Protecting C++ Dynamic Dispatch Through VTable Interleaving

Dimitar Bounov
Computer Science and Engineering
University of California, San Diego
dbounov@cs.ucsd.edu

Rami Gökhan Kıcı
Computer Science and Engineering
University of California, San Diego
rkici@cs.ucsd.edu

Sorin Lerner
Computer Science and Engineering
University of California, San Diego
lerner@cs.ucsd.edu

***Abstract*—With new defenses against traditional control-flow attacks like stack buffer overflows, attackers are increasingly using more advanced mechanisms to take control of execution. One common such attack is *vtable hijacking*, in which the attacker exploits bugs in C++ programs to overwrite pointers to the virtual method tables (vtables) of objects. We present a novel defense against this attack. The key insight of our approach is a new way of laying out vtables in memory through careful ordering and interleaving. Although this layout is very different from a traditional layout, it is backwards compatible with the traditional way of performing dynamic dispatch. Most importantly, with this new layout, checking the validity of a vtable at runtime becomes an efficient range check, rather than a set membership test. Compared to prior approaches that provide similar guarantees, our approach does not use any profiling information, has lower performance overhead (about 1%) and has lower code bloat overhead (about 1.7%).**

I. Introduction

For performance reasons, many applications are written in languages without garbage collection, without memory bounds checking, and without strong runtime type systems. At the same time, these applications are large and complex, and thus are usually implemented in a language that supports abstraction. As a result, oftentimes, the language of choice in practice for these applications is still C++.

Unfortunately, because there is no enforcement of runtime type safety, C++ applications are vulnerable to many kinds of attacks, including control-flow attacks that take control of the program's execution. The most commonly known such attack is the traditional stack buffer overflow that overwrites the return address. Because of the importance of such attacks, defense mechanisms have been developed to mitigate these attacks, including DEP [31], ASLR [29], stack canaries [6], shadow stacks [3], SafeStack [20], etc.

As a result of these defenses, the cost of mounting stack-based attacks has increased, and attackers have started looking for new ways of compromising control-flow integrity (CFI). One such approach which has received a lot of attention in the past few years is known as *vtable hijacking*. In vtable hijacking, an attacker exploits bugs in a C++ program to overwrite the pointers to virtual method tables of C++ objects. When the program performs a virtual method call later on, the attacker-controlled virtual method table is used for dispatch, causing the attacker to take over the control-flow of the program. There are several kinds of bugs that can be used to mount this attack, including heap overflow bugs and use-after-free bugs. The importance of vtable hijacking is highlighted by several high-profile attacks on up-to-date browsers [10], [14], [12], [37], [34].

Because of the importance of vtable hijacking, several recent research efforts have looked at how to protect against this attack, including [16], [35], [42], [30]. In general these techniques insert runtime checks before a virtual method call that try to establish the safety of the call. Most of those checks attempt to verify that when performing a virtual call on an object of type A, the virtual method table used will be that of A or a subclass of A. Enforcing this property efficiently is non-trivial, as Section III will discuss in more detail.

The main contribution of this paper is an entirely new approach for efficiently enforcing the above runtime type safety property for C++ virtual method calls, thus protecting against vtable hijacking. The key insight of our approach is a new way of laying out vtables in memory through careful ordering and interleaving. Although this layout is very different from a traditional layout, it is still backwards compatible with the traditional way of performing dynamic dispatch. Most importantly, with this new layout, checking the validity of a vtable at runtime becomes an efficient range check.

This achievement of using range checks to enforce runtime types is made possible by three technical ingredients. First, by ordering vtables using a preorder traversal of the class hierarchy we ensure that vtables for classes in each subtree of the hierarchy are consecutive in memory. Next by aligning the vtables properly, we demonstrate how range checks modulo alignment become precise enough to guarantee that only statically type-safe vtables are allowed at each virtual method call site. Finally through careful interleaving, we remove the extra padding for alignment, pack the vtables tightly and greatly reduce our memory footprint, while still being able to use efficient range checks modulo alignment.

As we will describe in more detail later in the paper, compared to prior approaches that provide similar guarantees,

NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23421

our approach does not use any profiling information, has lower performance overhead (about 1%) and has lower code bloat overhead (about 1.7%).

In summary, our contributions are:

- We present a new approach for enforcing precise CFI for C++ virtual method calls using simple range checks and alignment (Sections II through VI).
- We show how our approach can be adapted to work for all the corner cases of C++ hierarchies, including multiple inheritance and virtual base classes (Section VII).
- We formalize our approach in a well defined algorithm and prove several important properties about it, including correctness with respect to C++'s dynamic dispatch mechanism (Section VIII).
- We perform a comprehensive performance evaluation of our approach for runtime and memory overhead on the C++ SPEC2006 benchmarks and the Chrome browser (Section X). We even experimentally compare our approach against the state of the art for protecting against vtable hijacking that was recently added to LLVM (but which has not been published in an academic venue), and show that our approach reduces the runtime overhead from about 2% to about 1%, a significant reduction when considering the small margins that are in play (in essence, we cut the overhead in half). Finally we discuss the security guarantees provided by our algorithm.

II. Background

A. C++ VTables

Dynamic dispatch in C++ is implemented using tables of function pointers (vtables). We use a running example to explain how vtables work, and how our approach works. Consider the C++ code in Figure 1a, which defines the class hierarchy shown in Figure 1b. Figure 2 shows the traditional vtable layout for this example. Note that the details of vtable memory layout are dictated by two Application Binary Interfaces (ABIs) - Itanium and MSVC. We will use Itanium for the remainder of the paper, but our technique transfers naturally to MSVC.

VTables are contiguous arrays of pointer-sized entries containing function pointers and other metadata such as Run-Time Type Information (rtti), virtual base and call offsets (relevant to virtual inheritance) and offset-to-top (used for casting). Object instances contain at offset 0 a pointer (vptr) to the first function pointer (the address point) of their corresponding vtable. For example in Figure 2 the object of type `D` points to the 2nd entry in its vtable - `Dfoo`. Elements of the vtable are indexed relative to vtable's address point, with function pointers found at positive indices and other metadata laid out at negative indices. For example, in Figure 2, method `foo` can be found at index `0` relative to the address point; `bar` is at index `0x8`, and so is `baz`; finally `boo` is at index `0x10`.

Note that `bar` and `baz` are mapped to the same index since no class in the hierarchy has both a `bar` and a `baz` method. Also note that for simplicity, we only show one kind of metadata at negative offsets, namely Run-time Type Information - other kinds of metadata at negative offsets are handled similarly.

To illustrate how dynamic dispatch works, we use the sample method calls shown in Figure 1c. The resulting assembly code, shown in Figure 1d, works in three steps, which are labeled in the generated assembly:

1) Dereference the object pointer to obtain the vptr.
2) Use the method offset to index into the vtable to obtain the actual function pointer. For the first call site (`C1`) the called method `foo` has offset `0` and for the second call site (`C2`) the called method `bar` has offset `0x8`.
3) Finally invoke the obtained function pointer. For simplicity we omit the assembly code that passes parameters, including passing `this`.

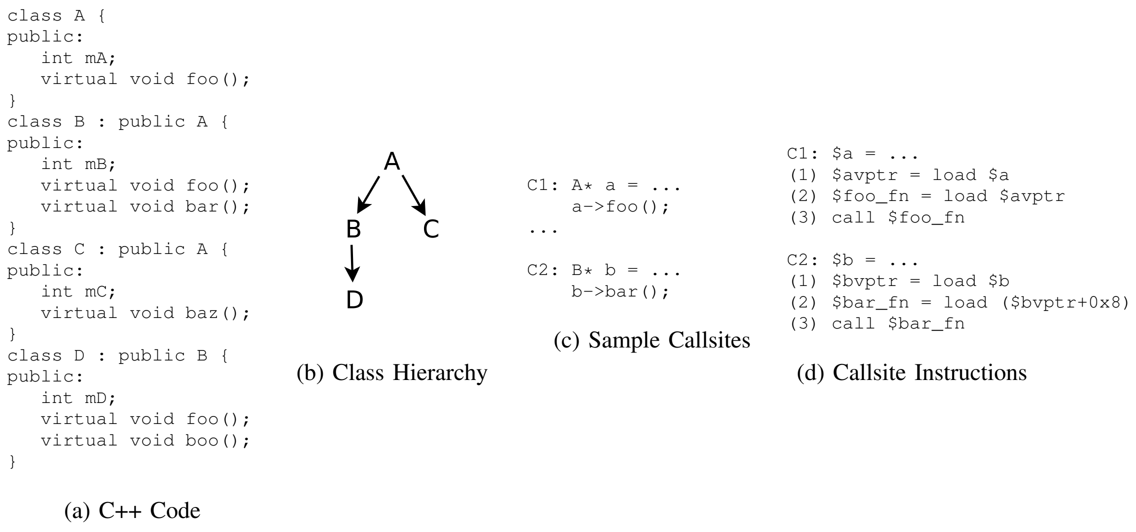
B. Threat Model

We assume an adversary capable of arbitrarily and concurrently modifying the heap (similarly to [35]). We assume that values in registers are safe from tampering. We assume that the attacker does not compromise the stack. This assumption is necessary since our technique is currently implemented at the LLVM Intermediate Representation (IR) level, where we have no control over register spilling. Thus it is possible that a value verified by our runtime checks is spilled to the stack prior to use, thus introducing a time-of-check-time-of-use vulnerability. To overcome this limitation we would need to reimplement our defense as a lower level compiler transformation.

