Link-Time Optimization of Dynamic Casts in C++ Programs

XUFAN LU and NUNO P. LOPES, INESC-ID / Instituto Superior Técnico - University of Lisbon, Portugal

A core design principle of C++ is that users should only incur costs for features they actually use, both in terms of performance and code size. A notable exception to this rule is the run-time type information (RTTI) data, used for dynamic downcasts, exceptions, and run-time type introspection.

For classes that define at least one virtual method, compilers generate RTTI data that uniquely identifies the type, including a string for the type name. In large programs with complex type inheritance hierarchies, this RTTI data can grow substantially in size. Moreover, dynamic casting algorithms are linear in the type hierarchy size, causing some programs to spend considerable time on these casts.

The common workaround is to use the `-fno-rtti` compiler flag, which disables RTTI data generation. However, this approach has significant drawbacks, such as disabling polymorphic exceptions and dynamic casts, and requiring the flag to be applied across the entire program due to ABI changes.

In this paper, we propose a new link-time optimization to mitigate both the performance and size overhead associated with dynamic casts and RTTI data. Our optimization replaces costly library calls for downcasts with short instruction sequences and eliminates unnecessary RTTI data by modifying vtables to remove RTTI slots. Our prototype, implemented in the LLVM compiler, demonstrates an average speedup of 1.4%, as well as an average binary size reduction of 1.7%.

CCS Concepts: • **Software and its engineering** → **Polymorphism**; **Compilers**.

Additional Key Words and Phrases: Link-Time Optimizations, Dynamic Casts, RTTI, C++, LLVM

1 Introduction

C++'s design is rooted in the so-called zero overhead abstraction principle [69], meaning that users only pay for the features they actually use. For example, method calls in C++ are dispatched statically by default to avoid potentially unnecessary run-time overhead. Methods need to be explicitly tagged with the `virtual` keyword to enable per-method dynamic dispatch.

A notable exception to this design principle is the run-time type information (RTTI) data, which is used by compilers and runtime libraries to implement dynamic downcasts, exceptions, and run-time type introspection (i.e., the `typeid` operator). Below is a simple program with three classes with simple straight-line type inheritance:

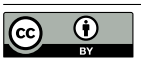
```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };

void f(A *obj) {
  if (dynamic_cast<B*>(obj))
    // obj is of type B or C or ...?
}
```

Function `f` receives a pointer to an object of type `A`. Note, however, that the object can also be of type `B` or `C`, since they both inherit directly or transitively from `A`. The dynamic cast checks if the object is of type `B` or any other class that inherits from it (in this case, `C`).

The implementation of `dynamic_cast` is not dictated by the C++ standard; compilers are free to implement it in whatever way they see fit. Compatibility between compilers on the same platform

Authors' Contact Information: Xufan Lu, luxufan@tecnico.ulisboa.pt; Nuno P. Lopes, nuno.lopes@tecnico.ulisboa.pt, INESC-ID / Instituto Superior Técnico - University of Lisbon, Portugal.



is ensured through an ABI, which specifies, among other things, the layout of objects in memory and the RTTI data the compiler must emit for the C++ runtime library's `dynamic_cast` algorithm.[1]

Most C++ projects compile their code on a per-file basis. This means that the compiler does not know anything about the rest of the code in other files and must assume the worst. One unfortunate implication is that the type hierarchy is usually open, i.e., the compiler does not know if there are other classes inheriting from the ones in the file being compiled. For example, when compiling a file with the code above, the compiler must allow other files to inherit from any of the three classes.[2] Thus, the `dynamic_cast` above may also succeed for an object of, say, type `D` that is defined in another file and that inherits from `B` or `C`.

Since C++ compilers must live with the open-world assumption for the type hierarchy, they are forced to always emit RTTI data, regardless of whether it is used by the program or not. The compiler never knows if there is a dynamic cast in one file referring a class defined in another file. The implication is that RTTI is not a zero-overhead abstraction: all programs need to pay for it even if they do not use exceptions or dynamic casts, or if they use them but only with a subset of the classes.

RTTI data can be quite large, specially for programs with deep type inheritance chains or namespaces with long names. For each class with virtual methods, an additional 32 to 40 bytes are used to store RTTI pointers. A string containing the mangled name of each class is also stored. Since these strings contain the full namespace and class names, they become very large. Below is an example of a mangled type string for a class of the Chromium web browser that occupies 375 bytes (note that Chromium has 45k classes!):

```
_ZTIZN9reporting12StorageQueue6CreateERKNS_12QueueOptionsEN4base17RepeatingCallbackIFvNS_17Uploader
Interface12UploadReasonENS4_12OnceCallbackIFvNS4_8expectedINSt4__Cr10unique_ptrIS6_NSA_14default_de
leteIS6_EEEENS_8internal11ErrorStatusEEEEEEEE13scoped_refptrINS_25EncryptionModuleInterfaceEESM_IN
S_17CompressionModuleEENS8_IFvNS9_ISM_IS0_ESG_EEEEE23StorageQueueInitContext
```

Mangled type names are long because they encode the fully qualified type name of all the parameters and return value. While some platforms compare these strings when doing dynamic casts, most do not and thus these strings are only used if the program uses the `typeid` operator. In fact, we show in Section 3.2 that most RTTI data is never accessed.

To avoid the overhead of RTTI data, large C++ programs, such as Chromium and LLVM disable the emission of RTTI data altogether using the `-fno-rtti` compiler flag. This disables all features that depend on RTTI data, including dynamic casts. LLVM, for instance, implements its own dynamic casting mechanism by hand, requiring some code and an extra field in all classes.

Given the advances in link-time optimization (LTO) in the past decade, we explore using LTO to optimize dynamic casts and remove unneeded RTTI data. We leverage LTO to obtain the type hierarchies and RTTI data that are internal to the program, i.e., classes that cannot be extended or accessed by external libraries. Our new optimization replaces calls into the C++ runtime library's implementation of `dynamic_cast` with optimized code sequences that exploit the full knowledge of the type hierarchies. Thus, we allow programs to use dynamic casts and exceptions without having to pay for RTTI data for the whole program.

The contributions of this paper are as follow:

(1) A new benchmark suite consisting of ten C++ programs that either use dynamic casts, or implement their own casting mechanism and that have been modified to use standard dynamic casts (total of 55 million lines of code).

[1] For example, the ABI used by Linux systems is specified in https://itanium-cxx-abi.github.io/cxx-abi/abi.html.

[2] C++ 2011 introduced a new `final` keyword that can be used to annotate leaf classes (no class can inherit from them) to help the compiler. Nevertheless, there is still no mechanism to annotate classes in the middle of the type hierarchy, so the problem of the type hierarchy being open persists even if developers were to annotate all leaf classes.

(2) Characterization of the benchmark suite with respect to how these C++ programs use dynamic casts in practice.
(3) A new link-time optimization algorithm that optimizes dynamic casts into short code sequences and that removes unneeded RTTI data.
(4) An implementation of the proposed algorithm in the LLVM compiler, supporting both the full LTO and ThinLTO compilation pipelines.
(5) Evaluation of the proposed algorithm on the benchmark suite, showing an average speedup of 1.4% and an average binary size reduction of 1.7%. For large applications that implement their own versions of dynamic casts, we show that using C++'s `dynamic_cast` results in up to a 4× slowdown and that our optimization can recover most of the overhead.

2 Run-Time Type Information (RTTI)

In this section, we give an overview of what RTTI data is and how it is used by C++ compilers and runtime libraries to implement dynamic casts.

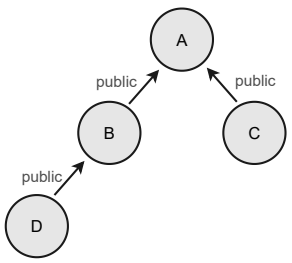
RTTI is a form of metadata that is used to represent type information that can be queried during program execution by the program itself through the `typeid` operator and by the C++ runtime library. To explain the general principles of RTTI, consider the following example C++ code on the left, and its corresponding type hierarchy graph on the right:

```
class A {
  int a;
public:
  virtual void foo();
};
class B : public A { int b; };
class C : public A { int c; };
class D : public B { int d; };

B* cast_to_B(A *a) {
  return dynamic_cast<B*>(a);
}

C* cast_to_C(A *a) {
  return dynamic_cast<C*>(a);
}

void f() {
  B *pb = new B();
  B *b = cast_to_B(pb);
  C *c = cast_to_C(pb);
}
```



In this code snippet, class `A` is polymorphic: classes `B` and `C` inherit directly from `A`. Function `f` first allocates an object of type `B` on the heap. Then it calls function `cast_to_B`, which receives an object of type `A` (or any derived type) as argument, and returns the same object if it is of type `B`. In this case it is, and hence `b` will have the same value as `pb`. Conversely, the dynamic cast in `cast_to_C` fails, and thus the function returns a null pointer.

Fig. 1 show the memory layout of an object of class `B`. The object itself occupies 16 bytes: 8 bytes for the virtual table (vtable) pointer (to implement dynamic function dispatch), and 4 bytes for each integer field. Each class (and not object) has a unique vtable, containing one pointer per virtual method in the class; in this case it has just one pointer for `foo`. Since `B` does not override the

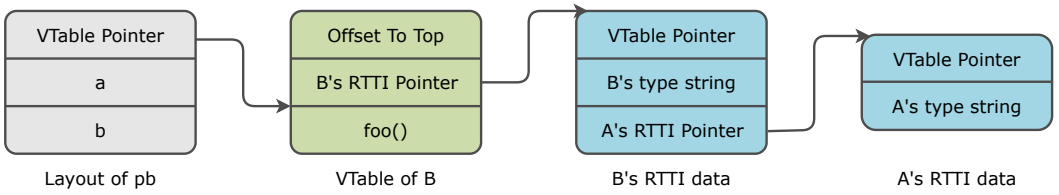


Fig. 1. Memory layout for an object pb of class B and its corresponding vtable and RTTI data (Itanium ABI).

definition of `foo`, the vtable points to `A::foo`. Each vtable also contains a pointer to the class's RTTI data, and an offset used to implement multiple inheritance, which we ignore for now.

