEv (i.e., &DD::kk) is cast into an integer and used to initialize coerce1. Since the operands in a store instruction are structs rather than pointers or integers, an LLVM pass is employed by Tips to normalize these instructions, ensuring they adhere to the inference rules governing pointer analysis in Section 3.1. Four instructions (lines 46–49) load two integers from coerce and pass them as arguments when calling test2() (line 50). The getelementptr instruction (line 46) is modeled by Gep$_{idx}$ (Table 1) in SVF [Sui and Xue, 2024] and Tips. However, prior work ignores the load instruction (line 47), resulting in a loss of data flow information. The code at lines 52–55 follows a similar pattern. By tracking integer-pointer value flows, Tips uses the Load and Store rules (Section 3.1) to model these load and store instructions, correctly propagating the points-to information that would otherwise be missed.

Figure 5 illustrates the points-to information for pointers identified by Tips, with objects allocated at lines 40-42 labeled as obj40, obj41, and obj42. An object field is denoted as (obj, btOffset, fldIdx), where btOffset is the byte offset and fldIdx is the flattened field index for object Obj. The **bold** rows highlight the points-to information uniquely identified by Tips, demonstrating its ability to manage integer-pointer value flows as detailed in Table 1.

In test2(), LLVM allocates a local object mfptr at line 26 and initializes it using the two parameters mfptr_ptr and mfptr_adjust, with the initialization instructions omitted. The process for making byte-level object pointer adjustments is simplified and presented at lines 28–30.

The getelementptr instruction at line 29 is governed by Gep_k (p = gep q, k) in TIPS, where mfptr_adjust is represented as k. By treating integers as pointers and propagating their points-to information, PointsTo(mfptr_adjust) = {0} holds at line 29. Here, Gep_k is converted into a Gep_c rule (p = gep q, c), where c is 0, indicating that no pointer adjustment is required. If c is non-zero, TIPS will adjust the object pointer at line 29 and convert the byte offset into a flattened field index (Table 2). The if statement at lines 32–36 (Figure 5) assesses whether mfptr_ptr is odd, corresponding to the orange part in Figure 2. As TIPS is not path-sensitive, PointsTo(mfptr_ptr) = {9, DD::kk()} at lines 32, 33, and 35. Similar to earlier pointer analysis approaches [Lhoták and Hendren, 2003, Sui and Xue, 2016b], using type information can refine imprecise points-to information. For instance, at line 33, only the constant integer 9 in PointsTo(mfptr_ptr) is treated as an encoded virtual table index. After decoding 9 to obtain the virtual table index 1, the virtual function address (&DD::gg) is loaded from DD's virtual table. By merging {DD::gg()} (line 33) with {9, DD::kk()} (i.e., PointsTo(mfptr_ptr) at line 35), we derive PointsTo(fptr) = {9, DD::kk(), DD::gg()} at line 37. Type information helps filter out the constant integer 9. Ultimately, TIPS identifies DD::gg() and DD::kk() as the targets for the member function call at line 14.

3.4 C++ Member Function Pointers in Virtual Inheritance

In C++, diamond inheritance occurs when two classes, BB and CC (lines 5–12 in Figure 6), inherit from a common base class, AA (lines 1–4), and a child class, DD (lines 13–17), inherits from both BB and CC. To ensure that only one instance of the AA sub-object exists within a DD object, virtual inheritance (lines 5 and 9) is introduced in C++. This adds complexity to the memory layout of C++ objects and poses significant challenges for pointer analysis of C++ member function pointers. In the context of the member function call at line 20, TIPS provides precise information by reporting that only DD::ff() is called. In contrast, SVF considers all functions in the virtual tables as possible target functions (a total of seven), leading to sound but imprecise results. To explore the intricate semantics associated with virtual inheritance in this C++ program, let us examine how TIPS utilizes GEP_c and GEP_k to handle the complexities introduced by C++'s virtual inheritance.

The LLVM-IR type definitions for AA, BB, CC, and DD are provided at lines 28–33. At the LLVM-IR level, both BB (line 29) and CC (line 30) contain a single instance of the AA sub-object. In addition, LLVM introduces two auxiliary classes, BB.base and CC.base, at lines 31 and 32. These auxiliary classes play an important role in DD's definition, ensuring that a DD object contains only one instance of the AA sub-object (line 33). In simpler terms, if BB and CC were directly used in DD's definition, a DD object would contain three instances of the AA sub-object, which is a situation we want to avoid.

The AA sub-object within a containing object is commonly referred to as a *vbase object*, and the distance from the beginning of the containing object to the vbase object is known as the *vbase offset*. C++ compilers store vbase offsets in virtual tables. For instance, both a DD object and its BB sub-object have vbase offsets of 40 bytes. In the case of the CC sub-object within a DD object (Figure 6(a)), it has a vbase offset of 24 (calculated as 40 − 16). Interestingly, an independent CC object (Figure 6(b)) has its vbase offset at 16, which differs from that of the CC sub-object within a DD object. Moreover, the CC sub-object is a part of dobj, and its virtual table differs from that of cobj. To address this problem, LLVM generates two constructors for CC: CC(CC *cptr) for initializing cobj (line 24) and CC(CC *cptr, char **vtt) for initializing the CC sub-object within DD. In this context, vtt is short for Virtual Table Table (VTT), which contains the virtual table pointers required to initialize the CC sub-object within a DD object. If there are other derived classes from class CC, they can reuse the constructor CC(CC *cptr, char **vtt) to initialize their CC sub-objects, provided that they supply their own VTTs. Similarly, C++ compilers also generate a constructor BB(BB *bptr, char **vtt) for initializing the BB sub-object within a DD object.

```
struct AA {
    char *aa;
    virtual void ff() { }
};
struct BB: public virtual AA {
    long *bb;
    virtual void gg() { }
};
struct CC: public virtual AA {
    int cc[2];
    virtual void hh() { }
};
struct DD: public BB, public CC{
    long dd;
    virtual void hh() { }
    virtual void ff() { }
};
typedef void (DD::*MptrTy)();
void test4(DD *dptr, MptrTy mfptr){
    (dptr->*mfptr)();
}
int main() {
    DD dobj;
    CC cobj;
    test4(&dobj, &DD::ff);
    return 0;
}

%AA = type { i32 (...)**, i8* }

%BB = type { i32 (...)**, i64*, %AA }

%CC = type { i32 (...)**, [2 x i32], %AA}

%BB.base = type { i32 (...)**, i64* }

%CC.base = type { i32 (...)**, [2 x i32]}

%DD = type {%BB.base, %CC.base, i64,%AA}
```

Object		Field	Byte Offset
DD	BB.base	vtable	0
		bb	8
	CC.base	vtable	16
		cc[0]	20
		cc[1]	24
		dd	32
	AA	vtable	40
		aa	48

(a) The DD object dobj

Object		Field	Byte Offset
CC		vtable	0
		cc[0]	8
		cc[1]	12
	AA	vtable	16
		aa	24

(b) The CC object cobj

Function	(dptr->*mfptr)()	
	SVF	TIPS
AA::ff()	✓	
BB::gg()	✓	
CC::hh()	✓	
DD::hh()	✓	
DD::ff()	✓	✓
non-virtual thunk to DD::hh()	✓	
virtual thunk to DD::ff()	✓	
#Functions Reported	7	1
Sound	Y	Y
Precise	N	Y