C. VTable Hijacking

While vtables reside in read-only memory and are thus safe from tampering, vptrs are stored in writable memory and become the target of vtable hijacking attacks [16]. Attackers usually exploit a bug already present in the program to overwrite the vtable pointer of an object and later trick the program into invoking a function on the compromised object. We illustrate this with a use-after-free bug, because this has been a common way of launching recent high-profile vtable attacks [14], [37]. Consider for example the code excerpt in Figure 3, which suffers from a use-after-free bug: object `d` is used (4) erroneously after it is deleted at (1). Now suppose that an attacker is capable of controlling a memory allocation (2) in the window between (1) and (4) in such a way that the memory allocator places the new memory where object `d` used to live. Furthermore, let's assume that the attacker also controls writes to this new allocation (3), thus being able to place attacker controlled data where the vptr used to be; the new vptr is constructed by the attacker to point to a specially crafted fake vtable with malicious entries. As a result, when `d->foo()` is invoked at (4) control-flow is redirected to a location of the attacker's choice (e.g. the `system` call).

Such use-after-free bugs have been successfully exploited in the wild to launch vtable hijacking attacks, most notably in browser security exploits [10], [14], [12], [37], [34], [24], [27]. However, use-after-free bugs are only one possible vector for vtable hijacking, another option being direct heap-overflow attacks [1], [2]). Our approach detects all vtable hijacking attacks, regardless of their source.



```
class A {
public:
   int mA;
   virtual void foo();
}
class B : public A {
public:
   int mB;
   virtual void foo();
   virtual void bar();
}
class C : public A {
public:
   int mC;
   virtual void baz();
}
class D : public B {
public:
   int mD;
   virtual void foo();
   virtual void boo();
}
```

(a) C++ Code

(b) Class Hierarchy

```
C1: A* a = ...
    a->foo();
...
C2: B* b = ...
    b->bar();
```

(c) Sample Callsites

```
C1: $a = ...
(1) $avptr = load $a
(2) $foo_fn = load $avptr
(3) call $foo_fn

C2: $b = ...
(1) $bvptr = load $b
(2) $bar_fn = load ($bvptr+0x8)
(3) call $bar_fn
```

(d) Callsite Instructions

Fig. 1: C++ Example

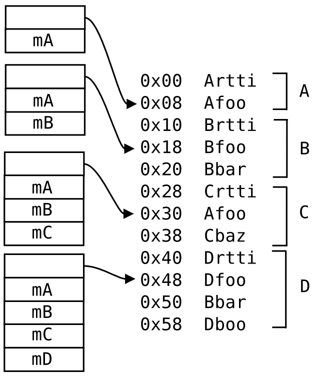


Fig. 2: Normal Vtable Layout in Memory

```
C1: $a = ...
    $avptr = load $a
    assert isvalid $avptr, A
    $foo_fn = load $avptr
    call $foo_fn

C2: $b = ...
    $bvptr = load $b
    assert isvalid $bvptr, B
    $bar_fn = load ($bvptr+0x8)
    call $bar_fn
```

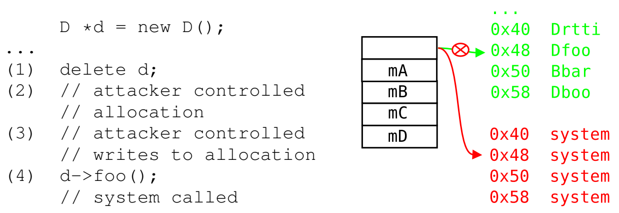
(a) Abstract Check

```
C1: $a = ...
    $avptr = load $a
    assert $avptr ∈ {0x8,0x18,
         0x30,0x48}
    $foo_fn = load $avptr
    call $foo_fn

C2: $b = ...
    $bvptr = load $b
    assert $bvptr ∈ {0x18,0x48}
    $bar_fn = load ($bvptr+0x8)
    call $bar_fn
```

(b) Vptr check semantics

Fig. 4: Instrumented Callsites



```
    D *d = new D();
...
(1) delete d;
(2) // attacker controlled
    // allocation
(3) // attacker controlled
    // writes to allocation
(4) d->foo();
    // system called
```

Fig. 3: VTable hijacking example

III. Overview

Having presented how vtable hijacking works, we now focus on how to prevent such attacks. Most defenses against vtable hijacking (ours included) employ Inline Reference Monitors (IRMs) inserted before dynamic dispatch sites. At an abstract level, the callsites from Figure 1d would be instrumented as shown in Figure 4a, where the isvalid $v C instruction denotes a runtime check whether the vptr $v is valid for a class of static type C. The key challenge, as we will shortly see, is how to make these checks efficient.

In practice, isvalid has the semantics of a set check, as shown in Figure 4b. For example, callsite C2 is safe if at runtime the loaded vptr $bvptr is in the set {0x18, 0x48}. This safe set is determined statically by looking at the type in the source code of the object being dispatched on. In this case, the method call is done using variable b in the source code of Figure 1c, which is statically declared of type B. The valid vtables that can occur at runtime for an object of static type B are the vtables of B and all of its subclasses, in our example B and D. This is why the safe set is {0x18, 0x48}, the vtables of B and D. Similarly, the set for C1 in Figure 4b are the vtables of A (the statically declared type of a) and its sublcasses, namely the vtables of A, B, C, and D.

Efficient enforcement of these vptr checks is difficult, due to the fact that vtable address points occur at non-uniform addresses in memory (Fig 2). This has forced previous work to resort to various general set data structures such as bitsets [22], [30] and linear arrays with inlined checks [16].

Our Approach. The key insight in our approach is that we carefully arrange and interleave the vtables so that the addresses to check for at a given call site become continuous modulo some alignment. We can check membership in these sets of continuous addresses using a simple range and align-

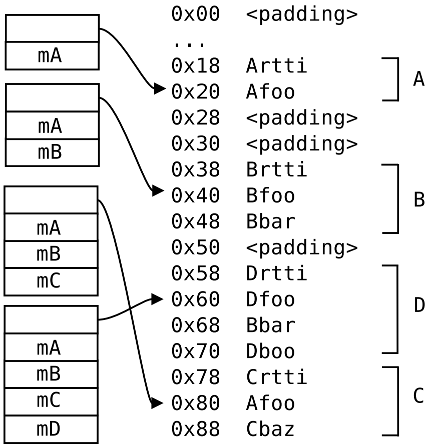


Fig. 5: Ordered Vtable Layout in Memory

```
C1: $a = ...
    $avptr = load $a
    assert (0x20 ≤ $avptr ≤ 0x80)∧($avptr & 0x1f == 0)
    $foo_fn = load $avptr
    call $foo_fn


C2: $b = ...
    $bvptr = load $b
    assert (0x40 ≤ $bvptr ≤ 0x60)∧($bvptr & 0x1f == 0)
    $bar_fn = load ($bvptr+0x8)
    call $bar_fn
```

Fig. 6: Ordered VTable Range Checks

Class	Set of Runtime Types	Start	End	Alignment
A	A,B,C,D	0x20	0x80	0x20
B	B,D	0x40	0x60	0x20
C	C	0x80	0x80	0x20
D	D	0x60	0x60	0x20

Fig. 7: Valid Address Point Ranges for Ordered VTables

ment check. These checks are very efficient and their overhead is independent of the size of the set being checked.

Our vtable layout is motivated by the observation that the sets of classes to be checked are not arbitrary. Instead these sets are all so-called *cones* in the class hierarchy, namely a set containing a given class and all of its subclasses.

As a result, our **first** key idea is to *order* the vtables in memory using a pre-order traversal of the class hierarchy. This places all the vtables belonging to a cone continuously in memory. In addition, if we align the vtables appropriately, this ordering allows us to check cone membership using a simple range check modulo alignment. Section IV will present this ordering technique, called OVT (Ordered VTables).

Although this ordering gives us the range checks we want, it has high memory overhead because of the additional padding required for alignment. As a result, our **second** key idea is to *interleave* the vtables in such a way that padding is not required anymore. Section V will present this interleaving technique, called IVT (Interleaved VTables).

Throughout the following sections, we present OVT and IVT through our running example. We then show how to efficiently implement the checks, and then how to handle complex cases like multiple inheritance and virtual inheritance. Finally, we will present the detailed algorithms for our approach.

IV. VTable Ordering

Figure 5 presents an ordered vtable layout for our running example from Figure 1. The vtables are ordered by a preorder traversal order of the class hierarchy (resulting in C and D being switched). Given this new layout, the vtables of any cone (subtree) in the class hierarchy are laid out consecutively. Furthermore, padding has been added so that all vtable address points are 32 byte aligned (multiples of `0x20`). Thus, in the new memory layout, the valid vptrs for a variable of static type C are the vtables for classes in the cone rooted at C, and these vtables are precisely the 32-byte aligned entries in a given range. The two checks from Figure 4b become aligned range checks as shown in Figure 6. The correct ranges for all classes in Figure 5 are shown in Table 7.

It is crucial to align vtables and perform the alignment checks, since otherwise the range check by itself would not catch vptrs that (incorrectly) point into the middle of a vtable. We choose alignment as the method for preventing middle-of-vtable pointers because as we will show in Section VI, we can perform such alignment checks very efficiently. The alignment size is 2^n for the smallest n such that 2^n is larger than the largest vtable, which allows us to fit any vtable between two alignment points, and thus place address points in memory at 2^n intervals.

Note that in our example (Figure 5) we cannot immediately start laying out vtables at offset 0. This is due to the fact that address points are generally not the first entry in a vtable but we want precisely address points to be aligned modulo a power of 2. In particular, since `Afoo` is the 2nd entry in A's vtable, we need to add `0x18` bytes of padding before A's vtable, so that `Afoo` ends up being a multiple of `0x20`.

V. VTable Interleaving

VTable ordering provides an efficient way of checking set membership for cones of the class hierarchy. However, the ordering approach imposes a significant memory overhead (as discussed further in Section X): every vtable in the same class hierarchy must be padded up to the smallest power of 2 larger than the size of the biggest vtable. This overhead can be especially problematic in memory constrained environment such as mobile and embedded development.