RTTI data is itself polymorphic. B's RTTI data contains 3 pointers: (1) a pointer to the vtable of one of the three RTTI classes defined in the C++ runtime library (depending on the inheritance type), (2) a string with the full type name (including namespaces) mangled according to the ABI, and (3) a pointer to the RTTI data of class A since B inherits directly from A. The RTTI data of A only has 2 pointers since it is a base class.

2.1 Dynamic Cast Algorithm

C++ supports inheriting from multiple classes in parallel, essentially allowing arbitrary type hierarchy DAGs (not just trees!), including inheriting from the same class multiple times, and private inheritance (for which the algorithm cannot traverse the path when casting). This makes the algorithm for casting quite complex.

In this paper we focus on the simple case only: the type hierarchy must be a tree and all inheritance must be public. In Section 3.3, we show that this covers the vast majority of the casts in our benchmark programs.

For our example, the cast from A to B in `cast_to_B` works as follows. First, it loads the vtable of object a. Then, it compares the RTTI data pointed to by the vtable with B's RTTI data. Since they are the same, the cast succeeds.

In general, the algorithm may need to traverse the RTTI data's pointers from the dynamic type until finding the target type. For a hypothetical cast from A to B where the object is of type D (e.g., call '`cast_to_B(new D)`'), the algorithm first loads the vtable and obtains D's RTTI data. Since this is not equal to B's RTTI data, the algorithm has to continue to the parent's RTTI data. It now hits B's RTTI data, thus completing the cast. A failed cast will usually traverse the whole type hierarchy until the base class (but may be able to stop earlier; more details later).

2.2 RTTI Data Comparison

Comparing RTTI data is a key operation in the dynamic cast algorithm, as well as for exception handling. Perhaps surprisingly, there are three distinct approaches for implementing this comparison:

- Compare the addresses of the RTTI data
- Compare the addresses of the type name strings
- Compare the type name strings with `strcmp`

The third method is the easiest to implement, but it incurs in significant run-time overhead, especially when the program uses long class names and namespaces. On some platforms, loaders do not unify all symbols in a single global namespace because that is quite expensive. String comparison is a solution for comparing RTTI data across DSOs that does not require such unification.

Many platforms adopt one of the other two methods. Both of these methods compare just two pointers, making them more efficient. However, for these methods to work correctly, the compiler and the linker must ensure that the addresses of RTTI data/type names uniquely identify the type. That is, different classes must have distinct RTTI addresses, and vtables of the same class must point to the same RTTI data.

To see why ensuring a bijective relation between types and addresses is challenging, consider the following two files that define a class `A` in the anonymous namespace:

```
// file1.cpp
namespace { class A { ... }; }

// file2.cpp
namespace { class A { ... }; }
```

Classes declared in an anonymous namespace are visible only in the file where they are declared. Hence, a program may define classes with the same name in different files as long as they are defined in anonymous namespaces. However, because their definitions are identical, their RTTI data and type names are also identical.

Compilers and linkers usually merge identical constant global variables when their addresses are not meaningful (e.g., when they are marked with `unnamed_addr` in LLVM). Because the addresses of RTTI data and/or type strings may be significant, this optimization cannot always be done, requiring coordination between the compiler and the linker. An opposite example, where ensuring uniqueness of addresses of equal RTTI data is intricate, is given below:

```
// class.h
class A { virtual void foo() { ... } };

// file1.cpp; compile with: clang++ -shared file1.cpp -o lib1.so
#include "class.h"

// file2.cpp; compile with: clang++ -shared file2.cpp -o lib2.so
#include "class.h"
```

Here, we build two shared libraries (`lib1.so` and `lib2.so`), each containing a copy of class `A`. If we then have a binary that loads both shared libraries, the run-time linker needs to ensure that the RTTI data is merged. Comparing type strings makes this case trivial.

2.3 Implementing Dynamic Casts: Itanium ABI

We now briefly present how dynamic casts are implemented on Linux (Itanium ABI) as an example. For each dynamic cast, the compiler introduces a call to a run-time library function, implemented in libstdc++ (GCC) and libc++ (LLVM). The interface of this function is as follows:

```
void* __dynamic_cast(const void *obj,
                     const __class_type_info *static_type,
                     const __class_type_info *dst_type,
                     std::ptrdiff_t static2dst_offset);
```

Consider again the following code snippet:

```
B* cast_to_B(A *obj) {
  return dynamic_cast<B*>(obj);
}
```

The cast library function gets called with the following arguments: (1) `obj`, (2) a pointer to `A`'s RTTI data, (3) a pointer to `B`'s RTTI data, and (4) `0` (assuming a public tree-shaped type hierarchy).

The last argument (`static2dst_offset`) is a hint given by the compiler to the runtime library. It is meant to speed up the casting algorithm, and can take one of the following values:

Table 1. Programs used for benchmarking, including their number of lines of code (LoC), the number of dynamic casts and the ‰ of dynamic casts vs the total number of IR instructions (after standard -O2 optimizations), the total number of classes, the number of leaf classes (no class inherits from them), and the % of leaf classes annotated with the `final` keyword. In the dynamic casts column, the numbers in parenthesis indicate the number of converted dynamic casts for programs that do not use C++'s `dynamic_cast`.

Program	Description	kLoC	Dyn Casts	Classes	Leaf Classes (final)
Blender	3D graphics	1,883	817 / 0.08‰	5,939	5,387 (17%)
deal.II	Differential equations	94	27 / 0.21‰	109	74 (0%)
Envoy	Distributed proxy	859	420 / 0.05‰	17,017	15,162 (0%)
OMNeT++	Event simulator	27	16 / 0.26‰	104	84 (0%)
POV-Ray	Ray tracer	113	89 / 0.24‰	546	431 (82%)
Solidity	Compiler	419	1,368 / 0.60‰	638	510 (25%)
Z3	Theorem prover	516	100 / 0.07‰	1,489	1,227 (0.2%)
Chromium	Web browser	45,168	(15,441) / 0.71‰	44,505	37,374 (27%)
LLVM	Compiler	2,334	(77) / 2.05‰	2,616	2,138 (17%)
V8	JavaScript compiler	3,768	(1,438) / 0.31‰	3,402	3,049 (57%)

- ≥ 0: The static type is a unique public non-virtual base type of the destination type. The value indicates the number of bytes between the static and destination object layouts.
- -1: No hint.
- -2: The static type is not a public base of the destination.
- -3: The static type is a multiple public base type but never a virtual base type.

Given that negative hints are uncommon (c.f. Section 3.3), we ignore them in the rest of the paper. Therefore, we ignore class hierarchies with virtual, private, and multiple inheritance.

A dynamic cast succeeds iff there is a path from the actual (dynamic) type of the object and the destination type, crossing only public edges. Given that we only consider non-negative hints, the destination type always points upward towards a public base class. This ensures that there is always at least one path from the static type to the destination type. Hence, the algorithm just needs to traverse the path (through the RTTI data pointers) from the dynamic type until the static type. If the destination type is found along the way, the cast succeeds, otherwise it returns a null pointer. If the hint is positive (in some cases of multiple inheritance), it is added to the `obj` pointer.

What we just described works for the simple case (tree-shaped type hierarchies). The actual implementations usually have several algorithms (three, in the case of the Itanium ABI), each specialized for a different kind of inheritance. This allows more efficient implementations for the common cases. The different algorithms are exposed through the RTTI vtable, hence a dynamic cast entails a lot of pointer chasing: (1) loading the vtable of the object, (2) loading the RTTI data of the class, (3) loading the vtable of the RTTI data, and (4) do an indirect call to the cast implementation.

3 Dynamic Casts in the Wild

In this section, we analyze how dynamic casts are used in the wild. We selected 10 programs of different domains and sizes, as listed in Table 1. In total, we consider 55 million lines of code.

We observe that dynamic casts are not frequent, accounting for less that 0.1% of the total number of instructions. The percentage of leaf classes is very high, but the usage of the `final` keyword varies significantly across programs. This keyword helps the compiler optimize, e.g., virtual method calls.