(c) Target functions reported

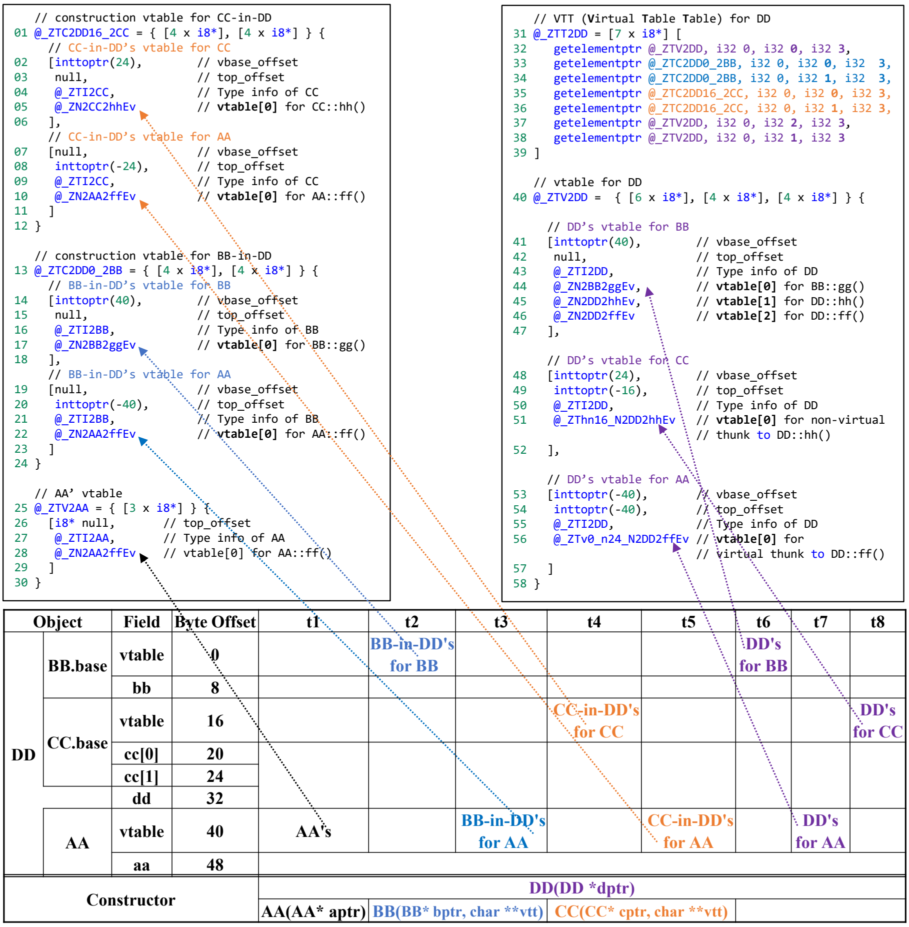
Fig. 6. Precision gains of Tips over SVF in analyzing C++ member function pointers in virtual inheritance.

Figure 7 gives a list of seven virtual tables and the VTT for DD (lines 31–39) used to initialize dobj when invoking DD(&dobj). The constructor DD(DD *dptr) sequentially calls AA(AA *aptr), BB(BB *bptr, char **vtt), and CC(CC *cptr, char **vtt) to initialize the vbase object, the BB sub-object, and the CC sub-object, respectively. The virtual table for AA is provided at lines 25–30. In addition, C++ compilers generate virtual tables for CC-in-DD (lines 1–12) and BB-in-DD (lines 13–24). DD's virtual table, outlined at lines 40–58, is divided into three sub-virtual tables: one for DD's relationship with BB (lines 41–47), one for CC (lines 48–52), and one for AA (lines 53–57).

At t1, we apply Gep$_c$ to acquire the address of the AA sub-object within a DD object and subsequently call AA(AA *aptr). We begin by setting the virtual table pointer of the vbase object to point to AA's virtual table (line 28). Since the BB sub-object is located at byte offset 0, no adjustment is required for the parameter bptr in BB(BB *bptr, char **vtt). Moving to t2, the virtual table pointer at byte offset 0 now points to BB-in-DD's virtual table for BB. To obtain the address of its base object, we load the vbase offset of 40 (by utilizing the Load rule) and then apply Gep$_k$ to adjust the object pointer, resulting in (char *) bptr + 40. Finally, at t3, we update the vbase object's virtual table pointer to point to BB-in-DD's virtual table for AA (line 22).

After initializing the BB sub-object, control flow returns to DD(DD *dptr). Here, we use Gep$_c$ to obtain the CC sub-object's address and pass it to cptr in CC(CC *cptr, char **vtt). At t4, we set the virtual table pointer for the CC sub-object (at byte offset 16) to point to CC-in-DD's virtual table for CC (line 5). We then load its vbase offset of 24 (line 2) by using the Load rule and apply the Gep$_k$ rule to obtain the vbase object's address. Finally, we proceed to set the virtual table pointer for the AA sub-object to point to CC-in-DD's virtual table for AA at t5 (line 10).

Once all the sub-objects are initialized, DD(DD *dptr) proceeds to set its three virtual table pointers. At t6, it assigns them to point to DD's virtual table for BB (line 44). This is followed by



```
   // construction vtable for CC-in-DD
01 @_ZTC2DD16_2CC = { [4 x i8*], [4 x i8*] } {
     // CC-in-DD’s vtable for CC
02   [inttoptr(24),         // vbase_offset
03    null,                 // top_offset
04    @_ZTI2CC,             // Type info of CC
05    @_ZN2CC2hhEv          // vtable[0] for CC::hh()
06   ],
     // CC-in-DD’s vtable for AA
07   [null,                 // vbase_offset
08    inttoptr(-24),        // top_offset
09    @_ZTI2CC,             // Type info of CC
10    @_ZN2AA2ffEv          // vtable[0] for AA::ff()
11   ]
12 }

   // construction vtable for BB-in-DD
13 @_ZTC2DD0_2BB = { [4 x i8*], [4 x i8*] } {
     // BB-in-DD’s vtable for BB
14   [inttoptr(40),         // vbase_offset
15    null,                 // top_offset
16    @_ZTI2BB,             // Type info of BB
17    @_ZN2BB2ggEv          // vtable[0] for BB::gg()
18   ],
     // BB-in-DD’s vtable for AA
19   [null,                 // vbase_offset
20    inttoptr(-40),        // top_offset
21    @_ZTI2BB,             // Type info of BB
22    @_ZN2AA2ffEv          // vtable[0] for AA::ff()
23   ]
24 }

   // AA’ vtable
25 @_ZTV2AA = { [3 x i8*] } {
26   [i8* null,        // top_offset
27    @_ZTI2AA,        // Type info of AA
28    @_ZN2AA2ffEv     // vtable[0] for AA::ff()
29   ]
30 }
```

```
   // VTT (Virtual Table Table) for DD
31 @_ZTT2DD = [7 x i8*] [
32    getelementptr @_ZTV2DD, i32 0, i32 0, i32 3,
33    getelementptr @_ZTC2DD0_2BB, i32 0, i32 0, i32  3,
34    getelementptr @_ZTC2DD0_2BB, i32 0, i32 1, i32  3,
35    getelementptr @_ZTC2DD16_2CC, i32 0, i32 0, i32 3,
36    getelementptr @_ZTC2DD16_2CC, i32 0, i32 1, i32 3,
37    getelementptr @_ZTV2DD, i32 0, i32 2, i32 3,
38    getelementptr @_ZTV2DD, i32 0, i32 1, i32 3
39 ]

   // vtable for DD
40 @_ZTV2DD =  { [6 x i8*], [4 x i8*], [4 x i8*] } {

     // DD’s vtable for BB
41   [inttoptr(40),         // vbase_offset
42    null,                 // top_offset
43    @_ZTI2DD,             // Type info of DD
44    @_ZN2BB2ggEv,         // vtable[0] for BB::gg()
45    @_ZN2DD2hhEv,         // vtable[1] for DD::hh()
46    @_ZN2DD2ffEv          // vtable[2] for DD::ff()
47   ],

     // DD’s vtable for CC
48   [inttoptr(24),         // vbase_offset
49    inttoptr(-16),        // top_offset
50    @_ZTI2DD,             // Type info of DD
51    @_ZThn16_N2DD2hhEv    // vtable[0] for non-virtual
                            // thunk to DD::hh()
52   ],

     // DD’s vtable for AA
53   [inttoptr(-40),        // vbase_offset
54    inttoptr(-40),        // top_offset
55    @_ZTI2DD,             // Type info of DD
56    @_ZTv0_n24_N2DD2ffEv  // vtable[0] for
                            // virtual thunk to DD::ff()
57   ]
58 }
```