To reduce this memory overhead, we introduce another technique, which is far more radical than the ordering idea: we make vtables *sparse* and *interleaved* in such a fashion that no space will be wasted, and address points will end up being consecutive entries in memory. As we will see shortly, the new layout of vtables will be very different than a traditional layout, but quite surprisingly will still be compatible with the traditional technique of doing dynamic dispatch.

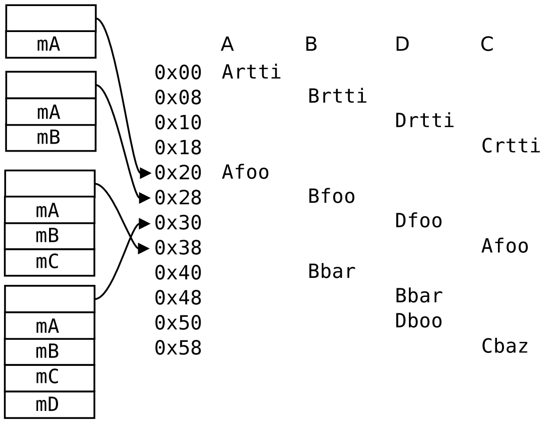


Fig. 8: Sparse and Interleaved VTables

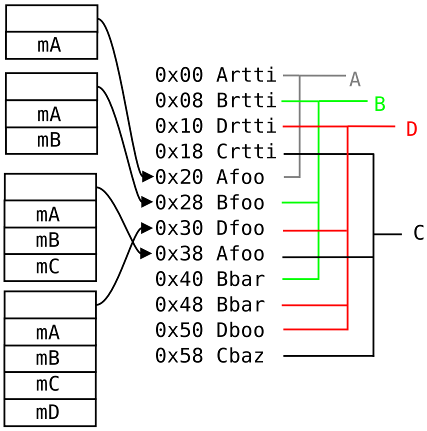


Fig. 9: Interleaved Vtable Layout in Memory

VTable Entry	Old Offset	New Offset
rtti	-0x8	-0x20
foo	0	0
bar	0x8	0x18
boo	0x10	0x20
baz	0x8	0x20

Fig. 10: New Method Offsets

```
C1: $a = ...
    $avptr = load $a
    assert (0x20 ≤ $avptr ≤ 0x38)∧($avptr & 0x7 == 0)
    $foo_fn = load $avptr
    call $foo_fn


C2: $b = ...
    $bvptr = load $b
    assert (0x28 ≤ $bvptr ≤ 0x30)∧($bvptr & 0x7 == 0)
    $bar_fn = load ($bvptr+0x18)
    call $bar_fn
```

Fig. 11: Interleaved VTable Range Checks

Continuing with our running example, Figure 8 presents the corresponding interleaved layout for the Ordered VTables from Figure 5. We have spread the vtable entries in separate columns to highlight what original vtable each entry belongs to. Figure 9 shows another view of the interleaved vtables, with the original vtable membership marked with lines of different colors.

To better understand how the interleaving works, consider the interleaved vtable for class D (3rd column in Figure 8). This vtable is equivalent to the original vtable, except that 3 empty entries have been inserted between Drtti and Dfoo, and 2 empty entries have been inserted between Dfoo and Bbar (in D's vtable). These additional entries are "empty" from the point of view of D's vtable, but in the final memory layout they are filled with entries from other interleaved vtables.

The number of empty spaces to insert between two entries is the key to getting the different interleaved vtables to "gridlock" together perfectly. The number of empty spaces inserted between two entries of an original vtable is determined by the shape of the class hierarchy and the number of new methods defined by each class in the hierarchy. Note that this can vary for different pairs of entries in a single vtable.

One very important property of our interleaved layout is that the amount of space inserted between two entries is always the same in all vtables that define or overload these two entries. For example, the additional empty space inserted between foo and bar is always two "empty" entries, in all vtables that define or overload them (B and D in this case). This means that, although vtable entries in our interleaved layout are now at different offsets than in a traditional layout, the offsets are still the same across all vtables which define or overload them, which is the property needed for dynamic dispatch. Figure 10 shows the old and new constant offsets for each one of the vtable entries in our running example. Our implementation must therefore not only change the vtable layout but also all the vtable offsets used in code sections for indexing into vtables.

Although here we have only given an intuition of how our approach works, a detailed algorithm for building vtables and computing the new offsets, and a formal proof of our preservation of vtable offsets between parent and child classes can be found in Section VIII.

Finally, runtime checks for interleaved vtables have exactly the same form as before – range checks modulo alignment. The only difference is that, whereas previously the alignment of ordered vtables varied by vtable, the alignment for interleaved vtables is always 8 bytes, the size of a vtable entry. A uniform 8-byte alignment is sufficient because, as can be seen in Figures 8 and 9, the address points of all vtables are now contiguous in memory: the address points for A, B, C and D are all consecutively laid out between 0x20-0x38. For example the valid address points for B (which are the vtables for B and its subclass D) are the 8-byte aligned values in the range 0x28-0x30. As another example, the valid address points for A (which are the vtables for A and its subclasses, B, C and D) are the 8-byte aligned values in the range 0x20-0x38.

The corresponding checks for IVTs for our example in Figure 4 are shown in Figure 11. The set of ranges for all classes in our example hierarchy is shown in Table 12. Note that unlike previous fine-grained vtable protection work [16], [22] the overhead of our runtime checks is independent from the size and complexity of the class hierarchy.

Class	Set of Runtime Types	Start	End	Alignment
A	A,B,C,D	0x20	0x38	0x8
B	B,D	0x28	0x30	0x8
C	C	0x38	0x38	0x8
D	D	0x30	0x30	0x8

Fig. 12: Valid Address Point Ranges for Interleaved VTables

```
...
cmp $vptr, $a
jlt FAIL
cmp $vptr, $b
jgt FAIL
and $vptr, 1111_l
cmp $vptr, 0
jne FAIL
... // Success
```

(a) 3-branch check

```
...
and $vptr, 1...1_{64-l}0...0_l
cmp $vptr, $a
jlt FAIL
cmp $vptr, $b
jgt FAIL
... // Success
```

(b) 2-branch check

Fig. 13: Naive Range Check Implementation

VI. Range Check Optimization

We implement 3 further optimizations to reduce the cost of range checks. These optimizations are adapted from similar optimizations in the implementation of forward-edge CFI in LLVM [22].

A. Single Branch Range Checks

Both IVT and OVT rely on an efficient check of the form "is value v in a range $[a, b]$ and aligned modulo 2^l ?". In the case of OVT each tree in the decomposed hierarchy has its own specific l. For Interleaved VTables, we always have $l = 3$ (i.e. we maintain that candidate vptr values are aligned modulo the size of a vtable entry).

A naive implementation of this check requiring 3 branches is presented in Figure 13a. Code is presented in a simplified assembly where `cmp` represents the unsigned comparison operator, `jlt` `jgt` and `jne` denote "jump-if-less-than", "jump-if-greater-than" and "jump-if-not-equal" respectively, and `and` represents the bitwise and operator. The 3 branches in Figure 13a respectively check whether `$vptr` is below the desired range, above it, or not properly aligned. We can eliminate the last branch by enforcing alignment rather than checking for it as shown in Figure 13b. This however still requires 2 branches.

We perform both the range and alignment check with a single branch using the instruction sequence in Figure 14, a technique we adapted from the LLVM 3.7 forward-edge CFI implementation [22]. Here `rotr $v, l` signifies the right bit rotation of `$v` by l bits.

To see how the code in Figure 14 is equivalent to the original range and alignment check, we will show that it fails in all cases when the original check does - when `$vptr` > `$b`, when `$vptr` < `$a`, and when `$vptr` is not aligned.

First note that if `$vptr` > `$b` then `($vptr-$a)` > `($b-$a)` and therefore `$diff` > `($b-$a)` and finally `($diff >> l)` > `(($b-$a) >> l)`. For unsigned comparisons `(rotr $diff, l)` ≥ `($diff >> l)`, and therefore `$diffR` ≥ `($diff >> l)` > `(($b-$a) >> l)`. Therefore in this case we will fail the check.

```
...
$diff = $vptr - $a
$diffR = rotr $diff, l
cmp $diffR, ($b-$a) >> l
jgt FAIL
... // Success
```

Fig. 14: Range Check Implementation

If `$vptr` < `$a` then `$diff` is negative and thus has its highest bit set. Therefore one of the `l+1` highest bits of `$diffR` is set. However none of the highest `l+1` bits of `(($b-$a) >> l)` can be set. Therefore, since the comparison is unsigned again we fail the check.

If `$vptr` is between `$a` and `$b`, and any of its `l` lowest bits is set, then after the rotation we will fall in the previous case and again fail the check.

Finally if `$vptr` is between `$a` and `$b` and `l` bit aligned, `rotr $diff, l` becomes equivalent to `$diff >> l`. Since no arithmetic operations overflow, the check succeeds in this case.

Thus we have reduced our runtime check to a shorter instruction sequence, containing a single branch that is never taken during normal program execution (unlike [16]). In the future, our enforcement check can be further sped up by using hardware acceleration through the Intel MPX bounds checking instructions coming in the Skylake architecture [15].

B. Single Value Ranges

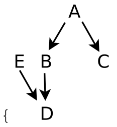
When a given range contains a single value the aligned range check can be reduced to an equality check. Traditionally one would expect all such cases to be devirtualized by earlier compiler optimizations. However, as we will discuss in Section X we observe singleton ranges surprisingly often. We believe that this discrepancy is due to the fact that LLVM's optimizer does not implement a C++ specific devirtualization pass. LLVM's optimizations are aimed to be language agnostic and devirtualization happens as the result of several simpler optimizations including Global Value Numbering and Constant Load Elimination. Each of those relies only on the information available at the LLVM IR level. We on the other hand implement a C++ specific IR transformation, that leverages C++ metadata propagated down to our optimization passes. Furthermore our transformations work in a fully statically linked setting, and thus assume that the whole class hierarchy is known at link time. Note that we could actually optimize this case further by devirtualizing these single target calls that LLVM does not optimize.