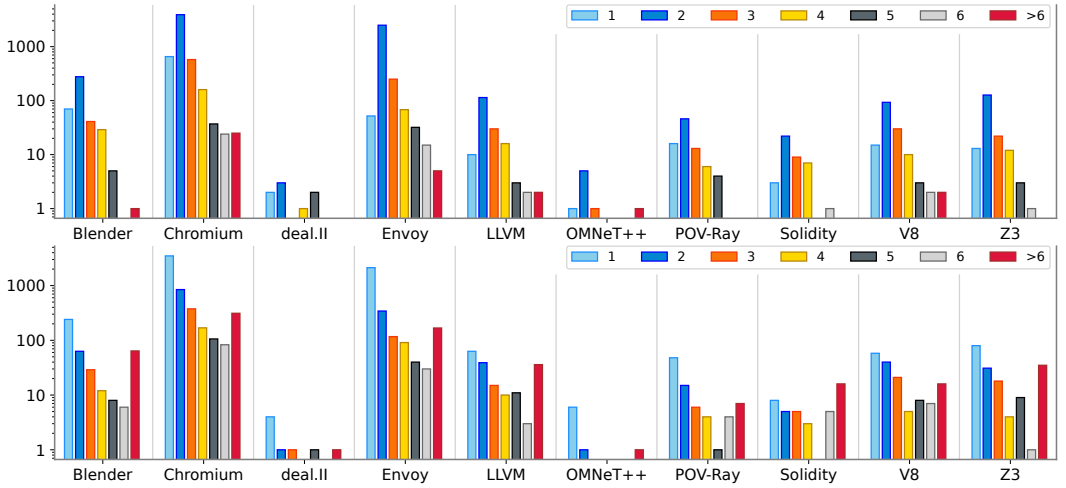


Fig. 2. Distribution of the height (top) and width (bottom) of the inheritance trees per program (log scale).

The three largest programs we consider compile *without* RTTI, and therefore they cannot use C++'s `dynamic_cast` mechanism. Instead, they deploy their own casting mechanism or use static casts only. LLVM, for example, requires all classes to have an additional integer field to store the class id, as well as some extra methods to implement its own `dyn_cast`.[3] Below is an example of the code required for the Alloca instruction class and part of its type hierarchy:

```
// Class Value
const unsigned char SubclassID;
unsigned getValueID() const { return SubclassID; }

// Class Instruction (inherits from Value)
unsigned getOpcode() const { return getValueID() - InstructionVal; }
bool classof(const Value *V) { return V->getValueID() >= Value::InstructionVal; }

// Class Alloca (inherits from Instruction)
bool classof(const Instruction *I) { return I->getOpcode() == Instruction::Alloca; }
bool classof(const Value *V) { return isa<Instruction>(V) && classof(cast<Instruction>(V)); }
```

The advantages are obvious: the implementation is extremely efficient, requiring a single integer comparison to determine whether an object of type `Value` is an instruction. Moreover, LLVM is able to cleverly pack all class ids in a 1-byte field (which contrasts with RTTI which occupies 8 bytes per object for the vtable pointer). Also, 1-byte fields can often be placed for free in the padding of classes. The disadvantages are also obvious: each class requires additional code and packing all classes within an 8-bit integer requires clever planning (note that although the field can only represent 255 ids, LLVM has 2.6k classes!).

3.1 Inheritance and Type Hierarchies

We now explore how inheritance is used and what the type inheritance trees look like in the benchmark programs. Fig. 2 shows the distribution of the height (top) and width (bottom) of the inheritance trees in log scale. Both metrics impact the running time of dynamic casts. Short trees dominate, with height up to 4 being the most prevalent. Nevertheless, most programs have a few

[3] https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html

Table 2. Statistics on type hierarchies and inheritance DAGs: number of polymorphic DAGs (in parentheses the % of LTO-internal DAGs), number of DAGs used in dynamic casts and in exception handling, and number of classes of each inheritance kind (virtual/public/private/multiple).

Program	Type DAGs (Internal)	Dyn Cast	EH	Virtual	Public	Private	Multiple
Blender	423 (98%)	35	2	2	5,394	93	89
deal.II	8 (100%)	1	0	8	99	1	5
Envoy	2,917 (100%)	135	2	457	13,756	254	793
OMNeT++	8 (100%)	2	2	0	96	0	0
POV-Ray	85 (100%)	9	3	1	449	1	25
Solidity	41 (50%)	18	2	73	550	25	91
Z3	178 (100%)	22	2	0	1,288	21	36
Chromium-M	5,382 (100%)	935	0	83	38,906	211	4,633
LLVM-M	177 (100%)	1	0	0	2430	11	149
V8-M	155 (100%)	18	1	17	3,228	19	82

Table 3. Statistics on the size of the type information data (% of the binary size), size of the type name strings (% of the binary size), % of unused type information, % of unused type name strings.

Program	Type Info (KB)	Type Strings (KB)	Unused Type Info	Unused Type Strings
Blender	152 (0.08%)	949 (0.49%)	76.7%	76.0%
deal.II	2.7 (0.37%)	3.4 (0.47%)	1.8%	1.8%
Envoy	528 (0.44%)	2,966 (2.48%)	60.9%	60.4%
OMNeT++	2.5 (0.27%)	1.7 (0.18%)	5.2%	0%
POV-Ray	13 (0.32%)	25 (0.60%)	66.3%	66.0%
Solidity	29 (0.18%)	108 (0.69%)	28.1%	27.9%
Z3	39 (0.15%)	62 (0.24%)	70.1%	70.0%
Chromium-M	1,259 (0.06%)	2,618 (0.13%)	59.1%	59.6%
LLVM-M	75 (0.15%)	213 (0.44%)	83.4%	83.3%
V8-M	92 (0.06%)	322 (0.20%)	68.6%	68.3%

(< 10) trees with height of 6 or more. Width measures the number of classes that derive from the same class. Although having just one derived class is the most frequent case, having more than six derived classes is the second most common case for many programs. This means that ideally the casting algorithm should take constant time on the width of inheritance trees.

Table 2 presents some statistics on type hierarchy DAGs. Most hierarchies are LTO internal (more on this later), which is fundamental for optimizations. Also, the percentage of DAGs used in dynamic casts and exception handling is very small, which indicates that most RTTI data is never used. The `typeid` operator is rarely used, with only Envoy (2), OMNeT++ (3), POV-Ray (1), and Z3 (4) using it (in parentheses the number of DAGs referenced). Public inheritance dominates, and multiple inheritance is uncommon except in Chromium where it is used by over 10% of the classes.

LLVM does not use C++'s `dynamic_cast`. We changed only the biggest type tree (rooted on Value) to use dynamic casts instead of using LLVM's own casting mechanism.

Table 4. Dynamic profile of dynamic casts: number of executed dynamic casts in millions, success rate (i.e., % of cases for which the cast did not return null), max/average number of RTTI data comparisons within the casting algorithm, max/average height of the inheritance sub-graph considered for casting (distance between the dynamic and the static types), % of the running time spent in dynamic casts (approximated, measured with `perf`), and % of casts with zero/negative/positive hints given to the dynamic cast runtime function.

Program	Casts	Success	Max/Avg Cmp	Max/Avg Height	Time	Zero/Neg/Pos hint
Blender	0.206	53.1%	12 / 5.42	4 / 3.04	0%	~100% / ~0% / 0
deal.II	181.8	100%	3 / 1.91	4 / 3.97	1.3%	100% / 0 / 0
Envoy	65.6	99.996%	12 / 2.50	4 / 1.46	0.02%	~100% / ~0% / ~0%
OMNeT++	40.8	100%	3 / 2.36	4 / 3.65	0.5%	100% / 0 / 0
POV-Ray	0.001	100%	12 / 4.46	5 / 2.84	0%	100% / 0 / 0
Solidity	3,278	8.7%	18 / 7.40	4 / 2.77	25%	79.6% / 20.4% / 0
Z3	0.037	100%	1 / 1.00	3 / 2.00	0%	100% / 0 / 0
Chromium-M	1.35	100%	29 / 2.10	8 / 2.01	4.5%	97.0% / 0 / 3.0%
LLVM-M	1,086	24.3%	36 / 18.2	6 / 3.20	59%	100% / 0 / 0
V8-M	423.2	100%	4 / 1.01	8 / 6.97	4.9%	100% / 0 / 0

3.2 RTTI Overhead

Table 3 presents the size of RTTI data (ignoring alignment and padding), as well as the percentage of such data that is provably never accessed by dynamic casts or exception handling (computed statically). RTTI data occupies less than 1% of the binary size of all programs except for Envoy, where RTTI occupies almost 3% (just the type strings occupy almost 3 MB).

Two thirds of the programs have more than half of the RTTI data unused. Programs deal.II and OMNeT++ have the least amount of unused RTTI data because the type DAGs that their casts and exception handling operate on dominate the total number of classes.

3.3 Dynamic Cast Usage

Table 4 shows the dynamic profile of the usage of dynamic casts by the benchmarks. The LLVM and Solidity compilers execute billions of casts, making the time spent on them highly significant (note that LLVM was modified to use `dynamic_cast`). These programs also exhibit a low cast success rate, as opposed to the remaining benchmarks for which the casts succeed most of the times.

We note that the number of RTTI data comparisons within the casting algorithm can be higher than the height of the considered sub-graph for two reasons: (1) multiple inheritance (the algorithm must follow several paths), and (2) the dynamic cast algorithm has several fast paths that lead to the evaluation of the same RTTI data multiple times if the fast paths fail.

4 Optimizing Dynamic Casts

Optimizing dynamic casts is challenging in the traditional compilation setting, where each file is compiled separately. Consider the following example:

```
// file1.h
class A { virtual void foo() { ... } };
class B : public A { ... };

// file1.cpp
#include "file1.h"
B* cast(A *obj) {
```

```
  return dynamic_cast<B*>(obj); // obj is of type B or ...?
}

// file2.cpp
#include "file1.h"
class C : public B { ... };
```

When compiling `file1.cpp`, one might be tempted to optimize the dynamic cast knowing that `obj` can only be of type `A` or `B`. If that was the case, the compiler could replace the dynamic cast with a comparison, for example of the vtable pointers as follows: `*(char**)obj == vtable_of_B` (assuming that vtables uniquely identify the type).

In the case above, we have another class derived from `B` defined in a different file. Hence, the optimization we described is illegal. The compiler never knows if the type hierarchy is extended in another file or not (except if the classes are defined in the anonymous namespace – thus only visible in the file).