Object		Field	Byte Offset	t1	t2	t3	t4	t5	t6	t7	t8
DD	BB.base	vtable	0		BB-in-DD's for BB				DD's for BB		
		bb	8								
	CC.base	vtable	16				CC-in-DD's for CC				DD's for CC
		cc[0]	20								
		cc[1]	24								
		dd	32								
	AA	vtable	40	AA's		BB-in-DD's for AA		CC-in-DD's for AA		DD's for AA	
		aa	48								
Constructor				DD(DD *dptr)							
				AA(AA* aptr)	BB(BB* bptr, char **vtt)		CC(CC* cptr, char **vtt)				

Fig. 7. Initialization sequence for the three virtual table pointers in a DD Object in Figure 6.

t7, where the virtual table pointer for AA (line 56) is configured, and t8, where the pointer for CC (line 51) is established. In two time steps, t7 and t8, GEP_c is once again applied to perform pointer adjustments for locating the two virtual table pointers, located at byte offsets 40 and 16.

Pointer adjustments are also relevant within virtual and non-virtual thunk functions, which are automatically introduced by C++ compilers. In the case of class DD, DD::ff() overrides AA::ff() (line 16 in Figure 6), where AA serves as a virtual base class, a virtual thunk function (line 56 in Figure 7) is generated. This function adjusts a AA * pointer by adding the virtual base offset (−40) stored at line 53. This adjustment allows us to obtain a DD * pointer, facilitating access to the data within a DD object. In addition, a non-virtual thunk function is introduced for DD::hh() (line 51) because DD::hh() is found to also override CC::hh() (line 15 in Figure 6). It is worth noting that while DD::hh() is a virtual function, CC is not a virtual base class. Consequently, GEP_c is employed within the non-virtual thunk function to perform necessary pointer adjustments.

The object pointer adjustments made during the execution of the constructor DD(DD *dptr) involve the application of the GEP_c and GEP_k rules, which are summarized in Table 4. Specifically, within the constructor, GEP_c is utilized in the type casts from DD * to CC * and AA *. However, in the contexts of BB(BB *bptr, char **vtt) and CC(CC *cptr, char **vtt), GEP_k is applied for the purpose of making pointer adjustments to access the vbase object.

Table 4. Object pointer adjustments in the initialization of a DD object, named dobj, in Figure 6.

==>	BB *bptr	CC *cptr	AA *aptr	Inside Constructor
DD *dptr	Casting but No Pointer Adjustment	Gep$_c$	Gep$_c$	DD(DD *dptr)
BB *bptr			Gep$_k$	BB(BB *bptr, char **vtt)
CC *cptr			Gep$_k$	CC(CC *cptr, char **vtt)
AA *aptr				AA(AA *aptr)

By converting one application of Gep$_k$ into multiple applications of Gep$_c$, Tips can eliminate the imprecision resulting from the field-insensitive application of Gep$_k$ in the context of virtual inheritance. When constructing constraint graphs for analyzed programs, null pointers (e.g., line 7 in Figure 7) within virtual tables should be represented as `inttoptr(0)` rather than `nullptr`. This allows Tips to treat null as a virtual base offset of 0 and propagate it as points-to information for integer variables. Thanks to enhanced field-sensitivity, Tips can differentiate between the three virtual table pointers within a DD object in a flow- and context-sensitive pointer analysis. Ultimately, as illustrated in Figure 6(c), Tips offers more precise analysis of C++ member function pointers in virtual inheritance compared to the state of the art [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, 2024, Yu et al., 2010], as explained above.

4 TIPS: DESIGN AND IMPLEMENTATION

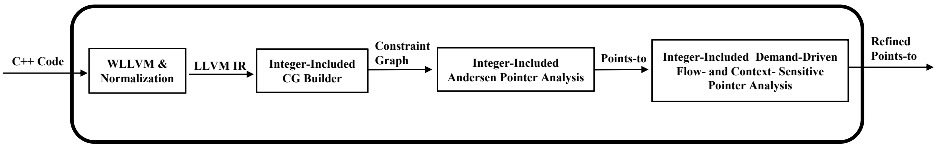

C++ Code
WLLVM & Normalization
LLVM IR
Integer-Included CG Builder
Constraint Graph
Integer-Included Andersen Pointer Analysis
Points-to
Integer-Included Demand-Driven Flow- and Context- Sensitive Pointer Analysis
Refined Points-to

Fig. 8. The workflow of Tips.

We developed Tips as an extension to LLVM 14.0, built atop SVF [Sui and Xue, 2024], shown in Figure 8. C++ source files are compiled using WLLVM [Ravitch, 2024], which generates whole-program LLVM bitcode files. Tips features an LLVM `ModulePass` [LLVM, 2024] that normalizes LLVM instructions like `load`, `store`, and `extractvalue` [LLVM, 2024], aligning them with pointer analysis rules outlined in Section 3.1. For instance, a `store` instruction "`store val, ptr`", where `val` is a `ConstantStruct` (`i64, i64`), is transformed to apply Gep twice to determine the structure's two fields pointed by `ptr`, and Store twice to store the two integers from `val` into these fields.

After normalizing LLVM instructions, we create an integer-inclusive constraint graph based on the pointer analysis inference rules from Table 1. We employ a modified version of Andersen's inclusion-based pointer analysis [Andersen, 1994] as a pre-analysis to obtain the initial points-to information for integers and pointers required for building such a constraint graph. This pre-analysis forms the basis for refining points-to information using our integer-inclusive, demand-driven, flow- and context-sensitive pointer analysis in Tips. Additionally, we leverage the Spare Value Flow Graphs (SVFGs) from the open-source SVF [Sui and Xue, 2016b, 2024] to support flow-sensitivity. Our context-sensitive analysis relies on call-sites, a common approach in C/C++ pointer analysis [Jeon and Oh, 2022, Li et al., 2023, Oh et al., 2014, Yu et al., 2010].

5 EVALUATION

In Section 5.1, we introduce a micro-benchmark suite with ground truth data, highlighting Tips's precision gains over SVF in analyzing C++ member function pointers across diverse inheritance scenarios. In Section 5.2, we evaluate Tips's precision and efficiency trade-offs, showing its ability to accurately identify critical C++ member function call targets, including thread entry functions, while maintaining acceptable analysis overhead in fourteen C++ programs.

Table 5. Precision achieved by Tips and SVF in resolving member function calls in a micro-benchmark suite (where VF and NVF stand for Virtual Functions and Non-Virtual Functions, respectively).