C. Constant Vptrs

We have observed a small percentage of cases where a vptr can be statically checked. In such cases the safety of the virtual call can be discharged statically, and no runtime check is needed. We believe such instances arise when short constructors are inlined into larger functions. At that point, the address pointer becomes available as a constant value in the larger function and can be used directly rather than being

```
...
class E {
public:
   int mE;
   virtual void faz();
}
class D : public E, public B {
...
```

(a) C++ Code



(b) Class Hierarchy

Fig. 15: Multiple Inheritance Example

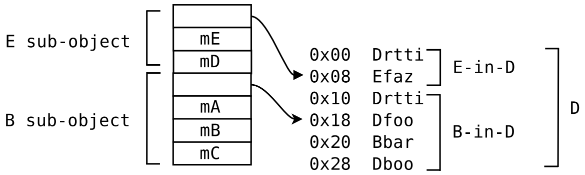


Fig. 16: Multiple Inheritance Memory Layout for D

loaded from the object. Similarly to the case of singleton ranges we believe these callsites have not been devirtualized by LLVM since devirtualization is the result of language agnostic passes.

VII. Complex Class Hierarchies

Our discussion so far assumed a simple tree-shaped hierarchy. C++ however supports multiple and virtual inheritance, which result in non-tree class hierarchies. To explain our handling of these we extend the example in Figure 1 with multiple inheritance by adding another base class E for D (Figure 15).

To handle multiple inheritance C++ dictates that for classes with multiple bases a separate sub-object is emitted within each object instance - one for each direct base class. Furthermore each sub-object contains a separate vptr pointing into a different sub-vtable within the object's vtable. In the ABI's terms the object's vtable becomes a *vtable group* consisting of multiple *primitive vtables*. For example Figure 16 shows the two sub-objects within D corresponding to the two base classes - E and B.

Each sub-object contains a vptr to a different sub-vtable within D's vtable group. Note that since each primitive vtable inherits its layout from precisely one (primitive) base vtable we can break down a complex hierarchy into several simple tree-shaped hierarchies containing only primitive vtables, as shown in Figure 17

Virtual Inheritance throws a kink in this approach as virtual diamonds can sometimes result in diamonds even in the decomposed hierarchies. We handle these cases by breaking one of the diamond edges to layout the vtables. This necessitates support for multiple ranges per check. In practice multiple ranges per check are very rare. Even in our largest benchmark (chrome), the average number of ranges per callsite is very close to 1 (≈ 1.005).

This decomposition allows us to reduce complex hierarchies with both multiple and virtual inheritance to forests of primitive tree hierarchies which we know how to handle.

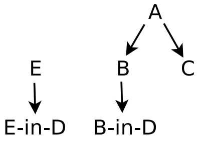


Fig. 17: Decomposed Class Hierarchy

```
1  Order(R)
2   // List of ordered and aligned vtable entries
3   ovtbl = [ ]
4   order = pre(R)
5   // Map of new address points for each class
6   addrPtM = { }
7
8   n = max(len(vtbl(C)) for C in order)
9   padLen = $2^{\lceil \log_2(n) \rceil + 1}$
10
11  for C in order do
12    v = vtbl(C)
13    nzeros = padLen - ((len(ovtbl)+addrPt(v)) $mod$ padLen)
14    ovtbl.append(zeros(nzeros))
15    addrPtM[C] = len(ovtbl) + addrPt(v)
16    ovtbl.append(v)
17
18  return (ovtbl, addrPtM)
```

Fig. 18: VTable Ordering Algorithm: Given the root R of a tree in the decomposed class hierarchy, the ***Order*** function returns the ordered vtable layout in memory, and a map from each class to its address point in the memory layout.

VIII. Algorithms

In this section we discuss the algorithms used for building Ordered and Interleaved VTables and their correctness. For both, we assume that the class hierarchy has already been decomposed into a collection of trees containing only primitive vtables. Both algorithms operate on one tree at a time. We denote by pre(R) the list of classes of a tree rooted at class R in preorder traversal order. We further denote by vtbl(C) the primitive vtable for class C (represented as a list of entries) in the tree, by addrPt(v) the offset of the unique address point in the primitive vtable v and by zeros(n) a function returning a list of n zeros.

A. Ordering

To build Ordered VTables, for each tree in the class hierarchy, we follow the algorithm in Figure 18. The presented algorithm finds the size of the largest vtable n (line 8) in the current tree, and using n, stores in padLen a sufficiently large power of 2 (line 9) so that no two consecutive address points in the preorder traversal are more than padLen entries apart. Next it performs a preorder traversal and appends primitive vtables consecutively (lines 11-16), while adding sufficient 0-padding (lines 13-14) before each appended vtable (line 16) so that its address point is a multiple of padLen. The algorithm returns the newly ordered vtables ovtbl along with the generated map addrPtM describing the new address points of all classes in the tree. This information is sufficient

```
1  Interleave(R)
2   // Map (classes->(old indices->new indices))
3   newInd = { }
4   ivtbl = [ ] // list of interleaved entries
5   order = pre(R)
6   posM = { } // Map (classes->cur. position)
7   addrPtM = { } // Map (classes->new addr. point)
8
9   i = 0
10  for C in order do
11    posM[C] = 0
12    addrPtM[C] = i
13    i++
14
15  do
16    for C in order do
17      v = vtbl(C)
18      if (posM[C] < len(v))
19        newInd[C][posM[C]] = len(ivtbl) - addrPtM[C]
20        ivtbl.append(v[posM[C]])
21        posM[C]++
22  while (ivtbl changes)
23
24  return (ivtbl, addrPtM, newIndM)
```

Fig. 19: VTable Interleaving Algorithm: : Given the root R of a tree in the decomposed class hierarchy, the ***Interleave*** function returns the interleaved vtable layout in memory, a map from each class to its address point in the memory layout, and a map from each class and each index in its old vtable to the corresponding new index in the memory layout.

to generate checks and update all uses of the old vtables with the new ordered ones.

Termination. The algorithm obviously terminates (it contains a single bounded loop) and preserves all original primitive vtables in t in ovtbl (since it appends each at some iteration in line 16).

Correctness. Since addrPtM[C] is set only once for each class C at line 15, and at that point it correctly reflects C's new address point in ovtbl we can reason that addrPtM correctly translates address points.

To show that address points can be efficiently checked for any subtree in C, it is sufficient to note that:

1) All address points are multiples of padLen, which is a power of 2.
2) For each subtree the valid address points lay in a continuous range due to preorder traversal (line 11).
3) In a given range, there is no value that is a multiple of padLen and not an address point (this follows from the fact that no 2 address points are more than padLen appart).

B. Interleaving

In Figure 19 we present the algorithm used for building the interleaved vtable for each tree of primitive vtables. To simplify presentation the algorithm shown here handles only the positive indices in a vtable and assumes that the original address point for each vtable is 0. Handling negative indices of vtables and a non-zero address point is analogical.

The algorithm in Figure 19 takes in the root R of a tree of primitive vtables and initializes the following data structures:

- newInd - a map from the classes in pre(R) and their old vtable indices to their new indices in the interleaved layout
- ivtbl - the new interleaved memory layout represented as a list of all the entries of the original vtables in the interleaved order
- posM - a temporary map containing for each class the index of the next vtable entry to be added to the interleaved layout
- addrPtM - a map containing the new address point in ivtbl of each class in pre(R).

In lines 10-13 the algorithm initializes the posM map to all 0s, and addrPtM with the new address point for each class. As evident from the loop, for each class C the new address point becomes its position in the preorder traversal. The core of the algorithm consists of the two nested loops on lines 15-22. The outer loop (lines 15-22) iterates as long as we are accumulating values in ivtbl, while the inner loop (lines 16-21) traverses the classes in preorder traversal order (order) accumulating (if possible) an entry from each vtable. The inner loop uses the posM map to keep track of its current position in each vtable. At each iteration of the inner loop, if we have not reached the end of the current vtable (line 18), we add one entry from this vtable to ivtbl (line 20) and increment the current position posM[C] for this vtable (line 21). Note that when adding a new entry to ivtbl in the inner loop, we also record in newInd (line 19) what the new index of this entry is. The algorithm returns a triple containing ivtbl, addrPtM and newIndM which is sufficient for building the interleaved vtable and updating all uses and indices of the old vtables.

Termination. The algorithm terminates, since:

1) The outer loop (lines 15-22) terminates when ivtbl stops changing.
2) Whenever we add an entry to ivtbl (line 20) we also increment some entry in posM (line 21).
3) All entries in posM are bounded due to line 18.

Correctness. To establish the correctness of the algorithm we must further show that:

1) All entries from the old vtables are present in ivtbl.
2) When we update address points and indices in the program using addrPtM and newIndM we will not change old behavior.
3) Indices for inherited fields are consistent between base and derived classes.