4.1 Link-Time Optimizations (LTO)

An alternative to compiling each file individually is using link-time optimizations (LTO) [28, 31, 41, 42, 62, 68], where each file is first compiled into an intermediate representation (LLVM IR in the case of Clang) instead of producing assembly right away. Then, the linker invokes the compiler with the IR of all files, which then optimizes the whole program at once.

The main benefit of LTO is that it enables inter-procedural optimizations (IPO) across files. IPO algorithms take advantage of symbols are that internal to the LTO unit, which means that they are not accessible from outside of the unit, to compute the set of all users of each internal symbol. LTO can be used to build dynamic libraries, as programs can still have externally-visible symbols, including the `main` function and the public API of a library. However, optimizations will skip external symbols (e.g., dead code elimination cannot remove a symbol if it is externally accessible).

We measured the percentage of classes internalized with LTO in our benchmarks, and it ranges from 98% to 100%. Our LLVM benchmark program is the optimization binary (`opt`). For comparison, we built LLVM as a dynamic library (`libLLVM.so`), and observed that only 24% of classes are internalized. This is because LLVM exposes a C++ API containing many classes.

In this work, we leverage LTO to obtain internal type hierarchies, where all types in the hierarchy are internal, to implement the optimization described above in a sound way.

4.2 Algorithm For Optimizing Dynamic Casts with LTO

The key observation that underpins our optimization is that if the type hierarchy is known, we can do partial evaluation of the dynamic casting algorithm at compile time, and produce an optimized code sequence that implements the remaining bits of the algorithm.

Let s be the static type of the cast (`A` in our running example), t the target type of the cast (`B` in the example), d the dynamic type of the object we want to cast, and o the offset/hint.

As we have seen before, if $o \geq 0$, the cast succeeds iff there is a path from d to t, otherwise it returns null. Hence, if the DAG below the target type is fixed, we implement the check for the existence of a path with a set membership check. Let T be the set of the target type and its derived classes ($T = \{\texttt{B}, \texttt{C}\}$ in the example), the cast succeeds iff $d \in T$.

The set membership check must be implemented according to the platform's ABI (recall Section 2.2). Our prototype targets the Itanium ABI, which does not ensure each type has a unique vtable. We could compare the pointers to RTTI data instead, since it is guaranteed to be unique, but we opted to change Clang to guarantee vtable uniqueness. This consists in removing the

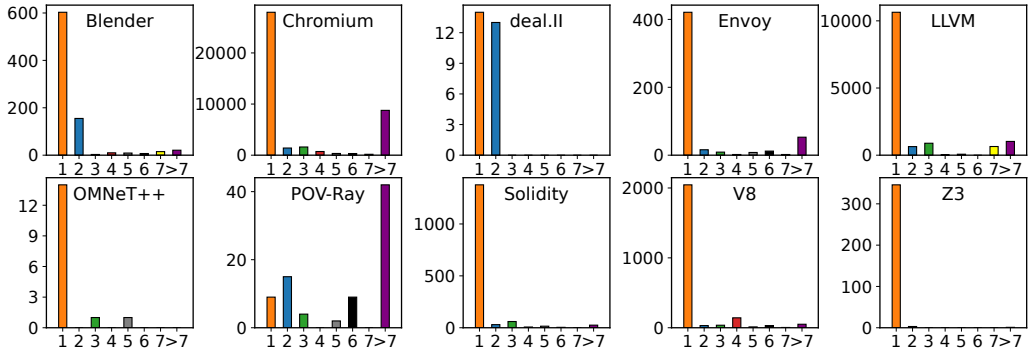


Fig. 3. Distribution of the number of candidate types (i.e., the size of T) per dynamic cast.

`unnamed_addr` attribute from vtables to prevent the compiler from merging identical vtables. Since identical vtables are rare, the impact is negligible.

Clang produces the following IR for the example function (note the explicit check for null pointers since dynamic cast is well defined for such pointers):

```
define ptr @cast(ptr %obj) {
entry:
  %isnull = icmp eq ptr %obj, null
  br i1 %isnull, label %ret, label %cast
cast:
  %dyncast = call ptr @__dynamic_cast(ptr %obj, ptr @_ZTI1A, ptr @_ZTI1B, i64 0)
  br label %ret
ret:
  %r = phi ptr [ %dyncast, %cast ], [ null, %entry ]
  ret ptr %r
}
```

Our optimization replaces the call into the library function with the following code sequence:

```
cast:
  %obj_vtable = load ptr, ptr %obj, align 8
  %isB = icmp eq ptr %obj_vtable, getelementptr inbounds (i8, ptr @_ZTV1B, i64 16)
  %isC = icmp eq ptr %obj_vtable, getelementptr inbounds (i8, ptr @_ZTV1C, i64 16)
  %success = or i1 %isB, %isC
  %dyncast = select i1 %success, ptr %obj, ptr null
  br label %ret
```

The code works as follows (refer to Fig. 1 for the memory layout). First, we load the vtable pointer of the object. Then, the vtable pointer is compared against the vtables of classes `B` and `C` (note that an object's vtable pointer points to the vtable address point, which is 16 bytes offset from the start of the vtable data). If either of the comparisons succeeds, the same object pointer is returned. In the case of multiple inheritance, the offset must be added to the object pointer (it is zero in this case).

Although we increased the code size slightly, from 11 to 15 instructions on x86, skipping the library call reduces the run time significantly.

In terms of correctness of the optimization, it works for non-negative offsets and requires all classes in T to be LTO internal, which ensures that no shared library can define a class that inherits from a class in T.

4.3 Efficient Membership Checks for Large Type Hierarchies

Fig. 3 shows the distribution of the number of candidate types (i.e., the size of T) per cast. Although having a single candidate is the most common case, more than half of the casts in POV-Ray have more than 7 candidates. Ideally, we would rather not emit a comparison per candidate.

For the cases we consider, the type hierarchies are always trees. If we lay out the vtables in a pre-order traversal ordering, it is guaranteed that all sub-classes of a class are laid out consecutively after the class. This is implemented by merging each tree into a single global variable and then changing references to this new variable with the corresponding offset.

Checking if a class is of a given type can now be done with a range check: $b \leq p \leq e + 16$, where p is the object's vtable pointer, b is the address of the vtable of the target class t of the cast, and e is the address of the last derived class of t.

The complexity of dynamic casts is, therefore, improved to $O(1)$ instead of being a function of the size of the type hierarchy. We use the range check when there are 3 or more candidates.

The type hierarchy for classes with multiple inheritance is not a tree. For this case, we could decompose the graph into multiple trees, and use one range check per tree, as proposed by Bounov et al. [10], or attempt to produce a global ordering for the various trees as used in PQ-Encoding [30]. Since multiple inheritance in casts is not common, we decided to not support it.

4.4 Removing Unused RTTI Data

RTTI data is used by three functionalities: dynamic casts (for polymorphic classes), exception handling, and the `typeid` operator. For non-polymorphic classes, since the only way their RTTI data can be used is through direct reference, the dead code elimination (DCE) optimization of LLVM can already remove unused RTTI data of such classes.

However, for polymorphic classes, RTTI data can be accessed indirectly through the RTTI slot in the vtables. This means that if the vtable of a class is referenced, DCE cannot remove its RTTI data.

We devised an algorithm to remove unneeded RTTI data, which runs after the dynamic cast optimization (since it removes the need for RTTI data). First, we compute an over-approximation of the live RTTI data, and then any data not marked as live is removed.

There are five cases to consider. Any non-LTO internal DAG is marked as live, since they may be used by an external library. For any dynamic cast, the static type and all its subclasses are marked as live (this over-approximates the RTTI data accessed by the casting algorithm for any dynamic type). For exceptions, we mark as live each thrown class and its parents (since we can have a catch block for any parent class). For the `typeid` operator, computing liveness is more involved. Currently, Clang compiles the `typeid` operator into the following LLVM IR:

```
%vtable = load ptr, ptr %obj
%rtti_ptr = getelementptr inbounds i8, ptr %vtable, i64 -8
%rtti = load ptr, ptr %rtti_ptr
```

This means that the information about the static type of the object is lost. Recovering that information would be very complex, requiring inter-procedural alias analysis. Instead, we modified Clang to preserve the static type by emitting the following additional IR:[4]

```
%a = call i1 @llvm.type.test(ptr %vtable, metadata !"_ZTS1A")  ; RTTI mangled name
call void @llvm.assume(i1 %a)
```

Now that we have the static type for each `typeid` operator, we mark as live the static types and all their subclasses.

The last case is the special cast `dynamic_cast<void*>(obj)`, which is compiled into:

[4]This technique is also used by the implementations of the CFI protection and the devirtualization optimization.