Test Cases	#Number of Target Functions Reported by Pointer Analysis for the Member Function Pointer in Each Test Case																	
	Single Inheritance						Multiple Inheritance						Virtual Inheritance					
	VF Only		NVF Only		Both		VF Only		NVF Only		Both		VF Only		NVF Only		Both	
	SVF	Tips	SVF	Tips	SVF	Tips	SVF	Tips	SVF	Tips	SVF	Tips	SVF	Tips	SVF	Tips	SVF	Tips
#F=10, #T = 2	20	2	0	2	20	2	51	2	0	2	40	2	68	2	0	2	65	2
#F=10, #T = 4	23	4	0	4	20	4	52	4	0	4	43	4	65	4	0	4	61	4
#F=10, #T = 8	27	8	0	8	21	8	44	8	0	8	41	8	62	8	0	8	62	8
#F=20, #T = 2	46	2	0	2	58	2	66	2	0	2	88	2	138	2	0	2	144	2
#F=20, #T = 4	49	4	0	4	57	4	82	4	0	4	81	4	121	4	0	4	136	4
#F=20, #T = 8	54	8	0	8	54	8	95	8	0	8	99	8	85	8	0	8	108	8
#F=30, #T = 2	64	2	0	2	60	2	104	2	0	2	105	2	169	2	0	2	187	2
#F=30, #T = 4	64	4	0	4	82	4	164	4	0	4	159	4	219	4	0	4	176	4
#F=30, #T = 8	72	8	0	8	73	8	153	8	0	8	132	8	205	8	0	8	181	8
Sound	Y	Y	N	Y	N	Y	Y	Y	N	Y	N	Y	Y	Y	N	Y	N	Y
Precise	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y

Our micro-benchmark suite includes 81 test cases, encompassing various scenarios involving different quantities of virtual and non-virtual member functions, spanning single, multiple, and virtual inheritance contexts. We also analyzed all seven C++ benchmarks from SPEC CPU 2006, focusing on three major ones: `447.dealii`, `453.povray`, and `483.xalan`, which use C++ member function pointers. Due to the lack of support for placement new in C++—a key feature in `483.xalan` discussed in Section 5.3—we excluded `483.xalan` from our analysis. We chose `447.dealii` (181,847 lines) and `453.povray` (155,163 lines) for their size and relevance to real-world inheritance scenarios (Section 5.2). In addition, we included 12 open-source C++ applications from Qt [Company, 2024], LLVM [Community, 2024b], Ninja [Community, 2024c], and GoogleTest [Community, 2024a], with nine (bolded in Table 7) having larger LLVM-IR bitcode files than `453.povray`. This selection helps extensively evaluate Tips's precision and efficiency against SVF in analyzing large C++ programs.

All C++ programs were compiled using WLLVM [Ravitch, 2024] with the "`-O0`" option, widely employed for static analysis purposes. Both Tips and SVF conducted their demand-driven, flow-, and context-sensitive pointer analyses with identical configurations ("`-flow-bg=100000 -cxt-bg=100000 -max-cxt=2`"). Our analysis environment consisted of a 3.50 GHz Intel Xeon E5 CPU, equipped with 512 GB of memory, and operated on a 64-bit Ubuntu 20.04 OS.

In our evaluation, we aim to answer two key research questions (RQs):

- **RQ1.** Can Tips resolve C++ member function pointers in various inheritance scenarios in our micro-benchmark suite more effectively than SVF?
- **RQ2.** Does Tips achieve precision gains over SVF in resolving C++ member function pointers at some acceptable analysis overheads in large C++ programs?

5.1 RQ1: Precision Improvements

In Table 5, our micro-benchmark suite comprises a total of 81 test cases categorized by single, multiple, or virtual inheritance. Each test case contains one member function call where various class member functions are invoked via a member function pointer.

Starting with single inheritance, we have 27 test cases, each involving two classes. These test cases fall into three sub-categories: classes with only virtual functions, classes with only non-virtual functions, and a mix of both. For each sub-category, we generate 9 test cases, introducing variations in the number of defined functions ($\#F \in \{10, 20, 30\}$) within each class and determining their overriding relationships randomly. Furthermore, we introduce variations in the number of member functions (i.e., targets) invoked via a member function pointer ($\#T \in \{2, 4, 8\}$).

In our motivating example presented in Figure 1, we demonstrated that Tips surpasses SVF due to SVF's limitations in handling virtual and non-virtual function calls. SVF's field-insensitive handling

Table 6. Efficiency of Tips and SVF in resolving member function calls in `447.dealii` and `453.povray`.

Analysis Stage	Metrics	`447.dealii`			`453.povray`		
		SVF	Tips	$\frac{\text{Tips}}{\text{SVF}}$	SVF	Tips	$\frac{\text{Tips}}{\text{SVF}}$
Pre-Analysis	AnalysisTime (Secs)	51.254	205.291	4.005	20.656	79.265	3.837
	#Addr	44,105	44,639	1.012	7,322	9,386	1.282
	#Copy	285,171	448,333	1.572	48,747	100,997	2.072
	#Gep	190,029	190,127	1.001	119,890	119,950	1.001
	#Load	20,800	30,151	1.450	13,033	19,597	1.504
	#Store	24,155	86,033	3.562	9,829	73,890	7.518
Demand-Driven	#SVFGNode	997,730	1,312,816	1.316	311,044	509,366	1.638
	#SVFGEdge	1,363,076	1,904,031	1.397	446,893	815,474	1.825
	AvgTimePerQuery (Sec)	2.645	2.981	1.127	0	3.469	NA
	AvgPtsSize	324	1.250	0.004	0	7	NA

of Gep$_k$ leads to sound but imprecise results for virtual function call targets. Simultaneously, SVF's neglect of integer-pointer value flows results in unsound results for non-virtual function call targets. When a test case exclusively contains virtual functions, SVF lists all virtual functions in a class's virtual table as possible targets for a member function call, resulting in imprecision. Conversely, in cases with only non-virtual functions, SVF becomes unsound, missing all target functions. Moreover, when a test case combines both virtual and non-virtual functions, SVF incorrectly identifies all virtual functions as targets for member function calls that should invoke non-virtual functions. In contrast, Tips maintains both soundness and precision across all 27 test cases.

When transitioning to multiple and virtual inheritance, we observe similar patterns as in single inheritance. We analyze 27 test cases for each inheritance scenario: three classes per test case for multiple inheritance (Figure 3) and four classes per test case for virtual inheritance (Figure 6). These test cases are generated similarly to single inheritance, with variations in the number of defined functions ($\#\mathsf{F} \in \{10, 20, 30\}$) within each class and random determination of overriding relationships. The number of functions invoked via a member function pointer also varies ($\#\mathsf{T} \in \{2, 4, 8\}$).

In tests with only virtual functions, SVF becomes imprecise (yet sound) compared to single-inheritance scenarios due to its field-insensitive Gep$_k$, exacerbated by multiple virtual tables (Figures 3 and 7). When only non-virtual functions are involved, SVF fails to identify any targets, similar to its performance in single-inheritance scenarios. In mixed tests, SVF wrongly identifies all virtual functions as targets, ignoring intended non-virtual calls. Conversely, Tips consistently achieves soundness and precision in all 54 test cases.

5.2 RQ2: Precision and Efficiency Tradeoffs

5.2.1 SPEC CPU 2006. Table 6 shows Tips's precision improvement over SVF in analyzing `447.dealii` and `453.povray`, despite longer pre-analysis phases due to its tracking of pointer and integer value flows. The pre-analysis for `447.dealii` takes 205.291 seconds (4.005× longer) and for `453.povray` 79.265 seconds (3.837× longer), resulting in larger constraint graphs with more AddrOf, Copy, Store, Load, and Gep edges. This increase in graph size is also due to instruction normalization (Figure 8). Thus, Tips's SVFGs have more nodes than SVF's, by 1.316× in `447.dealii` and 1.638× in `453.povray`, which are acceptable overheads given the precision gains.

Both Tips and SVF perform points-to queries for the member function pointers in these two programs on-demand using the SVFGs constructed during pre-analysis.

In the case of `447.dealii`, Tips reports an average of only 1.250 target functions per member function pointer, a significant reduction compared to SVF's 324 targets. The average query time (`AvgTimePerQuery`) are similar, with Tips taking 2.981 seconds and SVF 2.645 seconds. For `453.povray`, SVF fails to report any target functions, rendering it unsound. SVF's pre-analysis sets all member function pointers to null, preventing on-demand refinement of points-to information. In contrast, Tips reports an `AvgPtsSize` of 7 and an `AvgTimePerQuery` of 3.469 seconds.

Table 7. Efficiency of Tips and SVF in resolving member function calls in 12 open-source C++ programs, where nine of which (with larger LLVM-IRs to be analyzed than `453.povray`) are marked in bold.