Our strategy for proving the above properties is that we will establish several key lemmas and corollaries, which together will immediately imply the above properties. In the following lemmas, unless otherwise mentioned, when referring to the newInd, ivtbl and addrPtM data structures, we mean their state after the algorithm is done. We start with the following helper lemma:

Lemma 1. *Lines 19-21 are executed exactly once for each* $\texttt{C} \in \texttt{pre(R)}$ *such that* $\texttt{pos[C]} = i \; \forall i . 0 \leq i < len(\texttt{vtbl}(C))$

Proof: This follows from the fact that for all $\texttt{C} \in \texttt{pre(R)}$: (1) posM[C] is initialized to 0, (2) posM[C] increases monotonically and (3) the algorithm doesn't terminate until posM[C]=len(vtbl(C)). ■

The above lemma implies that each entry in newInd is set precisely once and never changed, and that for each $\texttt{C} \in \texttt{pre(R)}$ and $\forall i.0 \leq i < \mathit{len}(\texttt{vtbl(C)})$ newInd[C][i] is well defined. Additionally, since by the previous lemma the values of C and i span all classes in pre(R) and all vtable indices for each class, then the value of v[posM[C]] at line 20 spans (exactly once) all entries of all original primitive vtables. Therefore we can establish the following corollary:

Corollary 1. $\forall \texttt{C} \in \texttt{pre(R)}$ *and* $\forall E \in \texttt{vtbl(C)}$ E *occurs exactly once in* ivtbl.

Corollary 1 establishes requirement (1) - that each entry from the original vtables is represented once in the interleaved layout. Next we establish requirement (2) with the following lemma:

Lemma 2. $\forall \texttt{C} \in \texttt{pre(R)}$ *and* $\forall i.0 \leq i < \mathit{len}(\texttt{vtbl}(C))$ vtbl(C)[i]=ivtbl[addrPtM[C]+newInd[C][i]]

Since we use addrPtM and newInd to translate address points and vtable indices, after translation the expression used to obtain the i-th entry of the vtable of a class C would be ivtbl[addrPtM[C]+newInd[C][i]]. Therefore the above lemma states that after translating the indices and address points with addrPtM and newInd, when getting the i-th entry of the vtable of a class C, we would obtain precisely the same value (vtbl(C)[i]) that we would have gotten using normal vtables.

Proof: By Lemma 1 $\forall \texttt{C} \in \texttt{pre(R)}$ and $\forall i.0 \leq i < \mathit{len}(\texttt{vtbl(C)})$ newInd[C][i] is well defined and set only once at line 19. Lets denote by m the length of ivtbl at the time that newInd[C][i] is set. Since in that same iteration vtbl(C)[i] is appended at the end of ivtbl, and ivtbl only grows, it follows that at the end of the algorithm $\texttt{ivtbl}[m] = \texttt{vtbl(C)[i]}$. However at the end of the algorithm $\texttt{newInd[C][i]} = m - \texttt{addrPtM[C]}$ (line 19). Therefore, at the end of the algorithm ivtbl[newInd[C][i]+addrPtM[C]]=vtbl(C)[i]. ■

Next, to establish requirement (3) we first prove a small helper lemma:

Lemma 3. *For* $\forall \texttt{B, D} \in \texttt{pre(R)}$ *where* B *is a superclass of* D *if at the beginning of an iteration of the outer loop (line 16)* $\texttt{posM[B]} < \mathit{len}(\texttt{vtbl(B)})$ *then* $\texttt{posM[B]} = \texttt{posM[D]}$.

Proof: This follows by induction on the number of iterations of the outer loop. First posM[B] and posM[D] are both initialized to 0. Next, lets assume that at the beginning of some iteration of the outer loop $\texttt{posM[D]} = \texttt{posM[B]}$. Since $\texttt{posM[B]} < \mathit{len}(\texttt{vtbl(B)})$ and since $\mathit{len}(\texttt{vtbl(B)}) \leq \mathit{len}(\texttt{vtbl(D)})$ (because D is derived from B) it follows that $\texttt{posM[D]} < \mathit{len}(\texttt{vtbl(D)})$. Therefore, since both $\texttt{posM[B]} < \mathit{len}(\texttt{vtbl(B)})$ and $\texttt{posM[D]} < \mathit{len}(\texttt{vtbl(D)})$ at the start of the outer loop, then both posM[B] and posM[D] are incremented in that iteration of the outer loop at line 21. Therefore they will be equal in the next iteration of the outer loop as well. ■

Now using Lemma 3 we can finally establish requirement (3):

Lemma 4. $\forall \texttt{B, D} \in \texttt{pre(R)}$ *where* B *is a superclass of* D, *and* $\forall i.0 \leq i < \mathit{len}(\texttt{vtbl(B)})$, newInd[B][i]=newInd[D][i].

Proof: Let newInd[B][i] be set in the k-th iteration of the outer loop. Therefore at the start of the k-th iteration $\texttt{posM[B]} < \mathit{len}(\texttt{vtbl(B)})$ (due to line 18). Therefore by Lemma 3 at the beginning of that k-th iteration of the outer loop $\texttt{posM[D]} = \texttt{posM[B]} = i$. Furthermore, since $\texttt{posM[D]} = \texttt{posM[B]} < \mathit{len}(\texttt{vtbl(B)}) \leq \texttt{len(vtbl(D))}$ it follows that newInd[D][posM[D]] also is set in the k-th iteration of the outer loop. But posM[D]=posM[B]=i therefore newInd[D][i] is set in the k-th iteration of the outer loop.

Finally, let $\texttt{len(ivtbl)}_B$ denote len(ivtbl) when newInd[B][i] is set (in the inner loop) and $\texttt{len(ivtbl)}_D$ denote len(ivtbl) when newInd[D][i] is set. Since the inner loop follows preorder traversal, then all classes X visited by the inner loop from the time we set newInd[B][i] to the time we set newInd[D][i] are derived from B. By applying Lemma 3 for each of those classes X, it follows that $\texttt{posM[X]} = \texttt{posM[B]}$ and by the same argument as for D, for each of those classes lines 19-21 will be executed and thus an entry will be appended to ivtbl. Therefore $\mathit{len}(\texttt{ivtbl})_D = \mathit{len}(\texttt{ivtbl})_B + (\texttt{order.indexOf}(D) - \texttt{order.indexOf}(B))$. Therefore:

$$\texttt{newInd}[D][i] = \mathit{len}(\texttt{ivtbl})_D - \texttt{addrMap}[D] =$$
$$\mathit{len}(\texttt{ivtbl})_B + (\texttt{order.indexOf}(D) -$$
$$\texttt{order.indexOf}(B)) - \texttt{order.indexOf}(D) =$$
$$\mathit{len}(\texttt{ivtbl})_B - \texttt{order.indexOf}(B) = \texttt{newInd}[B][i]$$

The remainder follow from the fact that newInd[D][i] and newInd[B][i] are set only once by Lemma 1. ■

To rehash, we have shown that our interleaving algorithm is correct by establishing 3 properties - it preserves all of the original vtable entries (Corollary 1), it correctly computes the new indices for vtable entries in the IVT (Lemma 2), and finally it does not break the assumptions on layout preservation between base and derived classes (Lemma 4).

IX. Implementation

We implemented our technique in the LLVM [19] compiler framework. Our implementation required change to about 50 lines of C++ code and the addition of another 6820 lines of C++ code. Our change is split between the Clang compiler frontend (approx. 900 LOC), and several new link-time passes we added (approx 5900 LOC). Our tool supports separate compilation by relying on the link time optimization (LTO) mechanism built in LLVM [21]. To support LTO all intermediate object files are emitted in LLVM's higher-level IR, which

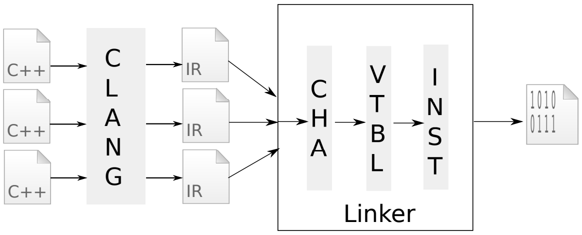


Fig. 20: Tool Workflow

retains additional semantic information necessary at link time to reconstruct the global class hierarchy and identify all sites in the code that must be modified. The workflow of our tool is presented in Figure 20.

First source files are fed through our modified Clang compiler, which adds some semantic information for use by later passes:

- Metadata describing the structure of vtable groups and local class hierarchies
- Placeholder instructions for vptr checks
- Placeholder instructions for new vtable indices

Note that placeholder instructions are necessary, as the precise ranges for vptr checks and new indices (for interleaving) depend on the global class hierarchy which is not known at compile time.

Work at link time is done as part of the LLVM gold plugin, and is spread in several steps:

- CHA - gather vtable metadata; construct global class hierarchy; decompose it in primitive vtable trees
- VTBL - build the new ordered/interleaved vtables
- INST - replace placeholder instructions based on the previous 2 steps

X. Evaluation

We evaluate our approach on several C++ benchmarks including several C++ programs from SPEC2006 as well as the Chrome browser. For the Chrome browser we evaluate performance overhead on several standard industry benchmarking suites including sunspider, octane, kraken and some of the more intensive parts of Chrome's performance tests suite. Figure 21 lists our benchmarks along with statistics like line count, number of classes, and number of static sites in the code where a virtual method is called. These benchmarks represent realistic large bodies of C++ code (e.g. Chrome contains over 1M lines of C++ code). Chrome especially exercised many corner cases of the C++ ABI, including combinations of virtual and multiple inheritance, all of which we handled. All of the benchmarks ran unmodified, with the exception of xalancbmk which contained a CFI violation: a class is cast to its sibling class and methods are invoked on it. We are not the only ones to report this violation – LLVM-VCFI also reports it. Because the layouts of the two sibling objects in this benchmark are similar enough, the violation went by unnoticed before. We fixed this CFI violation in xalancbmk (4 lines of code) before running it through our experimental setup.

Name	#LOC	#Classes	#Callsites	Avg. #T/C
astar	11684	1	1	1
omnetpp	47903	111	961	21.1235
xalancbmk	547486	958	11253	5.96188
soplex	41463	29	557	4.00359
povray	155177	28	120	1.74167
chrome	1M	20294	129054	51.0186

Fig. 21: Benchmark names along with several statistics: #LOC is number of lines of code; #Classes is the number of classes with virtual methods in them; #Callsites is the number of virtual method calls; and #T/C is the average number of possible vtables per virtual method call site, according to the static type system.