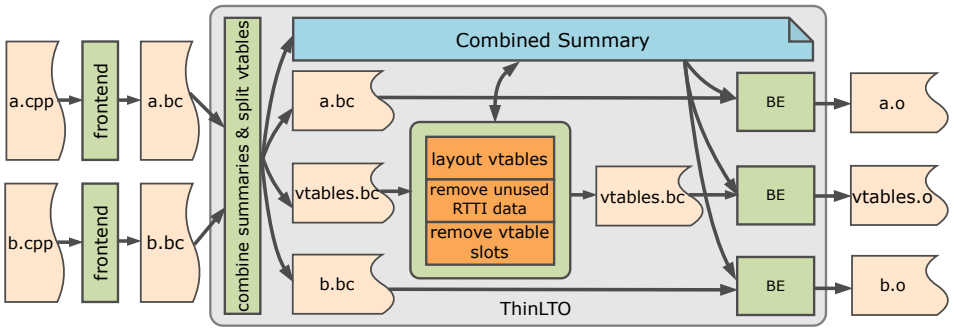


Fig. 4. Overview of the ThinLTO compilation process, including our extension to remove unused RTTI data and vtable slots (in orange). The backend (BE) optimizes each file using global knowledge about the program from the combined summary. It is also the BE that replaces dynamic casts with range checks (our optimization), and finally produces assembly.

```
%vtable = load ptr, ptr %obj
%offset_to_top_ptr = getelementptr inbounds i8, ptr %vtable, i64 -16
%offset_to_top = load i64, ptr %offset_to_top_ptr, align 8
%result = getelementptr inbounds i8, ptr %obj, i64 %offset_to_top
```

For this case, only the offset-to-top slot is live, but the type information is lost. We use the same `@llvm.type.test` mechanism as before to track which classes are live.

After computing the live RTTI data, we remove all the data that is not live. Since classes that have their RTTI data removed are not used in dynamic casts, we further remove the offset to top and the RTTI slots from vtables, saving an extra 16 bytes per class. Since the objects' vtable slot points into their vtable+16, we also need to patch the IR used for the `new` operator to skip the offsetting:

```
%obj = call ptr @_Znwm(i64 16)  ; allocate 16 bytes
store ptr getelementptr inbounds (i8, ptr @_ZTV1A, i64 16), ptr %obj, align 8  ; before
store ptr @_ZTV1A, ptr %obj, align 8  ; after
```

Although this transformation violates the C++ ABI, the transformation is still sound because it only works on classes that are internalized and thus the vtable layout is not meaningful.

4.5 Extension for Lightweight LTO (ThinLTO)

Regular LTO loads the IR of the whole program in memory during linking. For large programs, this is slow and leads to very high memory consumption (hundreds of GBs of RAM).

Recently, lightweight LTO algorithms have emerged, which avoid loading the whole program into memory at once. Instead, the compilation of each file produces a summary with information to be exchanged with other files (e.g., number of instructions of each function, to enable cross-file inlining). Then, at linking time, only the summaries of each file are loaded into memory and processed by inter-procedural analyses. Finally, the combined information is shared and each file is then optimized individually by taking into consideration the summarized global knowledge.

In LLVM, ThinLTO [41] implements such an algorithm. Although it is not as effective as regular LTO, the much faster compilation process makes it the default LTO build mode for large projects like Chromium. Fig. 4 gives an overview of the ThinLTO compilation process, as well as our extension to remove unused RTTI data and vtable slots (in orange).

By default, the flag '`-fsplit-lto-unit`' is enabled, which compiles each file into two modules (instead of just one). The split module contains only the vtable definitions and virtual functions

that do not access memory. The other module contains the remaining definitions (global variables and the other functions).

At link time, ThinLTO merges all split modules into a single module. The original motivation was to support the control-flow integrity (CFI) protection with type enforcement, which requires full knowledge of the type hierarchy. Fortunately, we require the same information, allowing us to leverage the same mechanism to implement the vtable layout algorithm.

Since the vtable layout algorithm merges vtables into a single global variable, we need to record in the summary the offsets for each vtable so each file can then adjust the references. For example, the following summary states that a dynamic cast with destination type `B` should do a range check between offsets 8 and 64 of the merged global:

```
rangemap: ((name: "B", (name: "A.merged", (offset: 8, offset: 64))))
```

For unused RTTI data elimination, we extend the per-file summary to contain information of the present dynamic casts, **`typeid`** operations, and excepting handling. Below is an example summary for a file with two dynamic casts, two uses of the **`typeid`** operator, and one **`throw`**:

```
dyncast: ((dst: "B", static: "A", hint: 0), (dst: "C", static: "A", hint: 0))
typeid: (type: "A", type: "B")
eh: (type: "A")
```

At link time, the summaries are combined to obtain the global usage of RTTI data. The linker determines which casts are guaranteed to be optimized away by the backends afterwards, and which are not (and thus need RTTI data alive).

As the RTTI data is not emitted into the split module, but rather scattered through the remaining modules, we extend the combined summary with RTTI liveness information.

5 Evaluation

We now assess whether the proposed optimization improves 1) the running time of our benchmark programs, 2) reduces the binary size, and 3) reduces the peak memory consumption.

Additionally, we modified LLVM (LLVM-M) so it uses C++'s **`dynamic_cast`** instead of using its own, hand-rolled, casting mechanism. The goal of this experiment was to understand the benefit of this manual optimization, and whether our optimization can realize the same benefits automatically.

For Chromium-M and V8-M, we changed Clang to compile static casts as if they were dynamic. These programs can be optionally compiled with safeguards to prevent security issues related with incorrect casts. We assess whether using optimized dynamic casts matches the security and performance of the existing hardening solution.

Experiments were run on a server running Ubuntu 24.04, with a 64-core AMD EPYC 9554P CPU with 768 GB of RAM. Table 5 lists the programs and the versions used for evaluation, as well as the benchmarking workloads. We used LLVM 18 as the baseline and to implement our optimization.[5]

5.1 End-to-End Performance, Binary Size, and Peak Memory Impact

Fig. 5 shows the end-to-end impact in terms of performance, binary size, and memory usage on our benchmark programs. All benchmarks get faster, except Z3 and Envoy, which show a slight regression. Solidity performs downcasts frequently and thus shows the largest improvement. deal.II also gets a significant speedup since all dynamic casts get replaced with 1 or 2 pointer comparisons.

In terms of size, all binaries become smaller due to the removal of unused RTTI data. We note that ThinLTO performs fewer inter-procedural optimizations and uses different inlining heuristics, which explains the differences in binary size.

[5]Code available at https://github.com/luxufan/llvm-project/tree/dyncastopt and benchmarks available at https://github.com/luxufan/cpp_dyncasts_benchmark.

Table 5. Programs used for evaluation, their version number, and the benchmarking workload used.

Program	Version	Evaluation Command
Blender	4.2	`tests/performance/benchmark.py`
Chromium-M	122.0.6249.0	Blink performance tests
deal.II	SPEC CPU 2006	SPEC's benchmark
Envoy	1.31	Envoy-perf's Siege performance script
LLVM-M	18.0.0	`opt -O2 z3.internal.bc` (optimize the LTO module of Z3)
OMNeT++	SPEC CPU 2006	SPEC's benchmark
POV-Ray	3.8.0	`povray -benchmark`
Solidity	0.8.26	`test/benchmark/external.sh`
V8-M	12.7.189	SunSpider 1.0.2 JavaScript benchmark
Z3	4.12	Five random files of SMT-LIB's QF_BV benchmarks

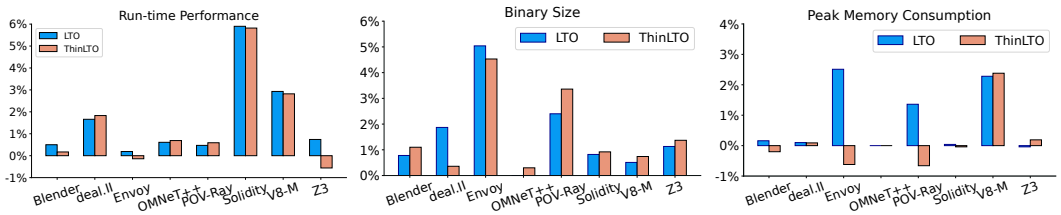


Fig. 5. End-to-end impact on performance (left), binary size (middle), and peak memory usage (right). Higher is better (benchmarks became faster, binary shrank, and the peak memory decreased).

Replacing dynamic casts with short code sequences impacts the inlining heuristic. Because the sequence is estimated to have a lower cost, the optimizer ends up inlining more. We observe up to 0.6% more calls getting inlined. Nevertheless, binaries still get smaller overall.

Besides the expected reduction of the `.rodata` and `.strtab` sections of the binaries (used, respectively, for the RTTI type info and RTTI type strings), another section shrinks as well. Because binaries are built by default in relocatable mode, each symbol in the program (including RTTI data) has an entry in the `.rela.dyn` section. Removing unused RTTI data also removes the corresponding relocation information. This explains the large binary size reduction.

Our optimizations reduces the peak memory usage of V8-M. This happens for two reasons: the optimized program does not need to load as much RTTI data into memory as before (because we replace `dynamic_cast` with pointer comparisons), and because we reduce the size of the vtables. Other benchmarks do not exhibit significant improvements because they allocate orders of magnitude more memory, overshadowing the memory savings achieved through our optimizations.

5.2 Benchmark Programs Profiling

Fig. 6 (left) shows the percentage of RTTI symbols removed by vanilla LLVM (without our optimization). We observe that LLVM can only remove up to 2.5% of the RTTI symbols. On the other hand, our optimization removes from 10% to 85% of the RTTI symbols (Fig. 6, middle). In absolute terms, our optimization shrinks the binary size by up to 2.4 MB (Chromium).