Open-Source C++ Programs		Pre-Analysis Time		#SVFGNode		#SVFGEdge		AvgTimePerQuery		AvgPtsSize	
		SVF	Tips	SVF	Tips	SVF	Tips	SVF	Tips	SVF	Tips
LLVM	**llvm-config**	70.367	183.597	585,021	821,341	734,221	1,166,045	9.103	10.514	416.000	4.000
	llvm-ml	1,163.700	4,180.240	3,244,376	5,498,863	3,729,991	8,777,931	0.467	0.540	104.615	1.231
	FileCheck	119.932	331.657	649,342	944,214	793,082	1,333,502	2.014	1.990	183.000	2.667
	llvm-mt	64.572	208.485	550,636	888,408	662,091	1,259,724	2.318	1.763	427.000	4.000
Qt	**cmake_automoc_parser**	60.281	162.899	1,120,003	1,923,808	1,345,662	3,181,915	0.003	0.017	0.000	1.000
	moc	73.235	198.442	1,373,555	2,236,289	1,674,963	3,646,680	0.003	0.018	0.000	1.000
	tracegen	60.094	154.167	1,059,216	1,779,135	1,262,519	2,977,068	0.003	0.012	0.000	1.000
	tracepointgen	54.701	142.359	1,023,327	1,737,779	1,210,535	2,906,594	0.002	0.011	0.000	1.000
Ninja/GoogleTest	ninja	7.644	8.707	134,104	174,825	166,363	236,430	0.775	1.117	40.500	11.500
	ninja_test	488.123	817.214	890,832	1,371,169	1,194,944	2,148,449	0.582	0.584	519.800	159.300
	build_log_perftest	5.448	5.522	107,651	133,850	133,300	174,170	0.008	0.013	0.000	1.000
	manifest_parser_perftest	5.651	5.772	107,547	134,088	132,725	174,183	0.008	0.013	0.000	1.000

We manually inspected the source code of the two large benchmarks, `447.dealii` and `453.povray`, and confirmed that all address-taken member functions are covered by Tips.

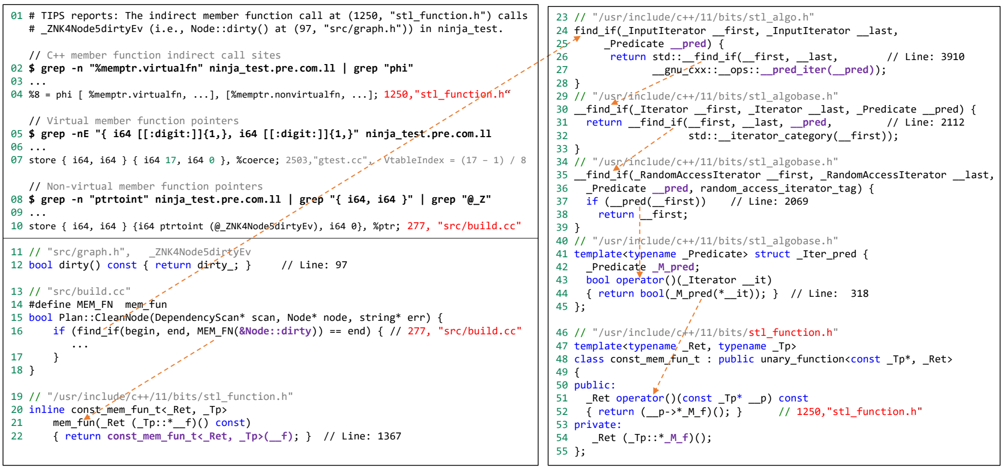
For `453.povray`, the address-taken member functions are exclusively non-virtual, and the classes in which they are defined do not contain any virtual functions. This scenario aligns with the non-virtual function-only micro-benchmarks included in Table 5, where SVF consistently scores 0, highlighting why SVF fails to identify such target functions in `453.povray`.

For `447.dealii`, virtual inheritance is used in its classes, exemplified by instances like "class Solver : public virtual Base<dim>" in `step-14.cc`. This corresponds to the virtual inheritance category listed in Table 5, where SVF exhibits significant imprecision. Some member function pointers in `447.dealii`, such as those pointing to crucial thread entry functions, necessitate precise resolution. For instance, the non-virtual template member function "LaplaceSolver::WeightedResidual<3>::solve_primal_problem()" in `step-14.cc` is one such critical thread entry function reported by Tips but missed by SVF. Resolving these member function pointers accurately is vital for analyzing these large C++ programs.

5.2.2 Twelve Open-Source C++ Programs. We begin by evaluating the precision enhancements of Tips compared to SVF, followed by case studies to explore the reasons behind these improvements.

Precision Improvements. Table 7 demonstrates Tips's superior precision over SVF across 12 open-source C++ applications, showing two trends in the "AvgPtsSize" column. Firstly, in scenarios similar to `453.povray` (shown in Table 6), where classes defining address-taken member functions lack virtual functions, SVF fails to identify non-virtual member function call targets. This is evident in all four Qt programs and two Ninja/GoogleTest programs (`build_log_perftest` and `manifest_parser_perftest`), where SVF's average points-to set size is zero, while Tips accurately identifies these pointers, usually with a slight increase in analysis time. Tips slightly outperforms SVF in `FileCheck` and `llvm-mt`, as SVF traverses many imprecisely introduced call edges during its pre-analysis. Secondly, for classes with virtual functions, as seen in the other six C++ programs from Table 7 and resembling the pattern in `447.dealii`, SVF's average points-to set size significantly exceeds that of Tips. This precision gain in Tips is attributed to the Gep_k rule (Table 3), and similar to the first pattern, SVF misses all non-virtual member function call targets, whereas Tips does not.

Case Studies. We performed manual analysis to evaluate Tips's precision advantage over SVF and to confirm the correctness of Tips's implementation. Let us examine GoogleTest's use in `ninja_test`. Due to multiple instantiations of class/function templates in C++, we found searching human-readable LLVM-IR text files useful for manual verification, while Tips automatically analyzes the equivalent LLVM-IR bitcode. To facilitate this, we crafted a Python script to embed line numbers and filenames into the LLVM-IR text files, leveraging existing debug information. This allows us



```
01 # TIPS reports: The indirect member function call at (1250, "stl_function.h") calls
   # _ZNK4Node5dirtyEv (i.e., Node::dirty() at (97, "src/graph.h")) in ninja_test.

   // C++ member function indirect call sites
02 $ grep -n "%memptr.virtualfn" ninja_test.pre.com.ll | grep "phi"
03 ...
04 %8 = phi [ %memptr.virtualfn, ...], [%memptr.nonvirtualfn, ...]; 1250,"stl_function.h“

   // Virtual member function pointers
05 $ grep -nE "{ i64 [[:digit:]]{1,}, i64 [[:digit:]]{1,}" ninja_test.pre.com.ll
06 ...
07 store { i64, i64 } { i64 17, i64 0 }, %coerce; 2503,"gtest.cc",  VtableIndex = (17 – 1) / 8

   // Non-virtual member function pointers
08 $ grep -n "ptrtoint" ninja_test.pre.com.ll | grep "{ i64, i64 }" | grep "@_Z"
09 ...
10 store { i64, i64 } {i64 ptrtoint (@_ZNK4Node5dirtyEv), i64 0}, %ptr; 277, "src/build.cc"

11 // "src/graph.h",   _ZNK4Node5dirtyEv
12 bool dirty() const { return dirty_; }     // Line: 97

13 // "src/build.cc"
14 #define MEM_FN  mem_fun
15 bool Plan::CleanNode(DependencyScan* scan, Node* node, string* err) {
16     if (find_if(begin, end, MEM_FN(&Node::dirty)) == end) { // 277, "src/build.cc"
         ...
17     }
18 }

19 // "/usr/include/c++/11/bits/stl_function.h"
20 inline const_mem_fun_t<_Ret, _Tp>
21     mem_fun(_Ret (_Tp::*__f)() const)
22     { return const_mem_fun_t<_Ret, _Tp>(__f); }  // Line: 1367
```

```
23 // "/usr/include/c++/11/bits/stl_algo.h"
24 find_if(_InputIterator __first, _InputIterator __last,
25      _Predicate __pred) {
26       return std::__find_if(__first, __last,        // Line: 3910
27              __gnu_cxx::__ops::__pred_iter(__pred));
28 }
29 // "/usr/include/c++/11/bits/stl_algobase.h"
30 __find_if(_Iterator __first, _Iterator __last, _Predicate __pred) {
31   return __find_if(__first, __last, __pred,         // Line: 2112
32                    std::__iterator_category(__first));
33 }
34 // "/usr/include/c++/11/bits/stl_algobase.h"
35 __find_if(_RandomAccessIterator __first, _RandomAccessIterator __last,
36   _Predicate __pred, random_access_iterator_tag) {
37   if (__pred(__first))    // Line: 2069
38     return __first;
39 }
40 // "/usr/include/c++/11/bits/stl_algobase.h"
41 template<typename _Predicate> struct _Iter_pred {
42   _Predicate _M_pred;
43   bool operator()(_Iterator __it)
44   { return bool(_M_pred(*__it)); }  // Line:  318
45 };

46 // "/usr/include/c++/11/bits/stl_function.h"
47 template<typename _Ret, typename _Tp>
48 class const_mem_fun_t : public unary_function<const _Tp*, _Ret>
49 {
50 public:
51   _Ret operator()(const _Tp* __p) const
52   { return (__p->*_M_f)(); }      // 1250,"stl_function.h"
53 private:
54    _Ret (_Tp::*_M_f)();
55 };
```

