NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking

Markus Bauer
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
markus.bauer@cispa.saarland

Christian Rossow
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
rossow@cispa.saarland

***Abstract*—The vast majority of nowadays remote code execution attacks target virtual function tables (*vtables*). Attackers hijack vtable pointers to change the control flow of a vulnerable program to their will, resulting in full control over the underlying system. In this paper, we present NoVT, a compiler-based defense against vtable hijacking. Instead of *protecting* vtables for virtual dispatch, our solution *replaces* them with `switch-case` constructs that are inherently control-flow safe, thus preserving control flow integrity of C++ virtual dispatch. NoVT extends Clang to perform a class hierarchy analysis on C++ source code. Instead of a vtable, each class gets unique identifier numbers which are used to dispatch the correct method implementation. Thereby, NoVT inherently protects all usages of a vtable, not just virtual dispatch. We evaluate NoVT on common benchmark applications and real-world programs including Chromium. Despite its strong security guarantees, NoVT *improves* runtime performance of most programs (mean overhead −0.5%, −3.7% min, 2% max). In addition, protected binaries are slightly smaller than unprotected ones. NoVT works on different CPU architectures and protects complex C++ programs against strong attacks like COOP and ShrinkWrap.**

***Index Terms*—NoVT, vtables, vtable hijacking, LLVM, CFI**

1. Introduction

The most popular programs are still written in memory-unsafe languages like C++ nowadays [1]. For decades, attackers have exploited the resulting memory vulnerabilities to compromise systems, e.g., abusing well-known vulnerabilities like stack buffer overflows. Program vendors have reacted by employing defenses such as `W⊕X`, ASLR [2] and stack canaries [3]. As a reaction, attackers are shifting their effort to mount *code-reuse* attacks exploiting heap errors. Browsers are a particularly rewarding and valuable target for heap-based code-reuse attacks, because browsers contain a massive amount of C++ code and browsers include a scripting engine that allows adaptive attacks. For example, 70% of all vulnerabilities in Google Chrome are memory safety problems [4]. The problem is not specific to browsers, though. In fact, code-reuse attacks are fairly common also in other popular working environments. For instance, use-after-free bugs are the main source of vulnerabilities in Windows [5].

A fundamentally important attack step in exploiting these vulnerabilities is *vtable hijacking*. Attackers commonly use heap corruption vulnerabilities (like a *use-after-free* bugs) to corrupt a C++ object. C++ objects contain the address of a *vtable*, which is a read-only list of function pointers, pointing to all methods of the given object. Using the memory corruption, an attacker can overwrite the address to an object's vtable with the address of an attacker-controlled memory structure. This memory structure is filled with a faked vtable containing arbitrary function pointers. As soon as the program uses the vtable to retrieve a method pointer, the attacker has full control over the instruction pointer.

Unfortunately, C++ programs contain numerous vtables as compilers use them to implement core language features of C++: class inheritance and virtual methods. The attack surface is significant. Each call of a virtual method relies on a vtable pointer, and some programs use millions of virtual method calls per second. Even worse, researchers showed that vtable hijacking attacks are Turing-complete, even if only existing vtables can be re-used (the COOP attack [6]).

As a consequence, researchers aimed to defeat vtable hijacking attacks, either with compiler-based program modifications or binary rewriting tools. Previous work [7]–[16] protects vtable-based virtual dispatch with security checks added to the program: before a virtual method is called, either the vtable or the method pointer in the referenced vtable are checked for validity. The set of valid vtables or methods is determined by the C++ class hierarchy and type system. However, these additional checks come with a performance penalty that is potentially non-negligible [8], [13]–[17]. Furthermore, recent work [6], [18] introduced new ways to bypass many existing defenses, including [8], [9], [11], [13], [14], [16], [17]. Most proposed solutions only protect vtable-based virtual dispatch, but not other usages of the vtable (e.g., virtual offsets and runtime type information) [7], [9], [10], [12], [19]. Finally, and quite surprisingly, no prior work has attempted to solve the root cause of vtable hijacking: the mere *existence* of vtables.

In this work, we radically change the way how C++ member functions are dispatched. Like prior work, we also observe that all possible class types can be determined at compile time [7], [8], [10], [17], [20], [21] if no dynamic linking is required. However, instead of following the well-explored idea of *protecting* vtables accordingly, we show that vtables can be *eliminated*, which tackles the root cause of vtable hijacking. Given the source code of a C++ program, we leverage the class hierarchy to eradicate vtables and to restrict virtual dispatches to the

minimal valid set of methods per call site. Technically, we replace vtables with class-unique identifiers (IDs). Instead of vtable pointers, class instances contain an ID that determines their class (dynamic type). Whenever a virtual method is dispatched on an object instance, we load the ID from the instance and build a `switch(ID) case` construct that calls the respective method for a given ID. For each virtual call, we only handle IDs of classes that are possible by its static type, effectively preventing function-wise code reuse attacks [6]. While this sounds inefficient at first, we actually *improve* the runtime performance of most tested programs. As a side effect, unlike most related works, we protect *all* constructs that relied on vtables before: virtual offsets, runtime type information, and dynamic casts. Our approach is generic, agnostic to the operating system and system architecture, and is applicable to other compilers and ABIs.

We present a prototype of our protection—NoVT (*No VTables*). NoVT is implemented in Clang/LLVM 10, based on the Itanium ABI. While still a prototype, NoVT can handle complex programs up to million lines of code. We evaluate NoVT on the SPEC CPU 2006 benchmark, Chromium, MariaDB, Clang, CMake and Tesseract OCR, where it has a mean runtime overhead of −0.5%, 2% worst case (i.e., speeds up programs on average). Binaries protected by NoVT are slightly smaller than unprotected ones. NoVT's protection level is optimal (as defined in ShrinkWrap [18]), protects against code-reuse attacks like COOP [6] and is applicable to any valid C++ program without code changes (given all linked C++ source code). Our prototype has been released as open-source software[1].

To summarize, our contributions are:

- We show that all C++ semantics can be implemented without relying on vtables.
- We introduce NoVT, a LLVM-based prototype that protects C++ programs by removing vtables. NoVT safeguards even complex programs against vtable hijacking, including Chromium (29.7 MLoC), LLVM / Clang (3 MLoC) and the C++ standard library (437 KLoC).
- The level of protection offered by NoVT is optimal and complete. NoVT also protects every use of vtables beyond dynamic calls, including virtual inheritance offsets that are vital for field access and object casts.
- NoVT shows *negative* performance overhead for most tested programs, and is thus the first vtable "protection" scheme that does not slow down the majority of programs. At the same time, NoVT also reduces binary size.

2. Background

Inheritance in C++ can become quite complex, as C++ supports features like multiple inheritance and virtual bases. We therefore start by providing relevant background information on these C++ details. The code in Figure 1 will serve as running example to illustrate how C++ handles classes with inheritance.

1. https://github.com/novt-vtable-less-compiler/novt-llvm

```
class NV {
    // 8 bytes, no virtual methods
    uint64_t nv1;
};
class A {  // 24 bytes
    uint64_t a1, a2;
    virtual void f();
};
class B : public NV, public virtual A {
    // 72 bytes (40 + 8(NV) + 24(A))
    uint64_t b1, b2, b3, b4;
    virtual void g();
    virtual void g2();
};
class C : public virtual A {
    // 56 bytes (32 + 24(A))
    uint64_t c1, c2, c3;
    virtual void h();
    virtual void h2();
};
class D : public B, public C {
    // 120 bytes (24 +48(B) +32(C) +24(A))
    uint64_t d1, d2;
    void f() override;
    void g() override;
    void h() override;
};
```

Figure 1. C++ code as running example with multiple (classes B and D) and virtual inheritance (classes B and C)

2.1. C++ Inheritance and Vtables

C++ classes can have *virtual methods* which can be overwritten by child classes. Calling a virtual method on a class instance invokes the method defined by the actual dynamic type at runtime, regardless of the type in source code (the static type). That means when we call `g()` on a pointer of type `B*`, either `B::g()` or `D::g()` can be executed, depending on the dynamic type of the instance. To dispatch virtual functions, all major C++ compilers use virtual function tables (*vtables*). A vtable is an array of function pointers of all methods of a class, and possibly additional information. Each class instance (with virtual methods) contains a pointer to the vtable of its class. When a virtual method is dispatched, the compiler emits code that loads the vtable pointer, fetches the function pointer from the table, and finally, calls this pointer.

2.2. C++ Multiple Inheritance

In C++, a class can have multiple base classes (*multiple inheritance*). Figure 2 shows the inheritance in our example, where classes B and D inherit from multiple classes. A and NV are base classes, C inherits only from A. A is always inherited *virtual*, so that D includes one copy of A, B and C each. Consequently, the four classes A–D form an "inheritance diamond", a common problem in languages with multiple inheritance.

The compiler computes the memory layout of a class according to the Itanium ABI [22]. Derived classes always include their base (parent) classes in memory. That is, the memory representation of a class instance always starts with its primary vtable pointer, followed by all direct

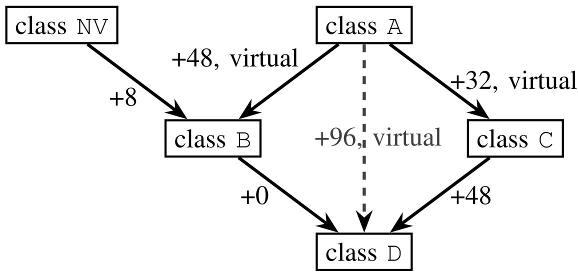

Figure 2. Class hierarchy graph for our running example (Figure 1). The given offsets relate to the memory layout in Figure 3 (e.g. NV starts at offset +8 in B). The dashed line represents *indirect* virtual inheritance.

base classes in order of inheritance, and last, all fields defined by this class are appended. Figure 3 shows the memory layout of our example classes. If an instance of type `D*` is cast to `B*` and method `g()` is dispatched, the generated code would look up that method in `D`'s vtable and call `D::g` with the correct *this* pointer (pointing to the `D` object). But if an instance of type `D*` is cast to `C*` and method `h2()` is dispatched, we would face two problems: First, the vtable layout of `B` and `C` is incompatible, because `g2` is stored at index 1 in `B`'s vtable, while index 1 in `C` is `h2()`. That means we cannot dispatch both `g2` and `h2` with the same vtable. Second, *this* would not point to an instance compatible with type `C`—it points to an instance of type `D`, which starts with fields `b1`-`b4`, not with `c1`-`c3` as expected by `C::g2`. To mitigate this problem, Itanium requires *secondary vtables*. The instance of `D` contains a second vtable pointer at offset 48 (the beginning of the `C` structure in `D`). When we cast `D` to `C`, we correct the pointer by this offset so that it now points to the secondary vtable, which gets a `C`-compatible layout, and methods inherited from `C` can be dispatched without additional effort. Methods overwritten by `D` get a special vtable entry that moves *this* back to the beginning of `D` before calling `D`'s implementation.