Fig. 6 (right) shows the percentage of vtables whose offset-to-top and RTTI pointer slots are removed by our optimization. It prunes more than half of the vtables in most programs, and prunes over 60% of vtables in several benchmarks. Pruning vtables has three additional benefits besides

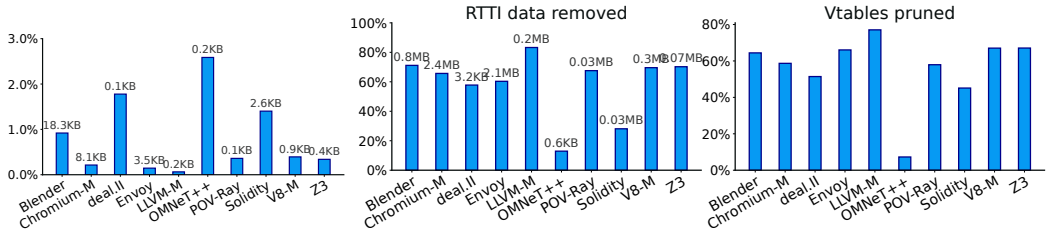


Fig. 6. On the left, the amount of RTTI data removed by vanilla LLVM (without our optimization). In the middle, the amount of RTTI data removed with our optimization. On the right, the percentage of vtables that have their offset-to-top and RTTI pointer slots removed by our optimization.

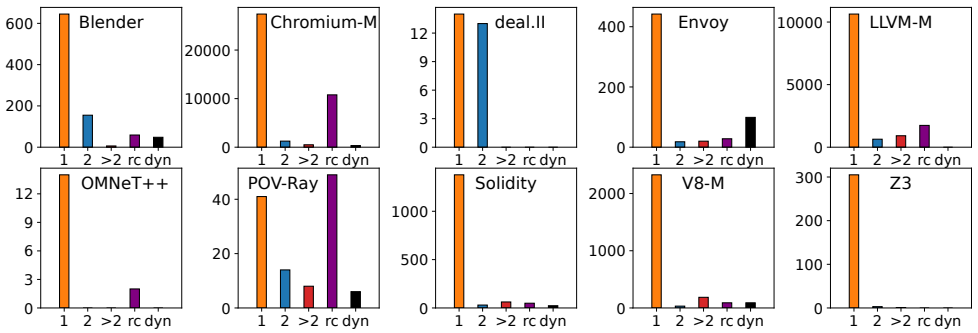


Fig. 7. Distribution of the dynamic casts optimization cases: replaced with 1, 2, or more checks, replaced with a range check (rc), or not optimized (dyn) (e.g., the cast uses a negative hint).

shrinking binaries. First, it reduces the number of symbol relocations, which in turns means faster application startup. Second, the memory usage of the program is reduced due to the smaller vtables, reducing the % of cache misses. Finally, the code sequence for the `new` operator gets slightly simpler.

The reasons why not all RTTI data is removed are: 1) not all dynamic casts can be removed (e.g., because they have a negative hint, which we do not support), 2) RTTI data of classes used in exception handling and `typeid` cannot be removed, and 3) we can only remove RTTI data that is LTO internal. For example, Solidity uses the Boost C++ library extensively. Since Solidity does not include Boost in its LTO module, but rather links with the static library, all classes inheriting from the Boost library are not LTO internal. Another example is OMNeT++, which uses `typeid` heavily, causing the compiler to conservatively mark over 80% of polymorphic classes' RTTI data as live.

Fig. 7 shows the distribution of the dynamic casts optimization cases. The first three bars indicate the number of casts replaced with 1, 2, or more checks. The fourth bar indicates the number of casts replaced with a range check (rc), which is used when there are 3 or more alternatives, except when the DAG is not a tree. The last bar measures the number of casts that were not optimized.

5.3 LLVM Case Study: Hand-Rolled Casts vs `dynamic_cast`

Because of the poor performance of `dynamic_cast`, LLVM implements a custom casting mechanism and metadata. LLVM reserves an 8-bit field in each objet to store the type id and requires each class to have a couple of methods to support its casting mechanism.

To assess whether using C++'s dynamic casts with our optimization roughly matches the performance of LLVM's hand-made solution (thus not needing it anymore), we changed LLVM to use

dynamic casts instead.[6] We took the largest type tree in the code base (rooted in the `Value` class), and changed it to use standard C++ features only. We first describe the changes made.

The first change was to make the `Value` class polymorphic, as that is a requirement to use `dynamic_cast`. We did so by adding a dummy virtual function. This step alone increases the peak memory consumption of LLVM by 4% and causes a 1% performance degradation due to the extra 8 bytes used for the vtable slot in every object.

The second change was to replace all uses of the custom casting functions (`isa` and `dyn_cast`) with `dynamic_cast`. Since there are too many such places, we opted instead to change the `classof` method of each class (called by the custom casting functions) to use `dynamic_cast`. Although we only replaced 77 sites, after inlining during compilation there are more than 10k uses of `dynamic_cast`.

The third change consisted in removing the type id field from the `Value` class. This change is tricky because the type id is used in `switch` statements to efficiently dispatch based on the type (instead of having a linear sequence of dozens of dynamic casts). We considered two options: 1) change the `getValueID` method to be virtual, and 2) subtract vtable addresses.

Below is an example of the changed method for the `LoadInst` class for the first option.

```
class Value { virtual unsigned getValueID() const = 0; };
class LoadInst : public Instruction {
  unsigned getValueID() const override { return Instruction::Load; }
};
```

The second option consists in computing the difference between the object's vtable pointer and `Value`'s vtable address. This works because we layout vtables consecutively. The code is as follows:

```
unsigned Value::getValueID() {
  char *vtable = *(char**)this;
  return (vtable - _address_of_Value_vtable) / 8;
}
```

We cheated a bit by taking advantage of the knowledge we have about the vtables' layout. The goal was to assess whether C++ should offer a solution for type switching. Since we pruned vtables to remove the RTTI and offset-to-top slots, all vtables have only 8 bytes.

A further complication is that not all LLVM types have a corresponding class. For example, all binary operators such as addition and multiplication instructions are represented by the class `BinaryOperator` although they have different type ids. We decided to change the `Instruction::getOpcode` method as follows:

```
unsigned Instruction::getOpcode() {
  if (auto *BO = dynamic_cast<BinaryOperator>(this))
    return BO->getOpcode();
  return getValueID() - InstructionVal;
}
```

Fig. 8 shows the impact on performance, peak memory consumption, and binary size compared to the unmodified LLVM for the two options just described, with and without our optimization. The solution based on the difference of vtable pointers has the best performance, being about 5% slower than LLVM's custom solution, while the virtual solution is over 20% slower (with our optimization). Without our optimization, the slowdowns are unbearable, which justifies why LLVM had to roll its own casting mechanism.

The slowdown of 5% is mainly due to LLVM's original `classof` methods in the `Constant` and `Instruction` classes having a single integer comparison, while our solution using dynamic casts compiles these methods into a range check (i.e., two comparisons). Without range checks (i.e.,

[6] Modified version of LLVM (LLVM-M) that uses `dynamic_cast` instead of its own casting mechanism: https://github.com/luxufan/llvm/tree/llvm-case-study-diff. We patched LLVM to output the vtables in the right order so we could use range checks, available at https://github.com/luxufan/llvm/compare/main...llvm-case-study-opt.

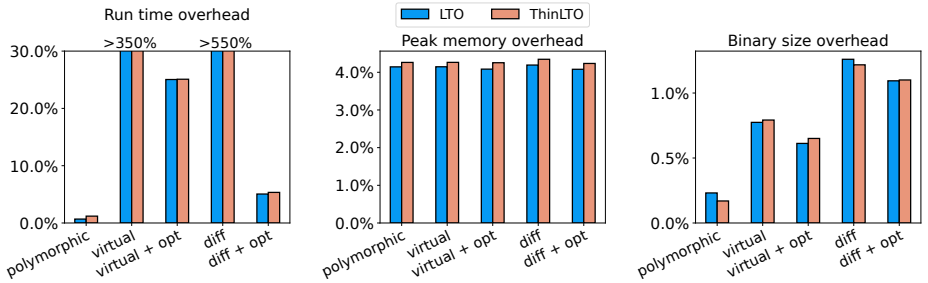


Fig. 8. Impact on performance, peak memory consumption, and binary size of the two alternatives for converting LLVM's `Value` type tree to use `dynamic_cast`, with and without our optimization. The first bar refers to changing the `Value` class to be polymorphic, virtual refers to the solution of making the `getValueID` method virtual, and diff refers to the solution of using the difference between vtable pointers. Lower values are better. The baseline is unmodified LLVM.

using one comparison per type candidate), the slowdown increases to 10%. The fact that LLVM manages to use a single comparison suggests that a cleverer layout of classes could be used to have dynamic casts compiled into a single comparison as well.

Overall, we believe that using C++ dynamic casts, together with a new type switching mechanism and our optimizations could replace the hand-rolled casting mechanisms used in large programs.

5.4 Chromium Case Study: Hardening Static Casts

C++ offers both static and dynamic casts. Static casts are free, but the compiler only validates upcasts, since that can be done easily at compile time. For downcasts, there is no checking at compile or run time. On the other hand, dynamic casts are always checked.

Incorrect static downcasts can lead to security vulnerabilities, namely type confusion [34, 39, 49, 58, 75]. The security impact of such vulnerabilities are significant, ranging from mere crashes to arbitrary code execution. Because of the potential hazards, Clang offers a static cast hardening mechanism through the '`-fsanitize=cfi-derived-cast`' flag. This mechanism instruments casts to check that the vtable of the object is one of the expected ones.