A. Runtime Overhead

Figure 22 shows the percentage runtime overhead of our approach, with the baseline being LLVM O2 with link time optimizations turned on. The bars marked OVT correspond to checks based on Ordered VTables, while IVT bars correspond to checks based on Interleaved VTables. For comparison, we also include the runtime of the very recent LLVM 3.7 forward-edge CFI implementation [22] (columns labeled LLVM-VCFI). This LLVM work has not been published in an academic venue, but as far as we are aware, it is the fastest technique to date that offers similar precision to ours. For each static type, LLVM-VCFI emits a bitset that encodes the set of valid address points for that type. Dynamic checks are then used to test membership in these bitsets. By default, LLVM-VCFI also checks downcasts, which we don't do. As a result, we disabled this option in LLVM-VCFI in our experimental evaluation. Runtimes for each benchmark are averaged over 50 repetitions.

Interleaving achieves an average runtime overhead of 1.17% across all benchmarks, and 1.7% across the Chrome benchmarks. Note that this is almost 2 times faster compared to LLVM-VCFI, which achieves 1.97% on average for all benchmarks, and 2.9% on Chrome benchmarks. Additionally, the average overhead of ordered vtables is 1.57%, which is higher than interleaved vtables, but lower than LLVM-VCFI. Given that interleaving and ordering employ the same runtime checks, the faster runtime of interleaving stems from better memory caching performance due to the removed padding.

One of the benchmarks (soplex) exhibits a small (<-1%) negative overhead for all 3 compared techniques. We believe this to be due to alignment or memory caching artifacts.

There are two benchmarks where, according to the exact numbers, LLVM-VCFI appears slightly faster than OVT: astar and omnetpp. We talk about each of the two benchmarks in turn. All of the overheads in astar are extremely small. This is because there is a single virtual method call site in this benchmark. If we look at the exact numbers for that benchmark, LLVM-VCFI has an overhead of about 0.1% and IVT has a slightly larger overhead, about 0.13%. The difference of 0.03% is so small that it is comparable to same-run variations caused by noise. Regarding omnetpp, LLVM-VCFI has overhead of 1.17% and IVT has overhead of 1.18% – again the difference of 0.01% is so small that it is in the noise of same-run variations.

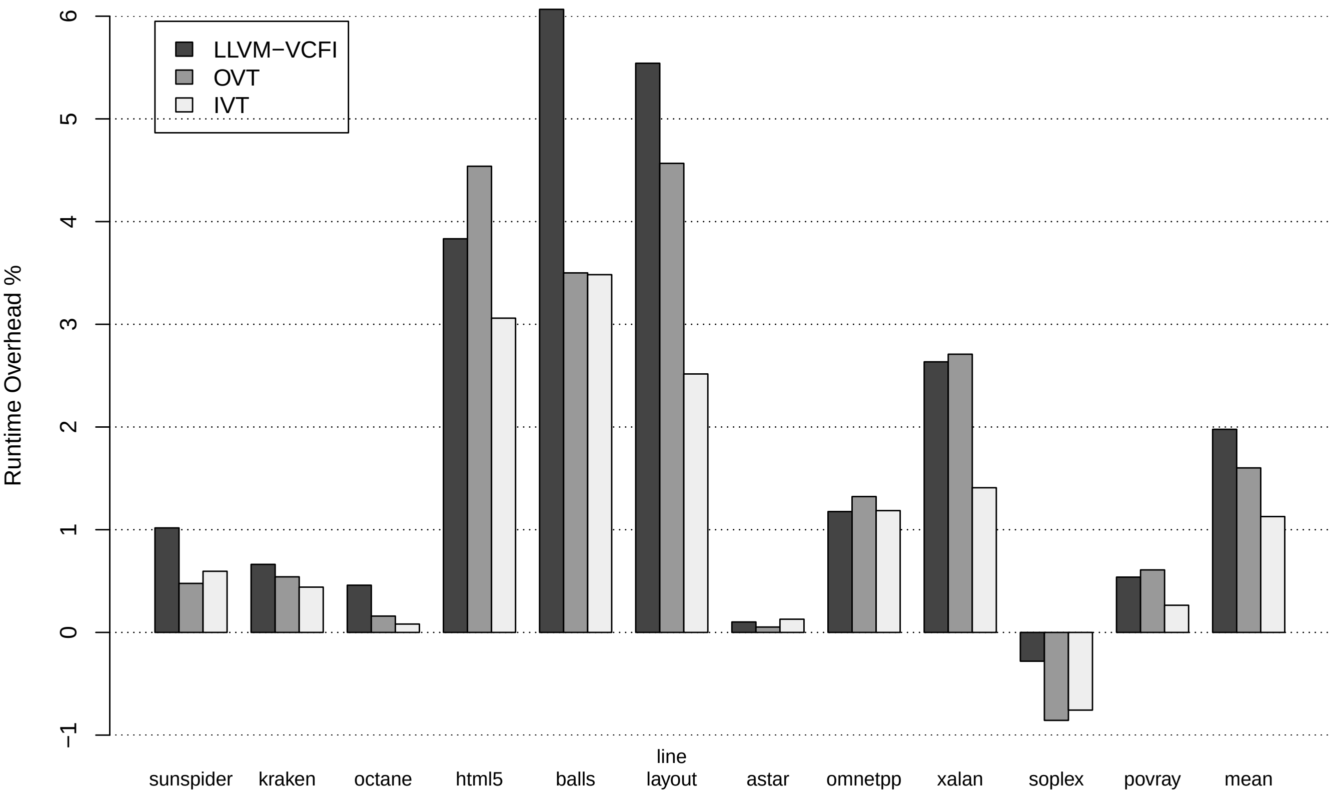


Fig. 22: Percentage runtime overhead. The baseline is LLVM O2 with link-time optimizations. LLVM-VCFI is the recent state-of-the-art LLVM 3.7 forward-edge CFI implementation; OVT is our ordering approach; IVT is our interleaving approach, which provides the best performance.

To better understand the source of runtime overhead in the interleaved approach, we disable our checks while maintaining the interleaving layout. We find that interleaving alone, without any checks, causes roughly 0.5% overhead across all benchmarks. This tells us that out of the total 1.17% overhead in the interleaving approach, 0.5% is due to caching effects from the vtable layout itself, and the additional 0.67% is caused by the checks.

B. Size Overhead

Figure 23 presents binary size overhead, again for LLVM-VCFI, OVT and IVT. On average, OVT has the highest increase in binary size, about 5.9% – this is because in addition to adding checks to the binary, OVT also adds a lot of padding to align vtables (and the vtables are also stored in the binary text). LLVM-VCFI has the next largest binary size overhead, at about 3.6%. LLVM-VCFI's binary size overhead comes from checks that are added to the binary, vtable aligning and from the bitsets that are stored in the binary. Finally, IVT has the smallest binary size overhead, at about 1.7%. The only overhead in IVT are the checks – there is no alignment overhead and no additional data structures that need to be stored.

C. Range Check Optimization Frequency

To better understand how frequently the range-check optimizations from Section VI are applied, Figure 24 shows the breakdown of these different optimizations as a percentage of all virtual method call sites. Each bar represents a benchmark, and the shaded regions in each bar show the percentage of virtual method call sites that are optimized in a particular way. More specifically, regions shaded `no_check` represent the percentage of call sites in which the check was optimized away due to a statically verified vptr. On average, these account for about 1.5% of all virtual method call sites. Regions shaded as `eq_check` represent the percentage of call sites for which the range check was optimized to a single equality. On average these account for a surprisingly large percentage of all call sites - approximately 26%, indicating that this optimization is particularly valuable. Finally the regions shaded `range` represent the remaining call sites, where a full range check was needed, on average approximately 72%.

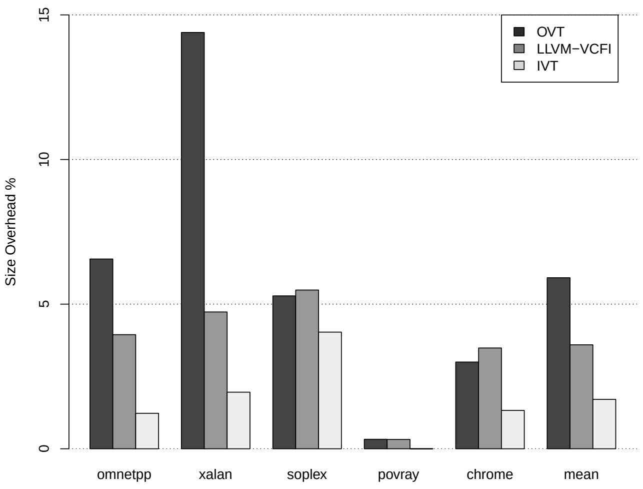


Fig. 23: Percentage binary size overhead. The baseline is LLVM O2 with link-time optimizations.