2.3. C++ Virtual Inheritance

With the sketched inheritance model, multiple copies of a C++ base class can be included in a derived one. This is counter-intuitive and not always desired. To solve this, C++ provides *virtual* inheritance. A virtual base class is always included only once, no matter how many base classes inherit from it. In our example, `A` is a base of both `B` and `C`, which in turn both are bases from `D`, but `D` includes `A` only once. To this end, the memory layout of a class with virtual bases is different: Instances of virtual bases are included at the end of the final class and deduplicated.

The computed memory layout (Figure 3) shows that the memory offsets between a class and its base are no longer constant. When method `B::g2` accesses the field `A::a1` on an instance of `B`, it knows that this field is at offset 56 (*this*+56) because `A` is at offset 48 and `a1` is at offset 8 in `A`. But if the same method is inherited to `D`, `A::a1` is at offset 104 (`A` starts at offset 96 in `D`). The same problem occurs when casting between these types. Again, C++ solves this issue with vtables. In addition to function pointers, a vtable contains the memory offset to

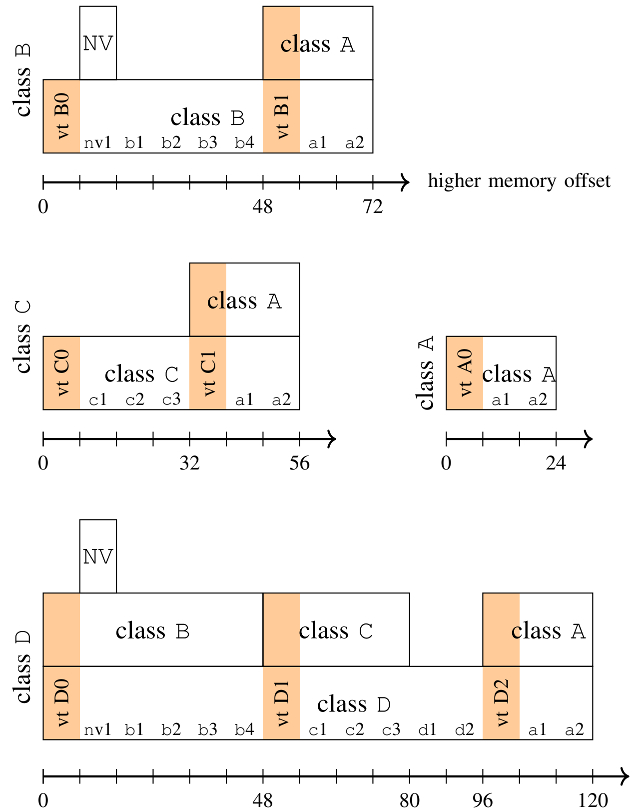

Figure 3. Memory layout of our example classes according to the Itanium ABI. Vtables B0 and D0 have a B-compatible layout. Vtables A0, B1, C1 and D2 have an A-compatible layout. Vtables C0 and D1 have a C-compatible layout.

all its virtual bases and virtual children (including indirect ones). When a field in a virtual base is accessed or a cast occurs, the compiler first loads the pointer to the object's vtable, then loads the *virtual offset* from the (known) index of the vtable, and finally, computes the address of the base or children as *this*+*virtual offset*. To this end, vtables can also contain a pointer to meta information about the class, so-called *runtime type information* (*RTTI*), which are used for exceptions and dynamic casts.

To build (and destroy) class instances, compilers may need to use so-called *construction vtables*. These transient tables contain the virtual offsets of a child class, but the inherited methods of a parent class. When constructing the child class, they ensure that no method on the child is called before the parent classes are fully initialized. Furthermore, construction vtables guide the objects to use the correct virtual offsets.

2.4. Vtable Hijacking

As vtables contain function pointers, they are a valuable target for memory corruption. To hijack control flow, an attacker can modify the vtable pointer stored in the first bytes of an instance, e.g., by exploiting a heap overflow or use-after-free vulnerability. Usually, the attacker overwrites the vtable pointer with an address of an attacker-controlled memory region. Then, the attacker fills this memory region with a pointer to the code they want to execute, creating a faked vtable. When a virtual method

is dispatched on the tampered class instance, the program loads a function pointer from attacker-controlled memory and calls it. By tampering with a single vtable pointer, the attacker can thus leverage a potentially small memory corruption to execute attacker-controlled code.

Researchers showed that attacks do not necessarily need fake vtables: "Counterfeit Object-Oriented Programming" (COOP) [6] attacks lead to turing-complete code execution by chaining pointers to existing, valid vtables from a sufficiently large program. Similar to ROP, several gadgets in form of faked objects with pointers to actual vtables are stored in an attacker-controlled buffer. Later a loop with a virtual dispatch inside is used to dispatch these gadgets one by one. Again, if this loop is in a virtual function, regular vtable hacking (with any vtable containing this virtual function) is used to start the attack. COOP breaks naïve protections that just check if vtable pointers actually point to a vtable, because all vtable pointers point to original vtables, from other classes.

However, virtual dispatch is not the only security-critical operation on vtables—a fact that several related works dismiss. An attacker can overwrite the vtable pointer with a pointer to attacker-controlled memory and change the virtual offset there. Whenever a method or field inherited from the base class is used from the manipulated object, the attacker has full control over the *this* pointer of that method or the address of that field, even if the invocation is non-virtual. With this power, common methods like attribute getters and setters can be turned into arbitrary memory read and arbitrary memory write primitives, while the control flow of the program stays unaltered. Therefore, we argue that a strong vtable protection must also protect virtual offsets in addition to virtual dispatch.

3. Attacker Model

We want to mitigate memory corruption attacks against C++ class instances in which an attacker aims to divert control to an arbitrary function outside of the instance's scope. Hereby, we explicitly include COOP-style attacks [6], i.e., we also aim to prevent the reuse of existing, yet arbitrary virtual methods. Furthermore, we aim to protect virtual offsets from arbitrary modifications, as outlined in Section 2.4. We assume that the attacker has arbitrary read and write access to the heap and other memory locations that contain object instances, and they know about the program's memory layout.

We assume that all executable pages are non-writable (`W⊕X`) and that return addresses are protected by other means (e.g., canaries, shadow stack, etc.). In line with all related works in this context, we ignore C-style function pointers that might even occur in C++ programs.

Summarizing, our threat model reflects common remote code execution attacks such as against Javascript engines in off-the-shelf browsers.

4. Design and Implementation

Our protection scheme NoVT removes vtables from a C++ program and replaces them with constant identifiers that are used for dispatching virtual calls. Whenever a vtable was used before, we generate a `switch-case` struct that dispatches the minimal set of possible identifiers, as determined by static types, and aborts execution otherwise. This section outlines our overall methodology. In Sections 4.1 and 4.2, we explain how we use *class hierarchy analysis* and a *class identifier graph* to determine the set of valid methods for each virtual call. In Sections 4.3 and 4.4, we show how we create dispatch functions and class identifiers to replace vtables. In Section 4.5, we show how we optimize the resulting structure to improve performance. In Sections 4.6, 4.7 and 4.8 we show how we build our prototype NoVT as a fork of Clang 10.

4.1. Class Hierarchy Analysis

To replace vtable-based virtual dispatch, we first need to learn the class hierarchies [20] of the program we want to protect. To this end, we add metadata to each class that has at least one virtual method, a virtual base or inherits a class with virtual methods or virtual bases. Classes without any virtual methods or inheritance cannot be used in virtual dispatch or virtual inheritance, and would not contain a vtable pointer anyways (class `NV` in our example). We ignore them in the remainder of this paper, as they thus never undermine security according to our threat model. For all other class, we record their virtual and non-virtual bases including the memory offset between the derived class and its base, and the defined virtual or overridden methods. We also store a reference to their vtables (which will be removed in a later step) and to all generated construction vtables (including layout class and memory offset from base class to layout class). To avoid name clashes and to support C++ templates, we mangle all class names according to the Itanium ABI [22].

From the stored inheritance information, we can construct a class hierarchy graph at link time (see Figure 2). Each node in this graph represents a class, each edge represents an inheritance path. Each edge is marked with the memory offset between both classes, i.e., the memory location of the base class inside the derived memory layout. When casting between these classes or dispatching inherited methods we need to correct the pointer (*this*, etc.) for this offset.

4.2. Class Identifiers

From the class hierarchy graph, we then determine the necessary class identifiers we need to create. Each class identifier will later replace a primary or secondary vtable, i.e., a class can have multiple class identifiers. Each class identifier is a pair $ID = (cls, o)$ where cls is a class and o is a memory offset. The memory offset denotes that identifier ID will later be written at offset o bytes from the start of the instance. If we read an identifier from an instance pointer and know its offset, we can compute the beginning of the instance (to adjust *this*). Class identifiers with offset 0 signal the beginning of an instance (like a primary vtable), offset $\neq 0$ have the same purpose as secondary vtables.

For objects in construction, we use combinations of two class identifiers. The first identifier denotes the class under construction and is later used to dispatch virtual methods. The second identifier denotes the class whose memory layout is applied, and is later used to resolve

virtual offsets. In our example, we need a construction identifier $((\texttt{C}, 0), (\texttt{D}, 48))$ ("C-in-D") while constructing the C instance in D. It denotes that the virtual methods from C should be used, but if fields from virtual base A are accessed we have to respect D's memory layout (e.g., A starts at +48 bytes, not at +32 as in C). Construction identifiers avoid that overridden methods of an object are called while the object is not fully constructed, they are similar to *construction vtables* in Itanium.

To this end, we construct a *class identifier graph* (see Figure 4 for an example). Each node in this graph is a class identifier, edges between identifiers show the inheritance between their classes: Two nodes (c_1, o_1) and (c_2, o_2) are connected with an edge $(c_1, o_1) \longrightarrow (c_2, o_2)$ iff i) c_1 is a base class of c_2, ii) casting from a c_2-pointer to a c_1-pointer modifies the pointer by $(o_2 - o_1)$ bytes. In practice that means: If offset o_2 in a class of type c_2 refers to a field inherited from c_1 at offset o_1, then these identifiers are connected. In our example, $(\texttt{A}, 0)$ and $(\texttt{B}, +48)$ both refer to the beginning of an A instance, therefore they are connected. $(\texttt{A}, 0)$ and $(\texttt{D}, +48)$ are not connected because A starts in D at offset $+96$.