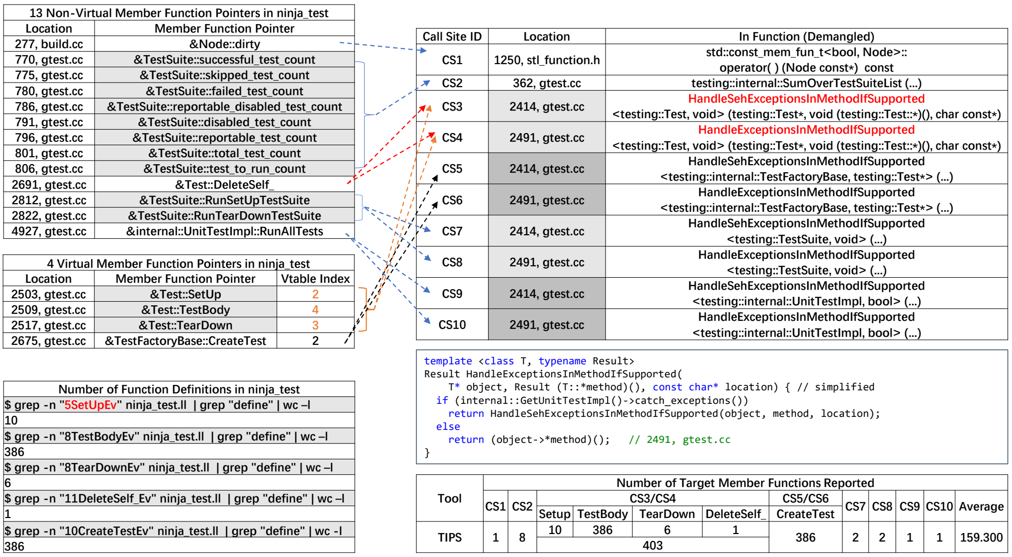
Fig. 9. Manual tracking of Tips's analysis on one member function call in ninja_test.

to accurately locate member function call sites and pointers in ninja_test, as demonstrated in Figure 9. For instance, as shown at line 1, Tips identifies a member function call in stl_function.h (line 1250) that may target Node::dirty() in src/graph.h (line 97). According to the information dumped at line 10, we determined the non-virtual function pointer (&Node::dirty) is declared in src/build.cc (line 277). By cross-referencing mangled and demangled function names from LLVM-IR texts to C++ source code, we could trace control and data flows from src/build.cc (line 277) to stl_function.h (line 1250), across the values depicted in bolded purple color in Figure 9. Thus, the non-virtual function pointer &Node::dirty at line 277 in src/build.cc may be invoked at line 1250 in stl_function.h. As both SVF and Tips are path-insensitive (to avoid the path-explosion problem), we ignored path conditions in our manual analysis.

Similarly, we manually tracked the value flows of additional member function pointers in ninja_test. As illustrated in Figure 10, there are 13 non-virtual member function pointers—targets missed by SVF—including the one in src/build.cc previously discussed and 12 in src/gtest.cc, along with 4 virtual member function pointers also located in src/gtest.cc. The pointers correspond to 10 call sites: CS1 in src/stl_function.h and CS2-CS10 in src/gtest.cc, with dashed arrows indicating usage. Analysis of LLVM-IR text files showed that two function templates, HandleExceptionsInMethodIfSupported() and HandleSehExceptionsInMethodIfSupported(), were instantiated four times each, covering CS3-CS10.

When C++ developers employ the TEST_F(test_fixture, test_name) macro (omitted in Figure 10) in gtest.h to introduce a test, GoogleTest generates a subclass of testing::Test with an overridden TestBody() function, executed at the two member function call sites, CS3 and CS4, at lines 2414 and 2491, respectively (gtest.cc). The object pointer from TestFactoryBase::CreateTest() reaches these two call sites, introducing different overridden virtual tables. Our tracking of four member function pointers—&Test::DeleteSelf_, &Test::SetUp, &Test::TestBody, and &Test::TearDown—revealed that Tips identified 403 potential target member functions at these two call sites. This includes 10 instances of SetUp(), 386 of TestBody(), and 6 of TearDown(), plus the non-virtual function Test::DeleteSelf_().

These findings align with the manual analysis's reachability data from the four member function pointers to call sites CS3 and CS4, as illustrated in the top section of Figure 10. Indirect confirmation of Tips's soundness in analyzing ninja_test was further obtained by enumerating function definitions in the LLVM-IR text files using grep, as shown in the bottom-left section of Figure 10. For



13 Non-Virtual Member Function Pointers in ninja_test	
Location	Member Function Pointer
277, build.cc	&Node::dirty
770, gtest.cc	&TestSuite::successful_test_count
775, gtest.cc	&TestSuite::skipped_test_count
780, gtest.cc	&TestSuite::failed_test_count
786, gtest.cc	&TestSuite::reportable_disabled_test_count
791, gtest.cc	&TestSuite::disabled_test_count
796, gtest.cc	&TestSuite::reportable_test_count
801, gtest.cc	&TestSuite::total_test_count
806, gtest.cc	&TestSuite::test_to_run_count
2691, gtest.cc	&Test::DeleteSelf_
2812, gtest.cc	&TestSuite::RunSetUpTestSuite
2822, gtest.cc	&TestSuite::RunTearDownTestSuite
4927, gtest.cc	&internal::UnitTestImpl::RunAllTests

4 Virtual Member Function Pointers in ninja_test		
Location	Member Function Pointer	Vtable Index
2503, gtest.cc	&Test::SetUp	2
2509, gtest.cc	&Test::TestBody	4
2517, gtest.cc	&Test::TearDown	3
2675, gtest.cc	&TestFactoryBase::CreateTest	2

Number of Function Definitions in ninja_test
$ grep -n "5SetUpEv" ninja_test.ll | grep "define" | wc –l
10
$ grep -n "8TestBodyEv" ninja_test.ll | grep "define" | wc –l
386
$ grep -n "8TearDownEv" ninja_test.ll | grep "define" | wc –l
6
$ grep -n "11DeleteSelf_Ev" ninja_test.ll | grep "define" | wc –l
1
$ grep -n "10CreateTestEv" ninja_test.ll | grep "define" | wc -l
386