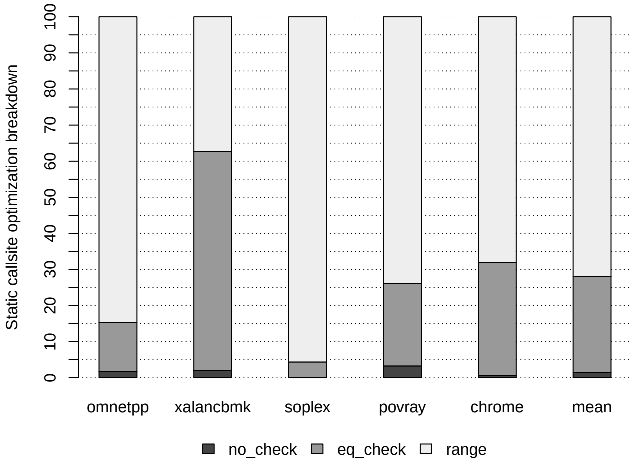


Fig. 24: Callsite optimization breakdown

D. Overhead comparison with existing work

Our experimental evaluation in the prior sections compared against the state-of-the-art LLVM 3.7 forward-edge CFI (LLVM-VCFI), which is the most efficient implementation of similar precision to our work. Here we broaden the scope of our comparison to also include performance numbers reported in other academic papers. In this broader setting, our technique achieves lower runtime and memory overhead compared to all vtable protection techniques of comparable precision. SafeDispatch [16] and LLVM-VCFI both achieve roughly 2% overhead on the Chrome and SPEC benchmarks we evaluated, compared to our overhead of about 1.17%. SafeDispatch also requires profile-guided optimizations, and the SafeDispatch paper [16] shows that when programs are profiled on one set of inputs, but then run on a different set if inputs, the overhead increases even more. VTV[35] has around 4% overhead on SPEC2006, whereas our overhead on the same subset of SPEC2006 (471.omnetpp, 473.astar, 483.xalancbmk) is about 0.9%.

vfGuard[30] and vtInt[42] both have higher overhead (18% and 2% respectively). The main focus of those techniques is identifying vtables and virtual call sites in stripped binaries, and extracting a class hierarchy. Because this is a very hard task, these techniques inevitably will have less precise information than our approach, which naturally leads to less precision in the checks.

As mentioned and fixed in a recent paper [13], there are some corner cases that certain vtable integrity techniques, including LLVM-VCFI, do not handle precisely. We have a way in our system to handle these cases, using multiple range checks, as was mentioned in Section VII. Because LLVM-VCFI does not handle these cases, we have disabled multiple range checks in our system when collecting the numbers reported in this section. If we enable these checks, thus getting more precise protection, we would handle the corner cases mentioned in [13] – although note that an empirical comparison of precision with [13] is difficult due to the large difference in tested versions of chrome (version 32 vs. version 42). If we enable multiple range checks, our overhead for Chrome remains the same, whereas our overall overhead for all SPEC2006 benchmarks goes from about 0.45% to 0.89%.

Finally, while an improvement from 2% to 1% may seem small initially, it is important to realize that it corresponds to a *halving* of the overhead, without the need for profile guided optimizations and also with minimal memory overhead.

E. Security Analysis

Our technique enforces the C++ type system constraints at each dynamic dispatch site. Namely we guarantee that if in the source code a given dynamic dispatch is performed on a variable of static type C, then the vtable used at runtime will be the vtable for C or a subclass of C. In column 5 of Figure 21 we list the average number of possible vtables for a dynamic dispatch site in each benchmark. As such, we believe our defense is effective against attack such as Counterfeit Object-Oriented Programing[32].

Since our technique is implemented at the LLVM IR level, it is possible for the later register allocator to decide to spill a checked vptr value on the stack. In this case, if the attacker can additionally mount a stack overwrite attack, this opens up the possibility for a time-of-check-time-of-use attack. The stack can be protected through a variety of mechanisms, including [6], [20] (although recent work [5] has shown that many of these stack defenses are not as strong as originally thought). To fully overcome these kinds of attacks, we believe we would need to implement our approach at a lower-level in the LLVM compiler, to achieve explicit control over register allocation, and prevent register spilling of vptr values between the time they have been checked and the time they are used.

Finally, similarly to LLVM-VCFI our technique does not currently handle C++ pointers to member function. Pointers to member functions are a C++ construct containing an index to a method in a vtable. When used for dynamic dispatch, the stored index is used to look up the target method in the vtable of the object on which we are invoking. In the case of Ordered VTables it is possible to check that the index is contained in the vtable of the target class with a single range check. This however still leaves considerable freedom for an attacker, and might be insufficient. In the case of Interleaved VTables handling member pointers is further complicated by the fact that vtables are broken up in multiple ranges. Member pointers are currently left as future work as discussed in Section XII.

XI. Related Work

The cost of control-flow hijacking attacks has motivated a rich body of research on prevention and mitigation. We can broadly split related work into 4 groups – vtable protection, general CFI enforcement, Software Fault Isolation (SFI) and other mitigation techniques.

A. VTable Protection

Closest to our work are techniques focusing on vtable protection and forward-edge control-flow enforcement. SafeDispatch [16] and VTV [35] both present compiler-based transformations that achieve similar precision to us. SafeDispatch incurs higher overhead – 2.1% on Chrome and SPEC2006 (vs. 1.1% for us across all benchmarks), and requires profile guided optimizations to achieve its overhead (which we do not). VTV has lower precision [13] and has higher overhead – 4.1% on the C++ benchmarks of SPEC2006 (vs. 0.9% for us on the C++ benchmarks SPEC2006). Additionally unlike those two techniques, the overhead of our runtime checks does not depend on the size of the class hierarchy.

Another branch of work focuses on VTable protection for COTS binaries, an approach that does not require source code. vfGuard [30] reconstructs a coarse class hierarchy from stripped binaries by relying on C++ ABI derived invariants. It incurs a higher overhead (18% on Firefox modules) due to the use of dynamic instrumentation. Their class hierarchy reconstruction is orthogonal and complimentary to our work. VTint [42] identifies writeable vtables and relocates them to read only memory, separate from other data. At each virtual method call site they check that the target vtable is read-only by attempting to write to it, and thus forcing an exception. Since these exceptions involve a context switch to the kernel, we believe that their overhead will be significantly higher compared to our technique. The reported overhead for vtInt is only 2% on average, however it is measured over a significantly smaller number of instrumented call sites. For example for xalancbmk the authors report only 1.12% overhead, but they find only 29 vtables and instrument 4248 call sites whereas we find 958 vtables and instrument 11253 call sites. In our experience we have not encountered any vtables laid out in writable memory by LLVM.

LLVM 3.7 [22] implements a virtual call CFI scheme utilizing bitsets (we called this the LLVM-VCFI technique in our experimental evaluation). As we have already shown in Section X, their technique has the same precision as ours, but at higher runtime overhead (1.97% vs 1.17%) and higher memory overhead (3.6% vs 1.7%).

Redactor++[8] provides a probabilistic defense against vtable confusion attacks with similar overhead to us – 1.1% over Chrome and SPEC2006. Unlike their work, our guarantees are not probabilistic.

B. General CFI

General CFI techniques protect all computed control transfers – including normal function pointer calls and returns. Due to this difference in scope a direct comparison of the runtime overhead between our technique and work described in this section is difficult. In general we achieve lower runtime overhead than all surveyed work here. It's important to keep in mind that we protect a smaller set of computed transfers than general CFI techniques, although for that smaller set, we typically provide stronger guarantees.

CFI was first introduced by Abadi et al. [3]. In their approach, fine grained CFG's derived from static analysis were enforced by grouping sets of targets into equivalence classes, and marking each with a secret cookie. Indirect control-flow instructions are instrumented to check at runtime the cookie (placed as a no-op prior to the target). This enforcement scheme is less precise than ours, as any two overlapping sets of targets must be merged. In our setting, a similar technique would not be able to distinguish different subtrees of a primitive hierarchy. MCFI [28] extends Abadi's work by adding a level of indirection via runtime maps from branches and branch targets to their corresponding equivalence class. Further MCFI utilizes a thread safe dynamic update mechanism that allows control-flow graphs to be merged at runtime, thus allowing separate compilation and dynamic linking. WIT [4] similarly uses equivalence classes (colors) to protect indirect control-flow and extends this technique to protect writes as well.

CCFIR [43], HyperSafe [39] and MoCFI [9] replace code pointers in writable memory with indices/pointers into trampoline sections. CCFIR utilizes randomization to reduce the chance that an attacker can guess the index of a specific sensitive function, while HyperSafe utilizes multiple springboard sections to increase precision. Our technique could possibly be used to extend these approaches by ordering the springboard sections appropriately. As a result higher precision might be achievable without additional runtime overhead and without the need for randomization/multiple trampoline sections. This is important, as the loss of precision has for example enabled exploits of CCFIR [11].

binCFI [44] extends CFI to COTS binaries (similarly to CCFIR) by combining reliable disassembly and binary translation. Computed transfers in binCFI are rewritten to index into translation table using the candidate target, which restricts the possible control-flows.

Opaque CFI[26] combines coarse-grained CFI with code randomization to defeat attackers with full access to the process code section. Their technique covers all control transfers and achieves 4.7% overhead over a set of SPEC benchmarks. Their technique employs bound checks similarly to us, and is the first CFI technique to mention the potential for hardware acceleration of bound checks via the upcoming Intel MPX instructions[15].

C. SFI

SFI [38], [23], [40], [33], [41] is generally built on top of a coarse-grained form of CFI, usually combining instructions into aligned bundles. SFI techniques also leverage hardware techniques such as segmentation or software techniques such as masking to restrict writes to a given region. SFI has also been applied to the Chrome through the Native Client [40] project, where it provides sandboxing for parts of the browser.

D. Other Mitigation Techniques

Modern operating systems employ DEP [31] and ASLR [29] to prevent code injection attacks and increase the cost of jump-to-libc attacks. PointGuard [7] and [36] propose pointer encryption as a means to prevent attackers from accurately redirecting control-flow. In their work code pointers are encrypted (e.g. XOR-ed) using a secret key, and unencrypted prior to use. An attacker would require the secret key to accurately redirect control flow. A slew of techniques have been also proposed and deployed for protecting the stack including stack canaries [6], SafeStack [20], shadow stacks [3] and SafeSEH [25]. These techniques provide additional safety complimentary to our work but recent work [5] shows that they are not as secure as previously thought.