We generate these class identifiers by traversing the class hierarchy graph top-down, e.g., bases (parents) before derived classes. Created identifiers are marked *virtual* if they have been generated using virtual inheritance, *non-virtual* otherwise. This marking is vanished after all identifiers have been created. For each class c, we create identifiers in these steps:

1) We create one default identifier representing the class at offset 0: $(c, 0)$. We mark this identifier as non-virtual.
2) We traverse all identifiers $id' = (c', o')$ of all base classes c' (given the memory offset o between c and c'):
   - If the identifier is marked non-virtual (i.e., it has not been created using any virtual bases), we create an identifier $(c, o + o')$ and connect it with an edge from id'. This new identifier is marked virtual iff c' is a virtual base of c.
   - If the identifier is marked virtual, we determine its root $(c'', 0)$ in the graph (a unique node generated by rule 1). Next we determine the virtual offset o'' of c'' in c. Finally, we create a new identifier (c, o'') and connect it with an edge from id'. The new identifier is always marked virtual.
3) For each construction vtable c-in-c' of the class c, we traverse all identifiers $id' = (c, o)$ of this class. For each identifier we compute the memory offsets o_1 (between $c + o$ and $c + 0$) and o_2 (between $c + o$ and $c' + 0$). We create a construction identifier $((c, o_1), (c', o_2))$ and connect it with an edge from id'.

There is a strong relation between vtables and class identifiers. Every class identifier corresponds to exactly one vtable, connections between identifiers imply that these vtables can be expected in the same location for a given (static) type. We can clearly see this connection

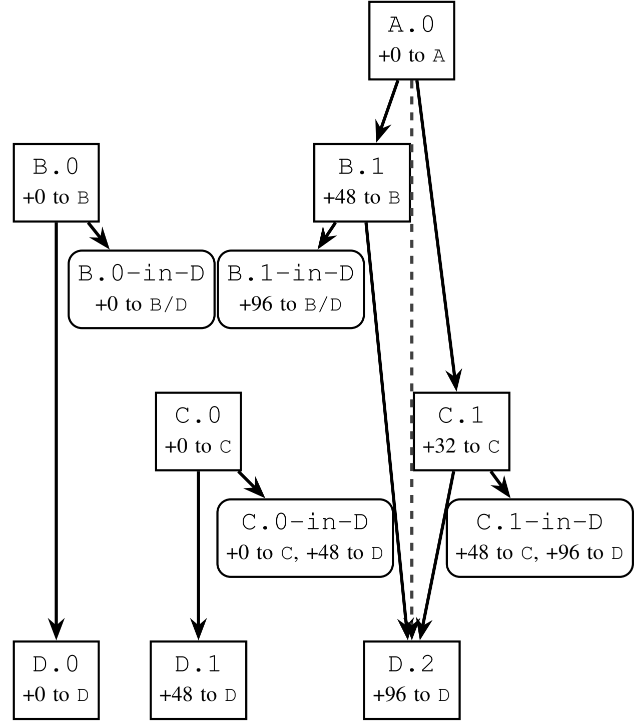

Figure 4. The class identifier graph of our example, including all possible construction identifiers (rounded rectangles).

when we compare the class identifier graph from Figure 4 with the vtables in Figure 3: Class B has two vtables: one primary vtable and one that has layout of class A's vtable (e.g., B cast to A). These two tables correspond to the class identifiers B.0 and B.1. Given a pointer with a static type of A, it points to an instance starting with a pointer to the primary vtable of A or the secondary vtable of B, therefore class identifiers A.0 and B.1 are connected. Similar for D: class D has three vtables that correspond to the class identifiers D.0, D.1 (D casted to C) and D.2 (D casted to A). Similarly all construction identifiers correspond to a construction vtable. We will use this connection later and replace vtable pointers with a unique number per class identifier. The edges between class identifiers help us to enumerate all possible class identifiers that can occur for a given static type.

4.3. Dispatch Function Generation

While a vtable-based dynamic dispatch can be compiled independent of other compile units, this is no longer possible for our protection (and, in general, for any protection that relies on class hierarchy analysis). To dispatch a function, any approach based on a class hierarchy analysis needs to know all possible functions that can be called, which might not even be declared in the current unit. To counter this and allow incremental compilation, we compile dynamic dispatch in two steps. When compilation requires a dynamic dispatch, we declare a *dispatch function* instead and replace the dynamic dispatch with a (static) call to this dispatch function. The generated dispatch function has exactly the same type as the virtual function we want to call. It is annotated with the static type of the object we dispatch on and the method name we

need to dispatch. To account for templates and overloaded functions, we mangle this method name according to the Itanium ABI [22].

We define these methods later in the linking phase. For each method, we identify the `this` argument and start with loading the class identifier using that pointer. We then create a `switch-case` structure (with LLVM's `SwitchInst`). Each possible class identifier is handled by a case in that switch instruction. We can get all possible class identifiers by traversing the class identifier graph, starting with the static type's primary class identifier. For example, when dispatching on class `C` we would start with identifier `C.0` and traverse `D.1` and `C.0-in-D`. For each case, we can determine the method to be called by the class identifier's (dynamic) type using standard inheritance rules. We emit a (static) call to this method and return its result. To catch potential memory corruptions, we emit a LLVM Trap intrinsic (x86's `ud2`) in the default case of the switch. If an invalid class identifier occurs at runtime, the program crashes to avoid control flow hijacks. The order of the traversal here is not important. During assembly generation, LLVM will reorder all emitted cases depending on the assigned identifier number. Also in the rare cases where LLVM does not reorder checks the order has a negligible effect on performance. We use the same method to implement other constructs that used to rely on vtables: Whenever the compiler needs to know the virtual offset of a virtual base, we emit a call to an *offset function* (typed pointer → word-sized int). This function is later defined by a `SwitchInst` that simply returns the correct offset for all possible class identifiers. Dynamic casts that cannot be resolved at compile-time are replaced with a *cast function* (typed pointer → pointer). For each possible class identifier, this function returns either the pointer (potentially with an offset) or `nullptr` if the class cannot be cast. Last, we replace every access to runtime type information with a call to an *rtti function* that returns a pointer to the correct RTTI structure when called.

4.4. Storing Class IDs and Removing Vtables

The mechanisms described until now already replace all C++ concepts that normally rely on vtables. We finally can delete the vtables themselves, and replace the instance's pointers to them with the class identifier that corresponds to a given vtable. To this end, we assign a unique number to every class identifier in the class identifier tree. Next, we find direct usages of the vtable, which happens only in class constructors (when the vtable's pointer is written to the instance). We replace that vtable pointer with the corresponding class identifier's number and write it into the memory slot that used to store the vtable pointer. The memory layout of classes does not change by this modification, i.e., we do not introduce memory overhead with this step. All dispatcher functions (and friends) load this identifier number later from the vtable pointer slot and use it as described. Finally, having removed all references to the vtables, they will be removed by a following *Dead Globals Elimination* pass.

4.5. Optimizations

We extend this basic methodology with optimizations that boost the runtime efficiency of our protection.

4.5.1. Dead Class Identifiers. Class identifiers that are never used in a generated function, or class identifiers whose identifier number is never written in a constructor can be safely removed. Removing them saves some branches in the generated functions as well as it allows for a more dense numbering. We evaluate optimizations on the C++ programs from SPEC CPU 2006 and Chromium. Applied after all other optimizations from this section, we can on average remove 29% of all class identifiers because they are never used in a constructor, and additional 22% because they are never used in a generated function.

4.5.2. Merge Cases. In the generated functions, many cases can be merged, for example, when multiple classes inherit the same method without overriding it. Whenever cases execute the same action (same method dispatched, return same virtual offset, etc.) we merge them into one case. This reduces code size and improves later optimization results. In our experiments, we could remove 24% of all case handlers on average.

4.5.3. Devirtualization. Whenever a generated function has only one possible case (possibly after combining cases, excluding the error case), we can statically determine the only legal path and remove the switch-case around it. Only a single static call or single constant offset remains, the virtual operation has been statically resolved (devirtualized). We mark the generated function for inlining to get rid of any performance impact. On average, 57% of all generated functions in our experiments can be devirtualized.

4.5.4. Merge Identifiers. We look for pairs of class identifiers that trigger the same behavior in all generated functions and merge them to a single identifier, thus reducing the total number of identifiers. These "equal" identifiers occur, for example, when a subclass does not overwrite any methods, or when a construction identifier is rarely used. We search these equal identifiers by iterating all generated functions: whenever a pair of two identifiers is handled in the same function (i.e., occurs in the same `SwitchInst`) and the two identifiers trigger the same behavior, we mark them as "potentially equal". We mark them as "not equal" if they trigger different behavior and must not be merged. In a second step, we check all pairs: If two identifiers are marked "potentially equal" but no "not equal" marking has been set, they can be safely merged without changing program semantics. This holds because the only (legal) usage of class identifiers are the generated functions, and there is no case where switching between the two would change the behavior of a generated function. The "potentially equal" and "not equal" relations are propagated to the merged identifiers, hence more than two identifiers will be merged if possible. With this optimization, we can merge 10% of all identifiers on average in our experiments.

4.5.5. Optimizing Identifier Numbers.

The performance of our protection is dominated by the efficiency of the compiled switch instruction. The efficiency of the generated code depends mainly on the number of cases, the density of the numbers and the maximal size of the numbers. Dense packs of numbers can be handled by more efficient constructions (e.g., jumptables) than sparse sets. We thus set out to assign consecutive numbers to identifiers that are used in the same dispatch function(s).

To this end, we first group identifiers that are used in the same generated dispatch function and trigger different behaviors there. We then assign numbers to each group independently. All identifiers in a group need a different identifier number to be distinguishable. This is not the case for identifiers in different groups. Such "colliding" identifiers are never used together and therefore do not have to be distinguishable. Consequently, each of the disjoint groups will have its own, independent numbering starting with 0. Hence, unrelated identifiers from different groups can have the same number. Assigning a number multiple times does neither harm correctness nor security. Whenever such a duplicate number occurs in the program, it is absolutely clear from the context to which of the identifiers it belongs.