Call Site ID	Location	In Function (Demangled)
CS1	1250, stl_function.h	std::const_mem_fun_t<bool, Node>:: operator( ) (Node const*) const
CS2	362, gtest.cc	testing::internal::SumOverTestSuiteList (...)
CS3	2414, gtest.cc	HandleSehExceptionsInMethodIfSupported <testing::Test, void> (testing::Test*, void (testing::Test::*)(), char const*)
CS4	2491, gtest.cc	HandleExceptionsInMethodIfSupported <testing::Test, void> (testing::Test*, void (testing::Test::*)(), char const*)
CS5	2414, gtest.cc	HandleSehExceptionsInMethodIfSupported <testing::internal::TestFactoryBase, testing::Test*> (...)
CS6	2491, gtest.cc	HandleExceptionsInMethodIfSupported <testing::internal::TestFactoryBase, testing::Test*> (...)
CS7	2414, gtest.cc	HandleSehExceptionsInMethodIfSupported <testing::TestSuite, void> (...)
CS8	2491, gtest.cc	HandleExceptionsInMethodIfSupported <testing::TestSuite, void> (...)
CS9	2414, gtest.cc	HandleSehExceptionsInMethodIfSupported <testing::internal::UnitTestImpl, bool> (...)
CS10	2491, gtest.cc	HandleExceptionsInMethodIfSupported <testing::internal::UnitTestImpl, bool> (...)

```
template <class T, typename Result>
Result HandleExceptionsInMethodIfSupported(
    T* object, Result (T::*method)(), const char* location) { // simplified
  if (internal::GetUnitTestImpl()->catch_exceptions())
    return HandleSehExceptionsInMethodIfSupported(object, method, location);
  else
    return (object->*method)();   // 2491, gtest.cc
}
```

Tool	Number of Target Member Functions Reported											
	CS1	CS2	CS3/CS4				CS5/CS6	CS7	CS8	CS9	CS10	Average
			Setup	TestBody	TearDown	DeleteSelf_	CreateTest					
TIPS	1	8	10	386	6	1	386	2	2	1	1	159.300
			403									

Fig. 10. Manual tracking of Tips's analysis on all member function calls in ninja_test.

instance, 5SetUpEv is the mangled name for SetUp(). Consequently, we verified that Tips comprehensively covers all overridden SetUp(), TestBody(), and TearDown() functions in ninja_test, with similar manual checks conducted for other call sites in this C++ program.

Our analysis shows the vital role of C++ member function pointers in real-world C++ applications. Tips outperforms SVF in resolving these pointers, prioritizing precision with small overhead.

5.3 Limitations

Currently, Tips only models integer arithmetic involved in C++ member function pointers. However, it does not yet handle more complex integer arithmetic operations like addition, multiplication, or division in other contexts (e.g., a * a + b * b). Furthermore, Tips is primarily designed for analyzing C/C++ programs on modern operating systems like Linux, where features like Memory Management Units (MMUs) and Address Space Layout Randomization (ASLR) are present. It is not directly suitable for analyzing bare metal programs on embedded systems (e.g., STM32), where physical memory addresses are hardcoded and cast into pointers for accessing different peripheral devices. Analyzing such systems would require domain-specific knowledge to model these hardcoded physical addresses accurately. Inline assembly code contained within C/C++ is also outside the scope of Tips since it operates at the LLVM-IR level (Figure 8). Finally, for handling C++ placement new, which is currently ignored, modifications to C++ compiler frontends to provide reliable type information would be necessary.

6 RELATED WORK

Below we summarize previous research on field-, flow-, and context-sensitive pointer analysis, and outline prospective client applications that could leverage Tips for improved accuracy and functionality.

Field-Sensitivity. Field-sensitive pointer analysis [Balatsouras and Smaragdakis, 2016, Pearce et al., 2007], discerns distinct fields within an object. Earlier, Andersen's pointer analysis lacks field sensitivity. Pearce et al. [Pearce et al., 2007] later introduced an exemplary field-sensitive pointer analysis by employing a field-index-based abstraction to represent different fields. This approach is representative of field-sensitive techniques. Both SVF [Sui and Xue, 2024] and our tool, Tips, employ field-index-based field-sensitive analysis. However, Tips represents an advance that expands the domain of pointer analysis to include integers. It maintains field-sensitivity, even when processing the Gep_k rule, crucial for precise handling of virtual inheritance and member function pointers in C++, as demonstrated in Figure 6.

Flow-Sensitivity. Flow-sensitive pointer analysis [Hardekopf and Lin, 2011, Kusano and Wang, 2016] respects program execution order through either a dense control flow graph or a sparse value flow graph for program representation. Initially, semi-sparse flow-sensitive analysis leveraged def-use chains for top-level pointers (e.g., virtual registers in LLVM-IR), often available in Static Single Assignment (SSA) form [Hardekopf and Lin, 2009, Zhao et al., 2018]. Recent developments have facilitated full-sparse flow-sensitive analysis, encompassing both top-level pointers and address-taken variables, achieved by creating SSA for address-taken variables based on pre-analysis [Sui et al., 2012]. Similar to SVF [Sui and Xue, 2024], our tool (Tips) adopts flow-sensitivity using the Sparse Value Flow Graph (SVFG) representation. By concurrently tracking both pointers and integers, Tips incorporates byte-level pointer adjustments for precise modeling of virtual table pointers, integrating strong updates in flow-sensitive pointer analysis for C++ (Figure 3).

Context-Sensitivity. Two main forms of context-sensitivity exist: call-site sensitivity [He et al., 2022, Jeon and Oh, 2022, Li et al., 2023, Oh et al., 2014, Yu et al., 2010] and object-sensitivity [He et al., 2022, Li et al., 2018, Liu and Huang, 2022, Lu and Xue, 2019, Ma et al., 2023, Milanova et al., 2005, Smaragdakis et al., 2011]. Call-site sensitivity suits system languages like C/C++, where objects can be allocated in various regions (heap, stack, and global memory), while object-sensitivity is better for languages like Java, where all objects are in the heap. Similar to SVF [Sui and Xue, 2016a, 2024], the context-sensitivity in our tool, Tips, is based on call-sites. However, we enhance our approach by incorporating integers into the domain and implementing the rules for integer-included context-sensitivity (Section 3.1) in our demand-driven context-sensitive pointer analysis for C++.

Client Applications. Pointer analysis, crucial for various static analysis techniques, supports applications like bug detection [Cai et al., 2021, Liu et al., 2016, Livshits and Lam, 2003, Yan et al., 2018], taint analysis [Arzt et al., 2014, Schubert et al., 2019, Wang et al., 2023], symbolic execution [Cadar et al., 2008, Guo et al., 2018, Trabish et al., 2020, 2018], and compiler optimization [Sui et al., 2013]. For instance, PhASAR [Schubert et al., 2019], an IFDS-based taint analysis framework using LLVM, utilizes pointer analysis to derive points-to information and construct improved call graphs, enhancing taint analysis for C++ programs. Additionally, in decompilers [Avast, 2024, Kang et al., 2015]—where integer-pointer casts are prevalent in lifted LLVM IRs—Tips's ability to track integer-pointer value flows is valuable for analyzing these IRs and advancing binary reverse engineering efforts [Erinfolami and Prakash, 2020, Fourtounis et al., 2020]. Tips can bolster control-flow integrity [Abadi et al., 2005, Fan et al., 2017, Jeon et al., 2017] for member function calls, addressing the shortcomings of existing unsound or imprecise pointer analyses [Hardekopf and Lin, 2009, Li et al., 2018, Liu et al., 2022, Shi et al., 2018, Sui and Xue, 2016b, Yu et al., 2010] (Figures 3 and 6).

7 CONCLUSION

We introduce Tips, the first field-, flow-, and context-sensitive pointer analysis for effectively resolving member function pointers in C++ programs. It enhances both soundness and precision in various inheritance scenarios, with small overhead. We also acknowledge plans for addressing limitations in future work.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported by an ARC Grant (DP240103194).