Kuznetsov et. al. [17] present CPI – a code pointer integrity technique that protects all data which influences control-flow. Their technique provides stronger guarantees than us and protects more computed transfers, but at a higher runtime cost 8.4% (23% on the C++ benchmarks in SPEC2006). A relaxation of CPI – Code-Pointer Separation (CPS) – provides less precise protection for virtual dispatch than us, but still covers more computed transfers (e.g. returns). CPS does not protect the integrity of pointers to code pointers, which include vptrs, and thus would allow vtable confusion attacks. CPS incurs 1.9% on average (4.2% on the C++ benchmarks in SPEC2006).

XII. Limitation and Future Work

Our approach currently protects only C++ dynamic dispatch. We believe however that it might be possible to adapt our technique so that it also checks the type safety of generic function pointers. The idea would be to use a trampoline-based technique such as CCFIR, while also laying out the trampolines in memory using a "mock" class hierarchy based on the function signatures. The merit of such an approach requires further investigation as function signatures might be too loose a safety criteria.

Another barrier to adoption that we plan on addressing is the lack of support for dynamic linking and loading. While dynamic linking could be supported through extending the runtime linker, dynamic loading is a more difficult feat. Merging the CFI policies of the modules concurrently with their code running in itself is a difficult problem [28]. In our setting this is further complicated by the ordering and interleaving we impose on pieces of data, and the immediate values in the code section that depend on it.

Another interesting direction for future work would be to adapt our technique to check C++ downcasts for safety at runtime. Exploiting bugs in programs can lead to incorrect C++ downcasts, which in turn can lead to type confusion and heap corruption. Recent work [18] has shown that C++ casts can be checked for safety with an overhead of about 7.6% on Chrome (64.6% on Firefox). Since our approach is precisely meant for checking that a vptr points to the vtable of a given class or any of its subclasses, we believe that our approach could possibly be a good fit for checking dynamic casts too. One slight caveat is that our approach would only work for classes that have virtual methods (polymorphic classes), but we believe this could be resolved using a hybrid approach: we could use our approach for classes with virtual methods, and the approach from [18] on all other classes.

Another direction for future work is protecting pointers to member functions, and more specifically checking the validity of the indices stored inside them. As already mentioned in Section X-E, our approach does not currently handle such pointers to member functions. LLVM-VCFI [22] also does not handle such pointers, but SafeDispatch [16] does, by adding an additional range check. OVT can trivially use the same check as SafeDispatch since entries for a single (primitive) vtable are still continuous. In the case of IVT this is more complicated as entries of one vtable are interleaved with entries of related subclasses and superclasses. One possible approach is to keep old vtables, and refer to them from IVT for dereferencing pointers to member functions, at the cost of additional code bloat. Another possibility is to check member pointers directly on the interleaved layout, but this would require coming up with a set of carefully crafted range and stride checks.

XIII. Conclusion

We have presented an approach for protecting the control-flow integrity of C++ virtual method calls. Our approach is based on a novel layout for vtables. Although our layout is very different from a traditional one, our layout is backwards compatible with the traditional way of doing dynamic dispatch. Most importantly, our layout allows us to check the safety of vtables using efficient range checks. We have implemented our approach in the LLVM compiler, and have experimentally evaluated its runtime and memory overhead, showing that it has lower overhead than the state-of-the-art techniques that provide the same guarantee.

Although this paper focuses on protecting dynamic dispatch, our approach could possibly be a stepping stone to more complicated forms of runtime enforcement. For example, as we have already alluded to, our approach could possibly be adapted to check the safety of C++ downcasts, or the type safety of arbitrary function pointers.

Acknowledgments

We would like to thank reviewers for their insightful feedback, and Dongseok Jang for his guidance and advice. This work was supported by NSF grant CNS-1228967 and a generous gift from Google.

References

[1] "CWE-122." Available from MITRE, CWE-ID CWE-122. [Online]. Available: https://cwe.mitre.org/data/definitions/122.html

[2] "CVE-2012-0167." Available from MITRE, CVE-ID CVE-2014-0160., 2011. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0167

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *CCS*, 2005.

[4] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, “Preventing memory error exploits with WIT,” in *S&P*, 2008, pp. 263–277.

[5] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, M. Qunaibit, and A.-r. Sadeghi, “Losing control : On the effectiveness of control-flow integrity under stack attacks,” In CCS, 2015.

[6] C. Cowan, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in *USENIX Security*, A. D. Rubin, Ed., 1998.

[7] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “Pointguard$^{\text{TM}}$: Protecting pointers from buffer overflow vulnerabilities,” in *USENIX Security*, 2003.

[8] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz, “Its a trap: Table randomization and protection against function-reuse attacks,” In CCS, 2015.

[9] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A. Sadeghi, “Mocfi: A framework to mitigate control-flow attacks on smartphones,” in *NDSS*, 2012.

[10] C. Evans, “Exploiting 64-bit linux like a boss.” http://scarybeastsecurity.blogspot.com/search?q=Exploiting+64-bit+linux, 2013.

[11] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, “Out of control: Overcoming control-flow integrity,” in *In S&P*, 2014.

[12] Google, “Heap-use-after-free in WebCore (exploitable),” https://code.google.com/p/chromium/issues/detail?id=162835, 2012.

[13] I. Haller, E. Göktas, E. Athanasopoulos, G. Portokalidis, and H. Bos, “Shrinkwrap: Vtable protection without loose ends,” in *ACSAC*, 2015, pp. 341–350.

[14] M. InfoSecurity, “Pwn2own at cansecwest 2013,” https://labs.mwrinfosecurity.com/blog/2013/03/06/pwn2own-at-cansecwest-2013, 2013.

[15] Intel, “Introduction to intel memory protection extensions,” https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, 2013.

[16] D. Jang, Z. Tatlock, and S. Lerner, “SafeDispatch: Securing C++ virtual calls from memory corruption attacks,” in *NDSS*, 2014.

[17] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *OSDI*, J. Flinn and H. Levy, Eds., 2014, pp. 147–163.

[18] B. Lee, C. Song, T. Kim, and W. Lee, “Type casting verification: Stopping an emerging attack vector,” in *USENIX Security*, 2015.

[19] LLVM Team, “The llvm compiler infrastructure project,” http://llvm.org/.

[20] ——, “http://clang.llvm.org/docs/safestack.html,” http://clang.llvm.org/docs/SafeStack.html, 2014.

[21] ——, “Llvm link time optimization: Design and implementation,” http://llvm.org/docs/LinkTimeOptimization.html, 2014.

[22] ——, “Control flow integrity design documentation,” http://clang.llvm.org/docs/ControlFlowIntegrityDesign.html, 2015.

[23] S. McCamant and G. Morrisett, “Evaluating SFI for a CISC architecture,” in *In USENIX*, 2006.

[24] Microsoft, “Vulnerability in Internet Explorer could allow remote code execution,” http://technet.microsoft.com/en-us/security/advisory/961051, 2008.

[25] Microsoft Visual Studio, “Image has safe exception handlers,” http://msdn.microsoft.com/en-us/library/9a89h429%28v=vs.80%29.aspx, 2005.

[26] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, “Opaque control-flow integrity,” in *NDSS*, 2015.

[27] H. D. Moore, “Microsoft Internet Explorer data binding memory corruption,” http://packetstormsecurity.com/files/86162/Microsoft-Internet-Explorer-Data-Binding-Memory-Corruption.html, 2010.

[28] B. Niu and G. Tan, “Modular control-flow integrity,” in *PLDI*, 2014.

[29] PaX Team, “Pax address space layout randomization (aslr),” http://pax.grsecurity.net/docs/aslr.txt, 2003.

[30] A. Prakash, X. Hu, and H. Yin, “vfguard: Strict protection for virtual function calls in COTS C++ binaries,” in *NDSS*, 2015.

[31] V. A. S. Andersen, “Data execution prevention: Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies,” http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[32] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications,” in *S&P*, 2015, pp. 745–762.

[33] C. Small, “Misfit: A tool for constructing safe extensible C++ systems,” in *USENIX Conference on Object-Oriented Technologies (COOTS)*, S. Vinoski, Ed., 1997, pp. 175–184.

[34] Symantec, “Microsoft Internet Explorer virtual function table remote code execution vulnerability,” http://www.symantec.com/security_response/vulnerability.jsp?bid=54951, 2012.

[35] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing forward-edge control-flow integrity in GCC & LLVM,” in *NDSS*, 2014.

[36] N. Tuck, B. Calder, and G. Varghese, “Hardware and binary modification support for code pointer protection from buffer overflow,” in *37th Annual International Symposium on Microarchitecture (MICRO-37*, 2004, pp. 209–220.

[37] VUPEN, “Exploitation of Mozilla Firefox use-after-free vulnerability,” http://www.vupen.com/blog/20120625.Advanced_Exploitation_of-Mozilla_Firefox_UaF_CVE-2012-0469.php, 2012.

[38] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *SOSP*, 1993, pp. 203–216.

[39] Z. Wang and X. Jiang, “Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity,” in *S&P*, 2010.

[40] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native client: A sandbox for portable, untrusted x86 native code,” in *S&P*, 2009, pp. 79–93.

[41] B. Zeng, G. Tan, and G. Morrisett, “Combining control-flow integrity and static analysis for efficient and validated data sandboxing,” in *CCS*, 2011, pp. 29–40.

[42] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, “Vtint: Protecting virtual function tables’ integrity,” in *NDSS*, 2015.

[43] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical control flow integrity and randomization for binary executables,” in *S&P*, 2013, pp. 559–573.

[44] M. Zhang and R. Sekar, “Control flow integrity for COTS binaries,” in *USENIX Security*, 2013, pp. 337–352.