We use a non-optimal recursive algorithm (shown in Appendix A / Algorithm 1) to assign numbers to a group of identifiers. For each set of identifiers larger than some threshold, the algorithm splits it into two subsets that will receive consecutive numbers recursively. One of the subsets is the maximal subset used in a generation function. Assuming a hierarchical structure of used identifier sets in our generated functions, this algorithm tries to follow that hierarchy. Our experiments show that this non-optimal algorithm significantly (50% on average) compresses the identifier space.

4.6. Implementation

We have implemented the NoVT prototype on top of Clang 10 and LLD 10. NoVT modifies C++ programs during compilation and linking. In the following, we will detail our implementation choices for both phases.

In the compilation phase, our modified compiler adds information about classes, inheritance and virtual methods as metadata to its output. We replace virtual dispatches by a call to a *dispatch function*. Similarly, we replace virtual offset loads by a call to a *offset function*. The same holds for `dynamic_cast` (*cast functions*) and RTTI loads (*rtti functions*). We enforce link-time optimization (LTO) in our compiler, so the output is always LLVM bitcode ready for further analysis. Vtables are still emitted, they will be removed later.

In the linking phase, we generate the newly introduced functions in a compiler pass introduced to `lld` or LLVM's `gold` plugin. First, we combine the metadata from all compilation units and reconstruct the full class hierarchy of the program. From this class hierarchy, we build an *identifier graph*, a structure that is similar to vtables and their inheritance relations. Using this graph, we assign IDs to all classes and change their constructors to write that ID to each class instance, replacing the vtable pointer. For each dispatch function, we determine the set of possible call targets from the identifier graph, and insert a `switch` statement that can call exactly these candidates. Similarly, we generate all necessary offset, cast and rtti functions. We optimize our identifier graph and the generated functions to counter possible performance overhead caused by the protection. Finally, the compiler's code generation applies off-the-shelf compiler optimizations.

4.7. Compiler-Assisted Optimizations

LLVM runs its optimizations that interact with our generated code and further improve the performance. First of all, LLVM's *Dead Global Elimination* pass eliminates all (now unused) vtables. Additionally, it identifies now unused methods and RTTI structures and eliminates them, which was not possible before our transformation. We evaluate the positive effect of this program size reduction in Section 5.4.

Next, LLVM can decide to inline some or all of our generated functions, especially if they are short or only called from few locations. On the other side, short methods might be inlined into our generated function. Both inlining operations save us a `call` instruction. In the best case, our virtual dispatch is compiled without a single assembly `call`. Inlining gets especially performant if LLVM can infer the result of the identifier number `Load` instruction using constant propagation (for example, if a constructor has been inlined before). LLVM has a pass that allows even interprocedural constant propagation, on our transformed program this optimization is able to devirtualize some further callsides. Finally, LLVM uses *tail calls*: when we call a regular method from a dispatcher function, LLVM emits a `jmp` instead of a `call`. As the dispatcher function and method have the same signature, the method can return instead of our dispatcher function later. Tail call optimization saves us a second `call`, a second `ret` and the space of a return address on the stack.

Last but not least, LLVM has several tweaks to efficiently implement a `SwitchInst` in assembly. The most well-known trick is a *jumptable*: Given numbers close to each other, LLVM uses that number to load the address of the next instruction and jump to it. Jumptables are similarly efficient than vtables (a `cmp` instruction more), but they are memory-safe because the jumptable index is bounds-checked before usage. Jumptables are scalable, their performance does not depend on the number of possible cases. Alternatively, LLVM can translate functions with only a few cases to a chain-like or tree-like structure of compares and jumps. These structures do not have the overhead of an additional memory access, given a small number of possible cases they are usually faster than jumptables (or regular vtable-based dispatch). Another trick is to use bitstring tests to check for many cases at once. Given many numbers that fall to a single case (e.g., many classes inheriting a non-overridden method), LLVM uses the bittest instruction (`bt`) to select a bit from a precomputed 64-bit word. If the selected bit is 1 the case is correct.

4.8. Usability

For convenience, we modified Clang to use LTO by default, to use our modified `lld` linker by default,

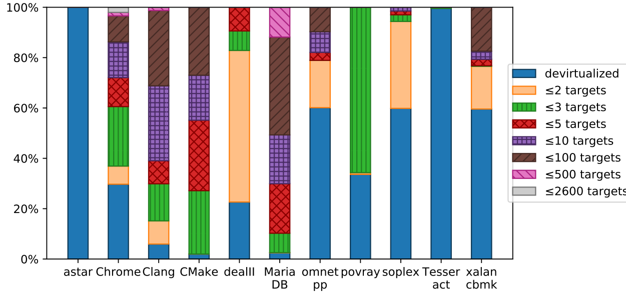

Figure 5. Virtual dispatchers broken down by their number of possible call targets.

and to include an additional library search path containing a pre-compiled protected `libstdc++`. With these small changes, our modified Clang is a drop-in replacement for typical compilers (real Clang, mainly compatible with g++). For most build systems, we only need to set an environment variable to get protected binaries (`CXX=novt-clang++`).

5. Evaluation

We now evaluate NOVT to see its impact on the programs to protect. Most important, we aim to assess the provided security level and the performance overhead. Next, we also want to know the impact on binary size and the limitations that come with this protection.

For our evaluation we use all C++ programs from the standardized SPEC CPU 2006 benchmark [23], namely *astar*, *dealII*, *namd*, *omnetpp*, *povray*, *soplex* and *xalancbmk*. Using that benchmark allows us to compare our results against other solutions (using the same benchmark). *namd* is a bit special because it does not use virtual inheritance nor virtual dispatch. We still include it in our evaluation to show the absence of any impact of our approach if no protection is necessary. To demonstrate scalability and practicability we also evaluate NOVT on Chromium 83 [24] (Google Chrome), which consists of 29.7 million lines of C/C++ code, many dependencies and is highly relevant in practice. Additionally Chromium does not tolerate any slight error in the C++ language semantics and therefore is a good test case to show that our approach does not break programs. To show a broad compatibility with a variety of popular C++ projects, we evaluate NOVT on MariaDB (a SQL database server with 2.4 MLoC), CMake (340 kLoC), Clang/LLVM 9 (2.9 MLoC) and Tesseract OCR (300 kLoC).

To get comparable results, all programs have been compiled with full protection (including a protected C++ standard library) and full link-time optimization enabled. Reference is always the same program compiled with unmodified Clang 10, full LTO and a statically linked LTO-ready version of the C++ standard library. For SPEC CPU 2006 we use all available optimizations (`-O3`), for Chromium we enable all provided optimization options in its build system. Other programs use their respective defaults for a release build.

5.1. Security Evaluation

NOVT protects any vtable-related operation against a memory-modifying attacker. This includes virtual memory offsets, dynamic casts and rtti access. We also protect member function pointers. Manual inspection of the generated dispatchers show that even concurrent memory modifications can not give an attacker more control about the instruction pointer, no intermediate values are stored on the stack. Our protection restricts the callable methods at each virtual dispatch location to the minimal set that would still be allowed by the type system, depending on optimizations even less. Haller et al. [18] showed that this restriction is *optimal* without (potentially expensive) context analysis.

This implies of course that an attacker can trigger other, possibly unintended methods by overwriting the type ID in memory, as long as these methods are still allowed by the type system. However, modifying the stored identifier number does not give full control over the instruction pointer, and the number of callable functions is considered low enough to prevent code-reuse attacks (including COOP attacks) [6]. In the worst case, this very limited set of methods might contain security-critical functionality: For example, assume classes `AdminUser` and `RegularUser` which both inherit from `User`. For a regular user object typed `User*` in source code, an attacker could overwrite the stored ID with the ID of `AdminUser`. This attack gives access to methods overridden by `AdminUser` that were unreachable before, but does not give access to any additional methods added by `AdminUser`, nor can the attacker trigger different methods with potentially incompatible signature or methods unrelated to `User`, as he could with regular vtables. The impact is quite program-specific, as an attack requires specific call patterns in code. This restriction applies for all solutions that restrict virtual dispatch based on a class hierarchy analysis, including [7]–[11], [17], [25]. Our level of protection is equal or better than these solutions.

In order to assess our intuition that NOVT prevents COOP-style attacks, we have to inspect how many valid targets each dispatcher allows to use. Successful COOP attacks [6] require a large set of "vfgadgets"—virtual functions that can be called by manipulating the object. Schuster et al. identified ten different types of vfgadgets that must be available for turing completeness (e.g., virtual functions reading memory, performing arithmetic operations, etc.), some of them more than once (e.g., reading memory to different registers). In unprotected programs, any virtual function (referenced in a vtable) is a possible target of any virtual dispatch in a COOP attack. With NOVT applied, all vfgadgets must be different implementations of the same method inherited from a single base class, with the exception of the "main loop" vfgadget, which in turn must dispatch that method on that base class. From Figure 5 we see that in a protected program, most virtual dispatchs have less than 10 possible targets, in particular for SPEC programs. The majority of these target functions likely does not qualify as vfgadgets because of their size and complexity. Even if, they are likely vfgadgets of the same type, because all target functions share the (high-level) semantics of the parent method.

Chromium represents a good worst-case example for vtable protection: its large codebase contain 39,100 virtual class definitions, with more than 54,700 vtables containing over 210,000 distinct virtual functions. NoVT restricts virtual dispatches to 2.9 possible targets on average. Excluding devirtualized calls, 5.7 possible targets per virtual dispatcher remain. This low number of valid targets does not allow to spawn successful COOP attack, i.e., the vast majority of virtual dispatches is COOP-safe. When looking at the maximum number of targets, we see that a small fraction of virtual dispatches have a higher number of targets (up to 2600). These dispatchers are required by some very generic interfaces in Chromium's source code that are implemented by many classes: `mojo::MessageReceiver`, `blink::ScriptWrappable`, `blink::GarbageCollectedMixin` and `google::protobuf::MessageLite`. However, we believe that these generic interfaces do not pose an additional risk for COOP-style attacks, because they all share similar function types and purposes. It is unlikely that all required types of vfgadgets are present in a set of almost-identical functions. For example, for any protobuf-inheriting object, the function `Clear` will always write memory, but never be usable as a memory-reading vfgadget. Restricting these dispatches further is not possible without breaking the legitimate use of these interfaces. This assessment is further supported by the COOP authors who state that a C++ class-hierarchy-aware restriction of virtual call targets reliably prevent COOP attacks even for large C++ target applications ("COOP's control flow can be reliably prevented when precise C++ semantics are considered from source code" [6], Section VII B).