We briefly illustrate how this hardening works. Consider the following C++ code fragment:

```
struct A { virtual ~A() {}; };
struct B : public A {};
struct C : public B {};
B* cast_to_B(A *obj) { return static_cast<B*>(obj); }
```

Clang produces the following LLVM IR:

```
@vtables = constant { [4 x ptr],  [4 x ptr] }
                    { [4 x ptr] [ptr null, ptr @_ZTI1B, ptr @_ZN1AD2Ev, ptr @_ZN1BD0Ev],
                      [4 x ptr] [ptr null, ptr @_ZTI1C, ptr @_ZN1AD2Ev, ptr @_ZN1CD0Ev] }
define ptr @cast_to_B(ptr %obj) {
  %vtable = load ptr, ptr %obj, align 8
  %vtable_int = ptrtoint ptr %vtable to i64
  %sub = sub i64 %vtable_int, ptrtoint (ptr getelementptr (i8, ptr @vtables, i64 16) to i64)
  %s1 = lshr i64 %sub, 5
  %s2 = shl i64 %sub, 59
  %rot = or i64 %s1, %s2
  %ok = icmp ule i64 %rot, 1
  br i1 %ok, label %ret, label %notok
```

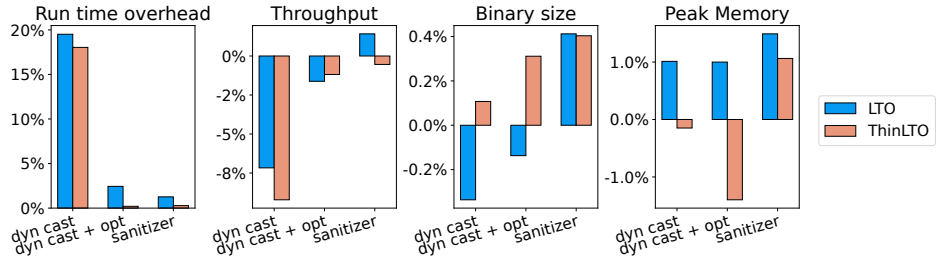


Fig. 9. Impact of compiling Chromium with static casts converted to dynamic with and without our optimization, and with Clang's cast hardening, in terms of total running time and throughput of the Blink benchmarks, binary size, as well as peak memory consumption. Blink has 57 performance (first plot) and 22 throughput tests (Second plot). The Baseline version is unmodified Chromium. Higher is better except for the first plot (run-time overhead increases, throughput increases, and binary size and peak memory decrease).

```
  ...
}
```

In a nutshell, the IR above checks if the object points to B's or C's vtables. This ensures that the cast is safe. The check is implemented using a clever rotation of the difference of the object's vtable pointer and the address of the first vtable (in this case B's). The check requires the vtables to be laid out in memory in a specific order (similar to our algorithm). The IR is equivalent to naively checking each vtable (in this case, `obj_vtable_slot == vtable_b || obj_vtable_slot == vtable_c`).

Chromium, being a security-sensitive application, supports building it with this Clang cast hardening flag [61]. An alternative to this hardening is to use dynamic casts. Since dynamic casts return null on failure, and that any memory access through the null pointer crashes the program, the security guarantees of using dynamic casts and Clang's hardening flag are similar. Therefore, we modified Clang to change static casts into dynamic casts and compiled Chromium in this mode. We wanted to assess whether using dynamic casts coupled with our optimization delivers a solution that is competitive with Clang's hardening instrumentation.

Fig. 9 compares the performance and binary size when using static casts (the baseline), when static casts are converted to dynamic casts with and without our optimizations, and when using Clang's hardening. Using dynamic casts without our optimization causes a large slowdown (~20%). However, our optimization recovers most of the overhead, offering a solution that is competitive in terms of performance, while producing a binary slightly smaller than Clang's hardening.

Using dynamic casts offers similar security guarantees to Clang's hardening, since dereferencing a null pointer in user-space code poses low risk. However, our optimization uses a range check when there are more than 2 type candidates. This check is less tight than Clang's, since we allow the object's vtable pointer to point into any slot of the vtables (e.g., the RTTI pointer or any virtual function), while Clang's code only allows pointers into the right vtable slot. It is unclear if ROP attacks similar to type confusion could be done by pointing into the suffix of a vtable of a valid type (cf. [64, 72] for a review on ROP attacks through vtables). It is also possible to use a check similar to Clang's hardening, albeit at a slight performance impact.

6 Related Work

We briefly survey the work on improving the performance and safety of dynamic casts. Table 6 summarizes the techniques for implementing dynamic casts, comparing them in terms of space and time complexity, whether they support multiple and virtual inheritance, whether they support open type hierarchies (e.g., do not require LTO), and whether they are transparent to users.

Table 6. Summary of the techniques for implementing dynamic casts, including their space and time complexity, whether multiple and virtual inheritance are supported, whether open type hierarchies are supported, and whether they are transparent to users. Let C be the number of classes, E the number of edges in the type DAGs, M the number of objects, h the height of the type hierarchy DAGs, and k the number of sub-trees in the type hierarchy DAGs ($k = 1$ if there is no multiple inheritance).

Technique	Space	Time	Multi	Virtual	Open	Transparent
C++ RTTI	$O(C+E)$	$O(h)$	✓	✓	✓	✓
MemCast [52]	$O(C+E)$	$O(h)$	✓	✓	✓	✓
FailFast [55]	$O(C+E)$	$O(h)$	✓	✓	✓	✓
FastCast [29]	$O(C)$	$O(1)$	✓			✓
LLVM's custom RTTI	$O(M)$	$O(1)$				
Schubert et al. [63]	$O(C)$	$O(1)$				✓
Cohen's encoding [15]	$O(h \cdot C)$	$O(1)$				✓
Jalapeño's encoding [5]	$O(h \cdot C)$	$O(1)$				✓
Packed encoding [71]	$O(k \cdot h \cdot C)$	$O(k)$	✓			✓
PQ-encoding [30]	$O(k \cdot C)$	$O(k)$	✓			✓
Graph coloring [46]	$O(C)$	$O(\log C)$				✓
Perfect hashing [27]	$O(C)$	$O(1)$	✓		✓	✓
ESE [4]	$O(C)$	$O(k)$	✓		✓	✓
Our optimization	$O(1)$	$O(k)$				✓

MemCast [52] uses a cache per cast to speedup repeated casts. FailFast [55] uses bloom filters to avoid iterating the whole type hierarchy when casts fail.

FastCast [29] encodes each class type as an integer and performs type checks in constant time. The id of each class is computed as the product of the ids of its base classes' ids and a unique prime number. Casting works by checking if the id of the object's type is divisible by the id of the destination type. This approach requires internal type hierarchies and is limited to small type hierarchies trees. It has been used by embedded systems [19, 20].

Schubert et al. [63] assign an integer interval to each class such that it is included in all of its superclass' intervals. Type checking is implement via an interval inclusion test. The R&B algorithm uses similar range checks for Java sub-type tests [57].

Cohen's encoding [15] records the distance to the root of the type hierarchy tree in each object. Type checking consists in a single lookup in the class' array by the distance tag of the object. Packed encoding [71] extends Cohen's encoding to support multiple inheritance by splitting type DAGs into multiple disjoint type subsets. Jalapeño's encoding [5] extends Cohen's encoding with two additional data structures to speedup the common cases in Java programs.

PQ-Encoding [30] extends Schubert et al. to support multiple inheritance. It uses PQ trees for efficiently computing intervals that satisfy the global interval inclusion property for type DAGs. If no such ordering can be found, it splits the DAG into multiple sub-trees akin to packed encoding.

Perfect hashing has been proposed as an efficient way to do sub-type checks [27]. It computes a hash value for each class such that all sub-type checks can be implemented with short code sequences. It supports open type hierarchies by recomputing the hashes when the program starts. ESE [4] reduces the time to add a new class by increasing the time to do sub-type checks (one per graph slice). An alternative way of finding an optimal encoding is to use graph coloring [46].

Mach7 [65] is an implementation of type switching for C++ that supports open type hierarchies. It uses dynamic casts and memoization to speed up subsequent uses. Inline caches are used to precede indirect calls with a sequence of tests for the most frequent targets [2, 12, 14, 17, 23, 36, 67, 74].

Unlike most of the techniques just described that use custom metadata, our algorithm leverages the vtable slot that is already present in all C++ polymorphic objects. Hence, our technique requires no additional memory or binary space. Moreover, our algorithm removes most RTTI data linked from vtables, thus offering dynamic casts at a very low cost and without major changes to compilers.

Link-Time Optimizations (LTO). LTO is now implemented in many C++ compilers, including GCC [31, 50], LLVM [41], and MSVC [62]. Several optimizations leverage LTO to obtain a closed world [6, 13, 32, 68, 73, 76]. In particular, devirtualization [3, 7, 18, 28, 56] attempts to replace virtual calls with direct calls. Inter-procedural alias analysis can be used to remove dynamic casts, since it allows one to potentially store-forward the vtable pointer of objects [9, 22, 33, 35, 43, 48, 60].

Security Hardening. Type confusion is a long-standing security problem in C++ programs [45, 66]. Clang's undefined behavior sanitizer (UBSan) [51] enforces cast safety by converting static casts to dynamic casts.

Other cast hardening techniques use custom metadata instead of relying on RTTI data in order to support programs that are compiled without RTTI [24–26, 34, 39, 47, 49, 58, 59]. Libcrunch [44] enforces cast safety by checking pointers at creation, using per-allocation type metadata. T-Prunify [75] leverages hand-written type checks using custom type information as used in, e.g., Chromium and LLVM, to prove that some casts are safe. Must [37] checks the types of buffers given to MPI operations (which are not typed) by keeping a map to store metadata about memory allocations. Various C++ dialects [8, 21, 40, 53] have been proposed to mitigate type confusion during type casting operations.