REFERENCES

Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 340–353.

Lars Ole Andersen. 1994. Program Analysis and Specialization for the C Programming Language. In *PhD Dissertation*. University of Copenhagen, Denmank, 111–152.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, USA, 259–269. https://doi.org/10.1145/2666356.2594299

Avast. 2024. A Retargetable Machine-Code Decompiler Based on LLVM. https://github.com/avast/retdec. Accessed May 10, 2024.

George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*. Springer, USA, 84–104.

Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, USA, 209–224.

Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: Practical Static Detection of Inter-Thread Value-Flow Bugs. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, USA, 1126–1140. https://doi.org/10.1145/3453483.3454099

The GoogleTest Open-Source Community. 2024a. Google Testing and Mocking Framework. https://github.com/google/googletest. Accessed May 10, 2024.

The LLVM Open-Source Community. 2024b. The LLVM Project. https://github.com/llvm/llvm-project. Accessed May 10, 2024.

The Ninja Open-Source Community. 2024c. Ninja. https://github.com/ninja-build/ninja. Accessed May 10, 2024.

The Qt Company. 2024. Qt Base. https://github.com/qt/qtbase. Accessed May 10, 2024.

Rukayat Ayomide Erinfolami and Aravind Prakash. 2020. Devil Is Virtual: Reversing Virtual Inheritance in C++ Binaries. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, USA, 133–148. https://doi.org/10.1145/3372297.3417251

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 the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 329–340. https://doi.org/10.1145/3092703.3092729

George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. 2020. Identifying Java Calls in Native Code via Binary Scanning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, USA, 388–400. https://doi.org/10.1145/3395363.3397368

Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial Symbolic Execution for Detecting Concurrency-Related Cache Timing Leaks. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, USA, 377–388. https://doi.org/10.1145/3236024.3236028

Ben Hardekopf and Calvin Lin. 2009. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, USA, 226–238. https://doi.org/10.1145/1594834.1480911

Ben Hardekopf and Calvin Lin. 2011. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *International Symposium on Code Generation and Optimization*. IEEE, USA, 289–298. https://doi.org/10.1109/CGO.2011.5764696

Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework for Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis (Artifact). *Dagstuhl Artifacts Ser.* 8, 2 (2022), 06:1–06:3. https://doi.org/10.4230/DARTS.8.2.6

Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: Call-Site Sensitivity Can Be Superior to Object Sensitivity Even for Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of programming Languages*. ACM, USA, 1–29. https://doi.org/10.1145/3498720

Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, USA, 2373–2387. https://doi.org/10.1145/3133956.3134062

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-Pointer Casts. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, USA, 326–335. https://doi.org/10.1145/2813885.2738005

Jakob Koschel, Pietro Borrello, Daniele Cono D'Elia, Herbert Bos, and Cristiano Giuffrida. 2023. Uncontained: Uncovering Container Confusion in the Linux Kernel. In *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, USA, 5055–5072.

Markus Kusano and Chao Wang. 2016. Flow-Sensitive Composition of Thread-Modular Abstract Interpretation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, USA, 799–809. https://doi.org/10.1145/2950290.2950291

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *International symposium on Code Generation and Optimization*. IEEE, USA, 75–86.

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-To Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, USA, 278–289. https://doi.org/10.1145/1273442.1250766

Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-To Analysis using Spark. In *Proceedings of the 12th International Conference on Compiler Construction*. Springer, USA, 153–169.

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proceedings of the 2018 26th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, USA, 129–140. https://doi.org/10.1145/3276988

Yuanbo Li, Qirun Zhang, and Thomas Reps. 2023. Single-Source-Single-Target Interleaved-Dyck Reachability via Integer Linear Programming. In *Proceedings of the ACM SIGPLAN Symposium on Principles of programming Languages*. ACM, USA, 1003–1026. https://doi.org/10.1145/3571228

Bozhen Liu and Jeff Huang. 2022. SHARP: Fast Incremental Context-Sensitive Pointer Analysis for Java. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, USA, 1–28. https://doi.org/10.1145/3527332

Peiming Liu, Yanze Li, Brad Swain, and Jeff Huang. 2022. PUS: A Fast and Highly Efficient Solver for Inclusion-Based Pointer Analysis. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, USA, 1781–1792. https://doi.org/10.1145/3510003.3510075

Peng Liu, Omer Tripp, and Xiangyu Zhang. 2016. IPA: Improving Predictive Analysis with Pointer Analysis. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, USA, 59–69. https://doi.org/10.1145/2931037.2931046

V Benjamin Livshits and Monica S Lam. 2003. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. IEEE, USA, 317–326. https://doi.org/10.1145/940071.940114

LLVM. 2024. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html. Accessed May 10, 2024.

Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (2019), 29 pages. https://doi.org/10.1145/3360574

Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, USA, 539–564. https://doi.org/10.1145/3591242

Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2005. Parameterized Object Sensitivity for Points-To Analysis for Java. *ACM Transactions on Software Engineering and Methodology* 14, 1 (2005), 1–41. https://doi.org/10.1145/1044834.1044835

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-Sensitivity Guided by Impact Pre-Analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, USA, 475–484. https://doi.org/10.1145/2594291.2594318

David J Pearce, Paul HJ Kelly, and Chris Hankin. 2007. Efficient Field-Sensitive Pointer Analysis of C. *ACM Transactions on Programming Languages and Systems* 30, 1 (2007), 4–es. https://doi.org/10.1145/1290520.1290524

Tristan Ravitch. 2024. Whole Program LLVM. https://github.com/travitch/whole-program-llvm. Accessed May 10, 2024.

Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-Procedural Static Analysis Framework for C/C++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, USA, 393–410.

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, USA, 693–706. https://doi.org/10.1145/3192366.3192418

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, USA, 17–30. https://doi.org/10.1145/1925844.1926390

Yulei Sui, Yue Li, and Jingling Xue. 2013. Query-Directed Adaptive Heap Cloning for Optimizing Compilers. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, USA, 1–11. https://doi.org/10.1109/CGO.2013.6494978

Yulei Sui and Jingling Xue. 2016a. On-Demand Strong Update Analysis via Value-Flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. ACM, USA, 460–473.

https://doi.org/10.1145/2950290.2950296

Yulei Sui and Jingling Xue. 2016b. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM/IEEE, USA, 265–266. https://doi.org/10.1145/2892208.2892235

Yulei Sui and Jingling Xue. 2024. SVF. https://github.com/SVF-tools/SVF. Accessed May 10, 2024.

Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static Memory Leak Detection using Full-Sparse Value-Flow Analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, USA, 254–264. https://doi.org/10.1145/2338965.2336784

Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Past-Sensitive Pointer Analysis for Symbolic Execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, USA, 197–208. https://doi.org/10.1145/3410246

David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, USA, 350–360. https://doi.org/10.1145/3180155.3180251

Xizao Wang, Zhiqiang Zuo, Lei Bu, and Jianhua Zhao. 2023. DStream: A Streaming-Based Highly Parallel IFDS Framework. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE/ACM, USA, 2488–2500. https://doi.org/10.1109/ICSE48619.2023.00208

Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-After-Free Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, USA, 327–337. https://doi.org/10.1145/3180155.3180178

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 *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM/IEEE, USA, 218–229. https://doi.org/10.1145/1772954.1772985

Jisheng Zhao, Michael G Burke, and Vivek Sarkar. 2018. Parallel Sparse Flow-Sensitive Points-To Analysis. In *Proceedings of the 27th International Conference on Compiler Construction*. ACM, USA, 59–70. https://doi.org/10.1145/3178372.3179517

Changwei Zou. 2024. TIPS. https://github.com/sheisc/TIPS.git. Accessed May 10, 2024.

Received 2023-09-28; accepted 2024-04-16