NoVT applies the same level of protection to virtual offsets. Whenever a virtual offset is retrieved from an object, we restrict the possible results to the minimal set possible in the type system. Without this protection, an attacker could hijack vtables to arbitrarily change the *this* pointer of some inherited methods. Attackers can leverage the *this* pointer to get arbitrary memory read/write primitives from a single corrupted vtable pointer. Only few related work protects virtual offsets [8], [11], and none of them restricts virtual offset to the minimal possible set.

NoVT actually found an invalid virtual dispatch in one of the SPEC CPU benchmarks (*xalancbmk*, in `SchemaValidator::preContentValidation`). This benchmark does an invalid sidecast of an object instance, then calls a virtual method on it and finally checks if the cast was valid at all. With traditional vtables, this error would remain undetected, because both intended and actual vtables have the same layout, but NoVT detects that this dispatch violates the type system and halts the program. We are not the first to report this issue [7], [17], and provide a patch file in our source release.

The protection strength could be improved further by randomization: In contrast to vtables, type IDs are not addresses and could be randomized independently of ASLR. A simple randomization could use a constant, random offset added to all type IDs, preserving the structure of the dispatchers including their speed. To break this randomization, an attacker has to leak one type ID from memory. A stronger randomization could assign all type IDs completely random. This approach will be less efficient, because dispatchers with jumptables are not

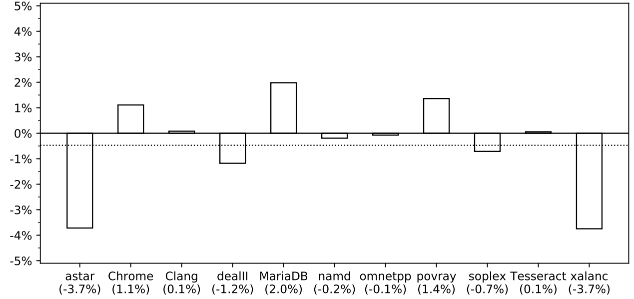

Figure 6. Performance overhead of NoVT on all programs

possible anymore, but an attacker would have to leak each type ID from an existing class instance before overwriting any stored ID.

5.2. Runtime Evaluation

The performance overhead of any protection may hinder its real-world adoption. In this section, we therefore provide thorough experiments that confirm that NoVT has a negligible (actually, *negative*) run time slowdown.

5.2.1. Benchmark Selection. We evaluate the performance overhead of NoVT using the SPEC CPU 2006 benchmark suite, parts of Chromium's benchmark suite, six well-known browser benchmarks and standard benchmarks of MariaDB, Clang and Tesseract OCR.

For SPEC, we run the full benchmark suite, excluding programs without C++. SPEC runs every program three times and takes the median runtime as result. We run each experiment 20x to get a meaningful result without outliers.

For Chromium, we use all standardized benchmarks contained in Chromium's benchmark set [26], which are *octane 2* [27], *kraken 1.1* [28], *jetstream2* [29], *Dromaeo DOM* [30] (not to be confused with the Dromaeo Javascript benchmark) and *speedometer2* [31], which is a real-world benchmark comparing the performance of the most popular web frameworks. We add *sunspider 1.0.2* [32] because it has been used to evaluate many related work [7], [8], [10]–[13], [15], [33]. Other relevant literature used additional benchmarks to assess performance impact on HTML/rendering performance, but these historical samples are not included anymore in modern Chromium (which switched from Webkit to blink). Instead, we benchmark the blink HTML engine's performance with the quite extensive blink benchmark set (326 samples in total). To avoid any bias, we selected all benchmark sets that i) contained more than just a few samples, ii) worked flawlessly on an unmodified reference Chromium and iii) whose results had a reasonably low standard deviation (below 1%). These were the blink benchmark suites *css*, *display_locking*, *dom*, *events*, *layout*, *paint* and *parser*. We take the geometric mean of all these results as Chromium's performance overhead. Again we repeated all experiments 20 times. For MariaDB, we use the DBT3 benchmark provided by the developers with the InnoDB database engine. For Clang, we measure the time it takes to compile and optimize SQLite. For Tesseract OCR, we use the benchmark from the Phoronix test suite [34]. We do not conduct runtime analysis on

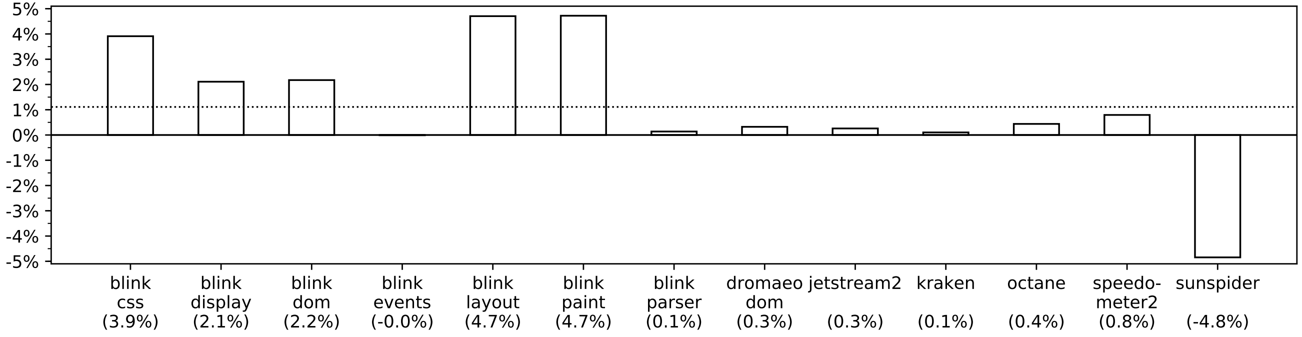

Figure 7. Performance overhead of NOVT on Chromium

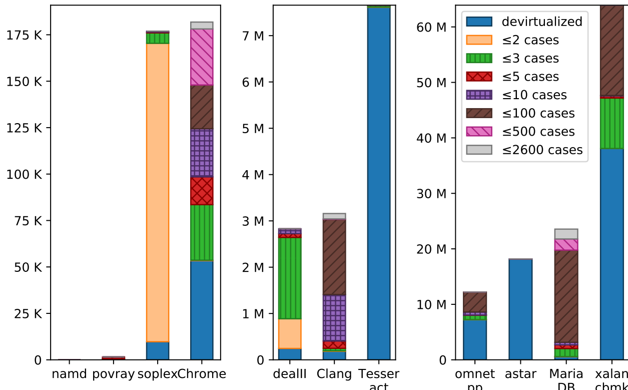

Figure 8. Virtual actions (calls, vbase offset, etc) per second, broken down by the number of `switch` cases in the NOVT dispatcher functions

CMake, because its runtime is mainly determined by the speed of the programs it invokes (e.g. compilers), and there exists no benchmark targeting CMake.

5.2.2. Benchmark Environment. We ran all SPEC benchmarks on an Intel Core i5-4690 CPU (4×3.5 GHz, no hyperthreading) with 32 GB of RAM. Operating system was Debian 10 "Buster" with kernel 4.19. We used the "performance" CPU governor, disabled CPU boost and applied `cpuset` to minimize the impact of environment and operating system on the measurements. The standard deviation of all benchmarks was at most 0.7%, and 0.32% on average. We ran all Chromium benchmarks on an Intel Core i7-6700k CPU (4×4.0 GHz with hyperthreading) with 16 GB of RAM and an AMD Radeon RX480 graphics card. Operating system was Ubuntu 18.04 with kernel 5.3. The standard deviation of all Chromium benchmarks was always below 0.92%, and 0.37% on average.

5.2.3. Performance Overhead. Figure 6 and Figure 7 show the performance overhead on the different programs and benchmarks. We can see that many programs actually get faster after protection (astar, deadII, soplex, xalancbmk), while few get slightly slower (chromium, povray, MariaDB). The average overhead is -0.5% and thus negative. The average overhead on the set of programs commonly used in related literature (SPEC CPU and Chromium) is -0.9%. That is, our benchmarks get faster on average, with a worst-case overhead of 1.98% on MariaDB. The best result is *astar*. This program has been completely devirtualized by NOVT and improves its performance by 3.7%. The highest overhead in a SPEC benchmark occurs on *povray*. Manual investigation shows that povray has only 28 classes, excluding standard library, and 1500 virtual calls per second—xalancbmk has 63 million virtual calls per second. Disabling the protection on parts of the class hierarchy reveals that its overhead does only loosely correlate to the number and structure of the generated dispatch functions. We believe that this overhead instead comes from subtle changes in the program's code layout. Evaluating hardware performance counters on synthetic microbenchmarks did unfortunately not give conclusive insights: NOVT's overhead does not correlate with cache miss rate. However, protected programs seem to have a lower branch misprediction rate.

We summarize the size and call frequency of the generated functions in Figure 8 and the number of virtual calls in Appendix B (Table 4). We can see that NOVT can handle billions of dispatches while still having a negative overhead in some cases. The same holds for virtual offsets. NOVT does not necessarily impose an overhead when virtual inheritance is used extensively. However, the actual performance overhead is not only subject to the class structure. Again, we rather speculate that performance is due to the code layout, which is out of our control.

For Chromium, we can easily see that the real-world benchmarks tend to have a much lower performance overhead than the synthetic HTML engine benchmarks. The worst case overhead is 4.7%, while the mean overhead is only 1.1%. We expect the real-world overhead to be close to the latter—in particular *speedometer2* (+0.8%) is a good candidate to measure this because it tests the performance of widely used web frameworks. *sunspider* (-4.8%) shows that even some parts of Chromium actually got faster. While this result looked suspicious at the first glance, we repeated this experiment twice to exclude any error on our side, but we can reproduce this behavior with a reasonably low standard deviation (0.5% max). Further investigation revealed that sunspider is not the only case where the protected Chromium is faster than the reference (like the "blink image_decoder" benchmark, roughly -2%). However, all other cases did not comply to our rules as described in Section 5.2.1, and we hence did not include them because they were not sufficiently representative.

Chromium seems to use less virtual dispatch than some of the SPEC benchmarks. We observed the most intensive use in the benchmark "blink display_locking"

with 510,000 virtual calls per second. On the other hand, Dromaeo DOM used only 20,000 virtual calls per second. The size of the generated functions was roughly proportional in all benchmarks, it is summarized in Figure 8. A direct connection between size and number of the called virtual functions could not be observed.

Summarizing, we can say that NoVT protects most programs without any performance penalties, and only some programs experience a slight yet negligible slowdown from the protection. Even complex programs like Chromium with large generated functions do not necessarily suffer from performance drain. Finally, with a focus on Chromium, we can say that the performance bias we introduced with our protection was more than compensated by the performance gain of enabling full link time optimization.

5.2.4. Compilation Time. The compilation time of our solution itself is unobtrusive. Our passes take a few milliseconds up to a few seconds to run on a SPEC program, and less than 40 seconds on Chromium. This compilation time is negligible in a build process of multiple minutes (SPEC) or multiple hours (Chromium), so we did not optimize the compile time in our prototype. Admittedly, building with Clang's link time optimization is slower than a regular build.

5.3. Generated Code Evaluation

NoVT replaces vtables with switch-case constructs in dispatcher functions, which the compiler then further optimizes during assembly generation. Table 1 shows that dispatchers generally follow one of four low-level code structures. Most dispatchers consist of a linear *chain* of `cmp` or `bt` checks on the type ID, followed by a conditional jump to the target function. These dispatchers usually check for only a small number (at most 4) of type IDs. They are considered fastest, because they do not access any memory. Dispatchers handling a large number of type IDs often utilize a *jumptable*, guarded by an initial range check. Jumptables require a memory read access, but scale well in the presence of many type IDs. If a jumptable is unsuitable, LLVM generates a *tree*-like comparison structure, that includes `cmp`-based range checks and bitset tests. Trees are in particular used if the number of type IDs is low, if the type IDs are sparsely distributed, or if many type IDs default to the same inherited method. While trees are slower than short compare chains, they also operate without memory access and are better suited if the type IDs are not distributed dense enough for a jumptable. The maximal tree depth we saw was five. Finally, multiple jumptables can be combined by a tree (*partial jumptable*). Our distribution algorithm tries to distribute type IDs as dense as possible, to avoid these nested structures.

From a security perspective, all generated code structures are equal—memory is only read, and all memory indices are bounds-checked. The type ID is read only once and all intermediate computation happens in registers, no TOCTOU attack (time-of-check-time-of-use) is possible.

5.4. Binary Size and Memory Overhead

We evaluate the size impact of NoVT by compiling each program with and without protection, strip the resulting binaries and compare their size. Both protected and unprotected version include a statically linked C++ standard library, and both versions run link-time optimizations over that standard library. Table 2 shows the size of the resulting binaries. It turns out that binaries usually get smaller after protection, with the exception of MariaDB. This might be counter-intuitive because each generated function should be larger than the vtable entries they replace, and many vtable entries are covered by more than one generated function. We identified two reasons for this observation: First, the virtual dispatch itself is smaller: in the protected version, a single `call` instruction to a generated function suffices to do a virtual dispatch, in vtable-based programs we would need a memory load and an offset calculation first. If the same dispatcher function is called from many locations we can save some bytes there. Second, our structure allows for more efficient dead code elimination: Our approach already identifies dead classes in its internal optimization steps and can remove them. From the final structure a simple dead global elimination pass can identify virtual methods that are never called and rtti entries that will never be accessed. In particular, the C++ standard library contains a lot of code that is not used in every program.

The memory overhead of our solution is negligible, memory usage reduced slightly for all tested programs. As we only change code and do not alter the memory layout of objects, the used heap memory (including data segment) does not change. Given that LLVM compiles all our calls inside generated functions as tail calls, our solution does not use any additional memory on the stack. So the only difference in memory consumption comes from the different size of the binary (which is usually smaller).

6. Compatibility and Limitations

NoVT is compatible to both, small and extremely large (e.g., Chromium) programs. We tested the correctness of all benchmarked programs with different inputs (usually from their benchmark suites). To this end, we compiled LLVM 9 and Clang 9 with NoVT and ran their quite extensive unit test suite (around 3800 tests) to confirm that no error was introduced by NoVT—excluding a single test that required dynamic linking, which NoVT does not support. CMake and MariaDB also have extensive test suites, we confirmed that NoVT does not alter the outcome of any test.

While all benchmarks were evaluated on the 64-bit x86 architecture, NoVT works for any architecture supported by LLVM. NoVT can thus, for example, also protect mobile or embedded applications. To demonstrate this, we compiled a set of 40 small test programs for different architectures, including 32-bit x86, ARM, 64-bit ARM, MIPS, 64-bit MIPS and 64-bit PowerPC. We verified that these protected programs work as expected by executing and using them in a QEMU emulator.

Our current prototype assumes the program to be compiled with Clang. We chose Clang due to its wide popularity and acceptance. Clang can build even the Linux kernel [35] and has interfaces compatible to GCC or Microsoft's C++ compiler. Having said this, porting our approach to other compilers is just a matter of engineering.

TABLE 1. SUMMARY OF ASSEMBLY CONSTRUCTS USED TO BUILD VIRTUAL FUNCTION DISPATCHERS.

Binary	Compare chains	Trees	Jumptable	Partial Jumptable	Other/ukn	Total
Chromium	39077 (82.71%)	3289 (6.96%)	4331 (9.17%)	114 (0.24%)	435 (0.92%)	47246
Clang	2307 (49.63%)	1094 (23.54%)	489 (10.52%)	528 (11.36%)	230 (4.95%)	4648
MariaDB	1107 (45.86%)	537 (22.25%)	382 (15.82%)	281 (11.64%)	107 (4.43%)	2414
CMake	259 (53.96%)	99 (20.62%)	55 (11.46%)	39 (8.12%)	28 (5.83%)	480
Tesseract	215 (78.75%)	10 (3.66%)	14 (5.13%)	1 (0.37%)	33 (12.09%)	273
astar	1 (100.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	1
dealII	72 (79.12%)	2 (2.20%)	11 (12.09%)	0 (0.00%)	6 (6.59%)	91
omnetpp	95 (59.38%)	12 (7.50%)	43 (26.88%)	0 (0.00%)	10 (6.25%)	160
povray	33 (100.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	33
soplex	98 (73.68%)	0 (0.00%)	33 (24.81%)	0 (0.00%)	2 (1.50%)	133
xalancbmk	1320 (81.89%)	136 (8.44%)	103 (6.39%)	37 (2.30%)	16 (0.99%)	1612

TABLE 2. SIZE OF BINARIES BEFORE AND AFTER PROTECTION

	Unprotected binary size	Protected binary size	Binary size overhead
Chromium	211,025 KB	197,959 KB	−6.19%
Clang	125,914 KB	125,779 KB	−0.11%
CMake	11,149 KB	10,629 KB	−4.66%
MariaDB	21,643 KB	24,779 KB	14.49%
Tesseract	3,647 KB	3,162 KB	−13.30%
astar	149 KB	143 KB	−3.96%
dealII	1,322 KB	817 KB	−38.13%
namd	379 KB	371 KB	−2.08%
omnetpp	1,447 KB	847 KB	−41.45%
povray	1,532 KB	1,520 KB	−0.79%
soplex	1,034 KB	551 KB	−46.69%
xalan	4,624 KB	3,983 KB	−13.87%

Finally, we do not lose compatibility by using full LTO. While it increases compilation time, using full LTO improves runtime performance. Alternatively, NOVT could be implemented based on the faster "thin LTO".

Our approach has one main limitation: It is not compatible to dynamic linking or dynamic loading of C++-based libraries. This is an inherent drawback of all protection schemes that require full knowledge of the class hierarchy during build time. After the program is compiled, it is not *per se* possible to add more allowed classes to a specific virtual dispatch function. If a class from a runtime-loaded C++ library is now inheriting from a class already known, the new loaded class can not be respected in the protection, calls to this class are not permitted. To the best of our knowledge, most vtable protections have this disadvantage [7], [8], [12], [17], [25], and those that support dynamic linking face either performance overheads [10] or weaker security guaranties [9], [11].

In fact, the lack of support for dynamic linking is not as crucial as it may look at first sight. First, programs compiled with NOVT can still dynamically link and load classical C libraries, or C++ libraries that expose a C-style interface. Other C++ libraries can be compiled with NOVT and then be linked statically into the final binary—like we did for `libstdc++` in our experiments. Second, modern application deployment systems like Flatpak, Docker or Snap already bundle and ship an application together with all its dependencies. Applications packaged by such a system do not have any advantage of dynamic linking a library. Even without such a deployment system, some applications statically link the majority of their dependencies. For example, release builds of Google Chrome contain statically linked libraries, including the C++ standard library. Third, many programming languages apart from C/C++ already use static linking as their default (or only) way of linking, including Go, Rust, Haskell and OCaml.

As a major endeavor, in principle one could extend NOVT to support dynamic libraries and dynamic loading of C++ code with an arbitrary interface. While this significant improvement would increase the compatibility of our protection, we expect a negative impact on performance.

7. Related Work

7.1. Attacks on Vtables

Code-reuse attacks are still the most prevalent attack method on C(++) programs nowadays. C++ programs are particularly prone to call-oriented programming (COP) [36], as they usually contain many indirect calls (necessary for virtual dispatch). Schuster et al. [6] present *Counterfeit Object-oriented Programming (COOP)*—a new attack targeting C++ programs by using only valid vtables. COOP attacks chain virtual functions together in a way that resembles the original calling structure of a C++ program, breaking most vtable protection schemes available at that time. Haller et al. [18] revealed common errors in vtable protections and improve upon the precision of GCC's *VTV* protection scheme. They prove that their correction to VTV is optimal given context insensitivity.

7.2. Vtable Protections

Seeing the popularity of vtable hijacking attacks, it is not surprising to see the wealth of literature on C++ vtable protection schemes. (Table 3) summarizes related works, all of which enforce a C++-specific CFI policy. We group these into approaches that rely on compilers or binary rewriting.

Protecting binaries is naturally harder as vtables and virtual calls must be extracted from a (possibly stripped) binary. A first such approach was *T-VIP* [13], which enforced that vtable pointers point to read-only memory, preventing arbitrary control over the instruction pointer. Later *vfGuard* [14] and *VTint* [15] enforced a stronger policy, only detected vtables are allowed as targets of vtable pointers. The recent *VCI* [16] scheme can extract a class hierarchy from binaries and restrict the set of possible vtables further, while still not achieving the precision of a compiler-based approach.

TABLE 3. RELATED WORK IN COMPARISON TO NOVT, GROUPED BY BINARY-BASED AND COMPILER-ASSISTED SOLUTIONS. WHILE STATE-OF-THE-ART BINARY DEFENSES ARE STILL STRUGGLING WITH COOP ATTACKS, SOURCE-BASED DEFENSES HAVE SOLVED THIS ISSUE FOR SOME TIME. HOWEVER, SURPRISINGLY MANY SOLUTIONS ARE NOT OPTIMAL AS OUTLINED BY SHRINKWRAP. SIMILARLY, ONLY FEW SOLUTIONS PROTECT OTHER VTABLE VTABLE USAGES THAN DYNAMIC CALLS.

Type	Related Work	Protection: calls / offsets / rtti & casts	Handles method pointers	Defeats COOP	Optimal (1)	Runtime Overhead	Remarks
Binary	T-VIP [13]	✓××	×	×	×	~25% (SPEC)	requires profiling
	vfGuard [14]	✓××	×	×	×	18.3% (Internet Explorer)	
	VTint [15]	✓××	×	×	×	0.4% (SPEC), 1.4% (Chromium*)	instruments less calls than other solutions
	VCI [16]	✓××	×	partial	×	7.79% (SPEC + Chromium*)	
Source-code based	VT-Guard [9]	✓××	✓	×	×	unclear	patent does not detail its performance penalties
	SafeDispatch [8]	✓✓×(2)	✓	✓	✓	2.1% (Chromium)	requires profiling for performance
	VTV [10]	✓××	✓	✓	×(3)	1% - 8.7% (SPEC)	requires profiling for performance
	Redactor++ [25]	✓××	✓	✓	✓	8.4% (SPEC), 7.9% (Chromium)	probabilistic defense, requires execute-only memory
	LLVM-VCFI [17]	✓××	×	✓	×	1.97% (SPEC), 2.9% (Chromium) [7]	
	VTrust [11]	✓✓✓(4)	×	✓	×	2.2% (partial SPEC)	
	OVT/IVT [7]	✓××	×	✓	✓(5)	1.17% (SPEC), 1.7% (Chromium)	ShrinkWrap-safe configuration doubles overhead on SPEC
	VIP [12]	✓××	×	✓	✓	0.7% (SPEC)	compilation can take hours
	CFIXX [37]	✓××	✓	✓	n.a.	4.96% (SPEC)	Object Type Integrity, not CFI. Requires MPX CPU instructions
	NoVT	✓✓✓	✓	✓	✓	−0.9% (SPEC), 1.1% (Chromium)	

(1) The solution restricts possible calls to the minimal possible set, as shown in ShrinkWrap [18]
(2) Virtual offset checks are not optimal and have additional runtime overhead
(3) ShrinkWrap [18] proposed a fix (without additional runtime overhead)
(4) For virtual offsets, RTTI access and dynamic casts, their check is weaker (no check if the valid vtable matches the static type)
(5) optional, with higher runtime overhead
(*) Only a few benchmarks have been used (*octane* etc.), but no HTML or rendering workloads (which typically have more overhead)

In contrast, compiler-based protections can get all necessary information from source code and hence have higher precision. *SafeDispatch* [8] was one of the first vtable-specific approach that protects virtual calls with little overhead. However, this little overhead could only be achieved by dynamic profiling of the application. SafeDispatch did not only protect virtual calls, but also included virtual offsets stored in vtables (with a weaker protection level, as outlined by ShrinkWrap [18]). It uses a class hierarchy analysis [20] to infer valid call targets. *Redactor++* [25] is another solution based on randomization and information hiding. While not enforcing CFI, it hides necessary vtable information such that COOP attacks require hardly feasible guesswork. Its defense is probabilistic and could be circumvented in some settings [38], and it requires a system offering execute-only memory. *VTrust* [11] improves over these solutions in terms of compatibility, and protects vtables without knowledge of a full class hierarchy. VTrust protects not only virtual dispatch but also virtual offsets and type information, but at a much weaker level: when resolving virtual offsets or loading rtti, any vtable can be used, the check is type-agnostic.

Despite these academic progresses, major C++ compilers use their own vtable protections: Microsoft included a canary-based solution named *VT-Guard* [9] in their Visual C++ compiler. This solution is not strong enough to prevent COOP attacks. GCC included a method called *VTV* [10] which has been improved by ShrinkWrap [18]. VTV focuses on compatibility, but has a non-negligible runtime overhead (and again requires dynamic profiling to achieve its performance). LLVM has its own forward-CFI approach [17] that includes a vtable-specific protection. While this solution is more performant than VTV, it is neither optimal nor complete. Again, all of these solutions focus on virtual dispatch exclusively.

Recent work has brought different improvements. Bounov et al. introduced *OVT* and *IVT* [7], two protections striving to improve performance by ordering and interleaving vtables. They achieve an overhead as low as 1.17% with a non-optimal protection (and the possibility to turn it into an optimal one). VIP [12] improves the security guarantees of vtable protections, introducing a pointer analysis technique that is used to reduce the set of possible vtables in a way that is not possible without context. However, their analysis takes up to an hour on SPEC, and 6 hours on Chromium. As an alternative to vtable-based CFI schemes, *Object Type Integrity* protects vtables pointers (instead of protecting virtual calls) [37]. Their prototype *CFIXX* has a reasonable overhead, but requires Intel's deprecated MPX CPU extensions.

NoVT improved over all previous solutions in terms of performance and protection. We protect all usages of a vtable, including virtual offsets, type information and

dynamic casts, while most previous implementations only protected virtual calls. Our protection is optimal in a context-free setting (as shown in [18]), for all protected usages. To the best of our knowledge, we are the first vtable protection that actually speeds up most of the programs it is applied to. At the same time, our solution lives with the same limitation (no dynamic linking) than most previous source-based solutions [7], [8], [12], [17], [25]. Solutions that (partially) support dynamic linking have a higher overhead [10], [11], [37] or lower security level [9], [11].

7.3. Control Flow Integrity

CFI [39] schemes aim to protect all control transfers, including function pointers and return addresses, whereas our protection specifically targets C++ vtables with much higher precision. Different CFI schemes with varying protection and methods have been proposed over the years [40]. MCFI [41] adds modularity support to classical CFI. CCFIR [33] uses a dedicated “springboard section” and randomization to improve on the performance of CFI. WIT [42] protects not only indirect control flows, but also aims at memory writes. MoCFI [43] adapts CFI to the requirements of mobile devices. While classical CFI usually works on source code, binCFI [44] is able to protect control flow on closed-source binaries. Opaque CFI [45] protects binaries even if the adversary has full knowledge about the code and memory layout.

HyperSafe [46] shares some design ideas with NoVT: To protect C-style indirect function calls and returns in hypervisors, HyperSafe replaces function pointers and return addresses with function IDs. At each indirect control transfer, the ID is checked and resolved through a per-callside lookup table. In contrast, NoVT targets C++, is usable in general and uses flexible switch instructions instead of lookup tables. µRAI [47] protects return targets on microcontrollers. Partial paths in the control-flow graph get a “function ID” assigned, one register is reserved to maintain the current ID at runtime. Instead of common return instructions and return addresses stored on the stack, µRAI can determine the unique correct return address from the current ID. Returns are implemented using a jumptable over all possible IDs, removing return addresses on the stack altogether.

As a different approach, code-pointer integrity and code-pointer separation has been introduced [48]. CPI intercepts attacks a step earlier by preventing modifications of code pointers. These methods are orthogonal to ours and can be used in combination. Other approaches try to stop attacks even before a code pointer can be corrupted, trying to enforce memory safety [49]–[51]. Advantages and disadvantages of these and further CFI protections are analyzed and summarized in [40] and [52].

7.4. Alternatives to Vtables

Decades ago, similar to our general idea, the compiler community has explored alternatives to vtables. Although these approaches did not have a security focus, and hence also did not discuss security-critical considerations in this respect, we will briefly describe them in the following.

Porat et al. [21] used static type checks and direct calls to speed up hot paths in virtual dispatch—virtual method implementations for frequently used classes are called directly, while less frequent classes fall back to classical vtables. Vtables are not changed, and no type IDs are introduced. However, this approach only offers small speed improvements, but no security gain.

SmallEiffel [53] was an experimental compiler for the Eiffel programming language that does not use vtables. Instead, it uses type IDs for classes and a binary search tree for virtual dispatch. While the basic idea of type IDs is similar to NoVT, SmallEiffel’s design can’t be easily ported to C++, because Eiffel lacks many features like mulitple or virtual inheritance that C++ has. Furthermore, SmallEiffel has an unclear security contribution, because unexpected type IDs trigger undefined behavior. In contrast, NoVT’s type identification system is more advanced, can deal with all object-oriented features of a modern language like C++, has a very strong focus on security and the generated dispatchers can deliver better performance than binary search trees.

8. Conclusion

NoVT uses a modified compiler to protect programs given complete source code (including libraries). We thereby radically change the way of *protecting* vtables and, instead, *eliminate* them. NoVT replaces traditional, vtable-based virtual dispatch with direct calls—based on a class identifier in each C++ class instance it calls an object’s method non-virtual. Using a class hierarchy analysis at link-time, NoVT determines which method implementations are possible according for each virtual call, at runtime only these methods are callable. After a protection with NoVT, no traditional vtables or vtable pointers remain in the program.

NoVT is compatible with all C++ programs, with the exception of dynamic linking and loading—as most previous solutions, NoVT relies on knowledge of the complete class hierarchy. Legacy software can easily be protected without any source code modifications or additional dependencies. According to [18], NoVT’s offered protection level is optimal for a type-based solution. NoVT is able to defend against strong vtable-based attacks like COOP, even for large programs. NoVT has been evaluated on SPEC CPU 2006, Chromium, MariaDB, Clang, CMake and Tesseract OCR. The introduced performance overhead is often negative, −0.5% on average and 2% in the worst case. The generated binaries are usually smaller after protection, no memory overhead is introduced, the impact on compilation time is minimal.

Acknowledgment

We thank our anonymous reviewers and our shepherd for their valuable feedback. Also we thank Jonas Bushart for his paper draft review.

Availability

Our prototype has been released as Open-Source Software, it is avaliable on Github:
https://github.com/novt-vtable-less-compiler/novt-llvm

References

[1] V. Lextrait, "The Programming Languages Beacon." [Online]. Available: https://www.mentofacturing.com/Vincent/implementations.html

[2] PaX, "Address Space Layout Randomization." [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[3] H. Etoh and K. Yoda, "Protecting from stack smashing attacks," 01 2000.

[4] The Chromium Projects, "Memory safety." [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety

[5] M. Corporation, "Microsoft Security Intelligence Report vol.16." [Online]. Available: http://download.microsoft.com/download/7/2/b/72b5de91-04f4-42f4-a587-9d08c55e0734/microsoft_security_intelligence_report_volume_16_english.pdf

[6] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. USA: IEEE Computer Society, 2015, p. 745–762. [Online]. Available: https://doi.org/10.1109/SP.2015.51

[7] D. Bounov, R. G. Kici, and S. Lerner, "Protecting C++ Dynamic Dispatch Through VTable Interleaving," in *23rd Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/protecting-cpp-dynamic-dispatch-through-vtable-interleaving.pdf

[8] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/safedispatch-securing-c-virtual-calls-memory-corruption-attacks

[9] M. R. Miller and K. D. Johnson, "Using virtual table protections to prevent the exploitation of object corruption vulnerabilities," Patent US 2012/0 144 480 A1, patent number US 2012/0144480 A1.

[10] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 941–955.

[11] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "VTrust: Regaining Trust on Virtual Calls," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/vtrust-regaining-trust-virtual-calls.pdf

[12] X. Fan, Y. Sui, X. Liao, and J. Xue, "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*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 329–340. [Online]. Available: https://doi.org/10.1145/3092703.3092729

[13] R. Gawlik and T. Holz, "Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 396–405. [Online]. Available: https://doi.org/10.1145/2664243.2664249

[14] X. Hu and H. Yin, "vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 01 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/vfguard-strict-protection-virtual-function-calls-cots-c-binaries

[15] C. Zhang, C. Song, K. Chen, Z. Chen, and D. Song, "VTint: Protecting Virtual Function Tables' Integrity," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 02 2015.

[16] M. Elsabagh, D. Fleck, and A. Stavrou, "Strict Virtual Call Integrity Checking for C++ Binaries," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 140–154. [Online]. Available: https://doi.org/10.1145/3052973.3052976

[17] T. C. Team, "Control Flow Integrity Design Documentation." [Online]. Available: https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html

[18] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos, "ShrinkWrap: VTable Protection without Loose Ends," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 341–350. [Online]. Available: https://doi.org/10.1145/2818000.2818025

[19] The Clang Team, "SafeStack." [Online]. Available: https://clang.llvm.org/docs/SafeStack.html

[20] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis." Springer-Verlag, 1995, pp. 77–101.

[21] S. Porat, D. Bernstein, Y. Fedorov, J. Rodrigue, and E. Yahav, "Compiler Optimization of C++ Virtual Function Calls," in *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies*. Toronto, Ontario, Canada: USENIX Association, Jun. 1996.

[22] CodeSourcery, Compaq, EDG, HP, IBM, Intel, R. Hat, and SGI, "Itanium C++ ABI." [Online]. Available: https://itanium-cxx-abi.github.io/cxx-abi/abi.html

[23] Standard Performance Evaluation Corporation, "SPEC CPU® 2006." [Online]. Available: https://www.spec.org/cpu2006/

[24] The Chromium Projects, "Chromium." [Online]. Available: https://www.chromium.org/Home

[25] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 243–255.

[26] The Chromium Projects, "Telemetry: Run Benchmarks Locally." [Online]. Available: https://chromium.googlesource.com/catapult/+/HEAD/telemetry/docs/run_benchmarks_locally.md

[27] Octane Team Google, "Octane 2.0." [Online]. Available: https://chromium.github.io/octane/

[28] Mozilla, "Kraken JavaScript Benchmark (version 1.1)." [Online]. Available: https://krakenbenchmark.mozilla.org/

[29] Webkit, "JetStream2." [Online]. Available: https://browserbench.org/JetStream/

[30] Mozilla, "Dromaeo: Javascript Performance Testing." [Online]. Available: http://dromaeo.com/?dom

[31] Webkit, "Speedometer 2.0." [Online]. Available: https://browserbench.org/Speedometer2.0/

[32] ——, "SunSpider 1.0.2 JavaScript Benchmark." [Online]. Available: https://webkit.org/perf/sunspider/sunspider.html

[33] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. USA: IEEE Computer Society, 2013, p. 559–573. [Online]. Available: https://doi.org/10.1109/SP.2013.44

[34] M. Larabel, "Tesseract OCR Benchmark - OpenBenchmarking.org." [Online]. Available: https://openbenchmarking.org/test/system/tesseract-ocr

[35] The kernel development community, “Building Linux with Clang/LLVM.” [Online]. Available: https://www.kernel.org/doc/html/latest/kbuild/llvm.html

[36] N. Carlini and D. Wagner, “ROP is Still Dangerous: Breaking Modern Defenses,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC’14. USA: USENIX Association, 2014, p. 385–399.

[37] N. Burow, D. McKee, S. A. Carr, and M. Payer, “CFIXX: Object Type Integrity for C++,” in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_05A-2_Burow_paper.pdf

[38] G. Maisuradze, M. Backes, and C. Rossow, “What Cannot be Read, Cannot be Leveraged? Revisiting Assumptions of JIT-ROP Defenses,” in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/maisuradze

[39] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-Flow Integrity,” in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS ’05. New York, NY, USA: Association for Computing Machinery, 2005, p. 340–353. [Online]. Available: https://doi.org/10.1145/1102120.1102165

[40] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal War in Memory,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.13

[41] B. Niu and G. Tan, “Modular Control-Flow Integrity,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 577–587. [Online]. Available: https://doi.org/10.1145/2594291.2594295

[42] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, “Preventing Memory Error Exploits with WIT,” in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP ’08. USA: IEEE Computer Society, 2008, p. 263–277. [Online]. Available: https://doi.org/10.1109/SP.2008.30

[43] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, “MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones,” in *Proc. of 19th Annual Network & Distributed System Security Symposium (NDSS)*, feb 2012. [Online]. Available: http://www.internetsociety.org/sites/default/files/07_2.pdf

[44] M. Zhang and R. Sekar, “Control Flow Integrity for COTS Binaries,” in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC’13. USA: USENIX Association, 2013, p. 337–352.

[48] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-Pointer Integrity,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’14. USA: USENIX Association, 2014, p. 147–163.

[45] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz, “Opaque Control-Flow Integrity,” in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 02 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/opaque-control-flow-integrity

[46] Z. Wang and X. Jiang, “HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. Berleley/Oakland, California, USA: IEEE Computer Society, May 2010, pp. 380–395.

[47] N. Almakhdhub, A. Clements, S. Bagchi, and M. Payer, “μRAI: Securing Embedded Systems with Return Address Integrity,” in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 01 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/murai-securing-embedded-systems-with-return-address-integrity/

[49] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C,” *SIGPLAN Not.*, vol. 44, no. 6, p. 245–258, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1543135.1542504

[50] ——, “CETS: Compiler Enforced Temporal Safety for C,” in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 31–40. [Online]. Available: https://doi.org/10.1145/1806651.1806657

[51] D. Dhurjati and V. Adve, “Backwards-Compatible Array Bounds Checking for C with Very Low Overhead,” 2006.

[52] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, “Control-Flow Integrity: Precision, Security, and Performance,” vol. 50, no. 1, 2017.

[53] O. Zendra, D. Colnet, and S. Collin, “Efficient Dynamic Dispatch without Virtual Function Tables: The SmallEiffel Compiler.” New York, NY, USA: Association for Computing Machinery, 1997.

Appendix A. Identifier Number Assignment

The full algorithm to assign numbers to class identifiers (as described in Section 4.5.5) is given in Algorithm 1.

Appendix B. Number of Protected Operations

We measured the number of virtual calls, virtual offset accesses and dynamic casts for all evaluated benchmarks. We report them in Table 4.

```
function CreateIdentifierNumbers:
    input  : identifiers // identifier subgraph set
    input  : next_number // initial 0
    output: next_number // next free number

    if ||identifiers|| > 5 then // small sets need no advanced ordering
        biggest_subset := ∅;
        /* find the biggest subset from all generated functions, ignoring small functions */
        foreach func in generated_functions do
            if func.used_ids ⊊ identifiers and 4 ≤ ||func.used_ids|| and ||func.used_ids|| > ||biggest_subset||
             then
                biggest_subset := func.used_ids;
            end
        end
        if biggest_subset ≠ ∅ then
            /* identifiers used together should get connected numbers */
            next_number := CreateIdentifierNumbers (biggest_subset, next_number);
            next_number := CreateIdentifierNumbers (identifiers − biggest_subset, next_number);
            return next_number
        end
    end
    /* Order small sets and sets that can not be split. The order of identifiers is close to prefix traversal of the identifier
       tree. */
    foreach id in identifiers do
        id.number := next_number++;
    end
    return next_number
end
```

Algorithm 1: The algorithm used to assign numbers to class identifiers.

TABLE 4. NUMBER OF PROTECTED OPERATIONS PER BENCHMARK

Benchmark	# virtual call	# virtual offset	# dynamic_cast	# rtti access	runtime
Chromium: blink css	1,123,380,274	364,138	0	0	251 sec
Chromium: blink display_locking	24,693,016	168,377	0	0	49 sec
Chromium: blink dom	1,542,188,247	630,578	0	0	80 sec
Chromium: blink events	8,315,599	70,093	0	0	94 sec
Chromium: blink layout	35,869,225,028	1,755,609	0	0	1150 sec
Chromium: blink paint	5,607,395,213	859,399	0	0	385 sec
Chromium: blink parser	4,331,060,409	523,289	0	0	453 sec
Chromium: dromaeo	1,947,841,146	30,748	0	1	149 sec
Chromium: jetstream2	804,807,604	316,224	2	5	180 sec
Chromium: kraken	59,387,742	59,417	0	1	34 sec
Chromium: octane	118,479,849	24,373	0	1	50 sec
Chromium: speedometer2	541,254,241	194,405	0	0	74 sec
Chromium: sunspider	5,766,321	18,361	0	1	18 sec
Clang	97,010,210	961,927	493,848	0	31 sec
MariaDB	15,547,629,306	59,940	43,740	0	263 sec
Tesseract	292,155,762	192,129	16,793	0	38 sec
astar	4,996,986,681	0	0	0	286 sec
dealII	201,873,776	98,486,795	225,926,036	0	188 sec
namd	0	0	0	0	292 sec
omnetpp	3,361,136,271	14	47,429,169	330	279 sec
povray	153,212	0	0	0	98 sec
soplex	3,259,515	32,123,619	161	0	202 sec
xalancbmk	9,867,616,106	612,327	48	0	160 sec