Control-flow integrity (CFI) [1, 54, 70] is a mechanism to protect indirect calls (including through vtables). There are also mechanisms to ensure integrity of the vtable slot of objects [10, 11, 38, 70]. TRaP randomizes the layout of vtables to prevent ROP attacks [16].

7 Conclusion

Dynamic casts are one of the few features in C++ that violate the language's design goal of having users pay only for the features they use. This is because in the traditional compilation setting, where each file is compiled individually, compilers operate without knowing the full type hierarchy and thus must always emit run-time type information (RTTI) just in case.

In this paper, we present a novel optimization for dynamic casts in C++ programs. We leverage the advances in link-time optimizations (LTO) of the past decade to obtain an internal type hierarchy. We show that our optimization replaces most dynamic casts with short code sequences with constant-time complexity (unlike the runtime library implementations, which are linear in the size of the type DAGs). Our optimization also removes most of the unused RTTI data, hence restoring C++'s design goal of having users only pay for the features they use.

Acknowledgments

The authors thank Richard Smith for providing the example with anonymous namespaces of Section 2.2, and Teresa Johnson and Reid Kleckner for feedback on earlier drafts.

This work was supported in part by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), and an unrestricted gift from Google.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *CCS*. https://doi.org/10.1145/1102120.1102165

[2] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. 2014. Improving JavaScript performance by deconstructing the type system. In *PLDI*. https://doi.org/10.1145/2594291.2594332

[3] Gerald Aigner and Urs Hölzle. 1996. Eliminating Virtual Function Calls in C++ Programs. In *ECCOP*. https://doi.org/10.1007/BFb0053060

[4] Hamed Seiied Alavi, Seth Gilbert, and Rachid Guerraoui. 2008. Extensible encoding of type hierarchies. In *POPL*. https://doi.org/10.1145/1328438.1328480

[5] Bowen Alpern, Anthony Cocchi, and David Grove. 2001. Dynamic type checking in Jalapeño. In *JVM*. https://www.usenix.org/legacy/events/jvm01/full_papers/alpern/alpern.pdf

[6] Bowen Alpern, Anonthy Cocchi, and David Grove. 2012. Some new approaches to partial inlining. In *VMIL*. https://doi.org/10.1145/2414740.2414749

[7] David F. Bacon and Peter F. Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *OOPSLA*. https://doi.org/10.1145/236337.236371

[8] Nicolas Badoux, Flavio Toffalini, Jeon Yuseok, and Mathias Payer. 2025. type++: Prohibiting Type Confusion with Inline Type Information. In *NDSS*. https://doi.org/10.14722/ndss.2025.230053

[9] George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *SAS*. https://doi.org/10.1007/978-3-662-53413-7_5

[10] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *NDSS*. https://doi.org/10.14722/ndss.2016.23421

[11] Nathan Burow, Derrick Paul McKee, Scott A. Carr, and Mathias Payer. 2018. CFIXX: Object Type Integrity for C++. In *NDSS*. https://doi.org/10.14722/ndss.2018.23279

[12] Brad Calder and Dirk Grunwald. 1994. Reducing indirect function call overhead in C++ programs. In *POPL*. https://doi.org/10.1145/174675.177973

[13] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. 1986. Interprocedural constant propagation. In *CC*. https://doi.org/10.1145/12276.13327

[14] Jiho Choi, Thomas Shull, and Josep Torrellas. 2019. Reusable inline caching for JavaScript performance. In *PLDI*. https://doi.org/10.1145/3314221.3314587

[15] Norman H. Cohen. 1991. Type-extension type test can be performed in constant time. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 626–629. https://doi.org/10.1145/115372.115297

[16] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *CCS*. https://doi.org/10.1145/2810103.2813682

[17] Jan de Mooij, Matthew Gaudet, Iain Ireland, Nathan Henderson, and J. Nelson Amaral. 2023. CacheIR: The Benefits of a Structured Representation for Inline Caches. In *MPLR*. https://doi.org/10.1145/3617651.3622979

[18] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP*. https://doi.org/10.1007/3-540-49538-X_5

[19] Damian Dechev, Rabi Mahapatra, and Bjarne Stroustrup. 2008. Practical and Verifiable C++ Dynamic Cast for Hard Real-Time Systems. *JCSE* 2 (12 2008), 375–393. https://doi.org/10.5626/JCSE.2008.2.4.375

[20] Damian Dechev, Rabi Mahapatra, Bjarne Stroustrup, and David Wagner. 2008. C++ Dynamic Cast in Autonomous Space Systems. In *ISORC*. https://doi.org/10.1109/ISORC.2008.20

[21] Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M.K. Martin, and Steve Zdancewic. 2013. Ironclad C++: a library-augmented type-safe subset of C++. In *OOPSLA*. https://doi.org/10.1145/2509136.2509550

[22] Alain Deutsch. 1994. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI*. https://doi.org/10.1145/178243.178263

[23] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient implementation of the smalltalk-80 system. In *POPL*. https://doi.org/10.1145/800017.800542

[24] Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *CC*. https://doi.org/10.1145/2892208.2892212

[25] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *PLDI*. https://doi.org/10.1145/3192366.3192388

[26] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *NDSS*. https://doi.org/10.14722/ndss.2017.23287

[27] Roland Ducournau. 2008. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.* 30, 6, Article 33 (Oct. 2008). https://doi.org/10.1145/1391956.1391960

[28] Mary F. Fernández. 1995. Simple and effective link-time optimization of Modula-3 programs. In *PLDI*. https://doi.org/10.1145/207110.207121
[29] Michael Gibbs and Bjarne Stroustrup. 2006. Fast dynamic casting. *Softw. Pract. Exper.* 36, 2 (Feb. 2006), 139–156. https://doi.org/10.1002/spe.686
[30] Joseph (Yossi) Gil and Yoav Zibin. 2005. Efficient subtyping tests with PQ-encoding. *ACM Trans. Program. Lang. Syst.* 27, 5 (Sept. 2005), 819–856. https://doi.org/10.1145/1086642.1086643
[31] Taras Glek and Jan Hubička. 2010. Optimizing real world applications with GCC Link Time Optimization. arXiv:1010.2196
[32] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. 2005. Interprocedural parallelization analysis in SUIF. *ACM Trans. Program. Lang. Syst.* 27, 4 (July 2005), 662–731. https://doi.org/10.1145/1075382.1075385
[33] Mary W. Hall and Ken Kennedy. 1992. Efficient call graph analysis. *ACM Lett. Program. Lang. Syst.* 1, 3 (Sept. 1992), 227–242. https://doi.org/10.1145/151640.151643
[34] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2016. TypeSan: Practical Type Confusion Detection. In *CCS*. https://doi.org/10.1145/2976749.2978405
[35] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.* 21, 4 (July 1999). https://doi.org/10.1145/325478.325519
[36] Urs Hölzle and David Ungar. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI*. https://doi.org/10.1145/178243.178478
[37] Alexander Hück, Jan-Patrick Lehr, Sebastian Kreutzer, Joachim Protze, Christian Terboven, Christian Bischof, and Matthias S. Müller. [n. d.]. Compiler-aided Type Tracking for Correctness Checking of MPI Applications. In *Correctness*. https://doi.org/10.1109/Correctness.2018.00011
[38] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *NDSS*. https://doi.org/10.14722/ndss.2014.23287
[39] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. In *CCS*. https://doi.org/10.1145/3133956.3134062
[40] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX ATC*. https://www.usenix.org/legacy/publications/library/proceedings/usenix02/full_papers/jim/jim.pdf
[41] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: Scalable and incremental LTO. In *CGO*. https://doi.org/10.1109/CGO.2017.7863733
[42] Timothy M. Jones, Sandro Bartolini, Jonas Maebe, and Dominique Chanet. 2011. Link-time optimization for power efficiency in a tagless instruction cache. In *CGO*. https://doi.org/10.1109/CGO.2011.5764672
[43] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *PLDI*. https://doi.org/10.1145/2491956.2462191
[44] Stephen Kell. 2016. Dynamically diagnosing type errors in unsafe code. In *OOPSLA*. https://doi.org/10.1145/2983990.2983998
[45] Ofek Kirzner and Adam Morrison. 2021. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *USENIX Security*. https://www.usenix.org/system/files/sec21-kirzner.pdf
[46] Andreas Krall, Jan Vitek, and R. Nigel Horspool. 1997. Near optimal hierarchical encoding of types. In *ECOOP*. https://doi.org/10.1007/BFb0053377
[47] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *CCS*. https://doi.org/10.1145/2508859.2516713
[48] J. M. Larchevêque. 1992. Interprocedural type propagation for object-oriented languages. In *ESOP*. https://doi.org/10.1007/3-540-55253-7_19
[49] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type Casting Verification: Stopping an Emerging Attack Vector. In *USENIX Security*. https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-lee.pdf
[50] David Xinliang Li, Raksit Ashok, and Robert Hundt. 2010. Lightweight feedback-directed cross-module optimization. In *CGO*. https://doi.org/10.1145/1772954.1772964
[51] LLVM. 2025. UndefinedBehaviorSanitizer, a fast undefined behavior detector. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
[52] Sadie J. Macintyre-Randall. 2023. *Enforcing C++ type integrity with fast dynamic casting, member function protections and an exploration of C++ beneath the surface.* Ph. D. Dissertation. The University of Kent. https://kar.kent.ac.uk/102955/
[53] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/

1065887.1065892
[54] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *CCS*. https://doi.org/10.1145/2660267.2660281
[55] Rohan Padhye and Koushik Sen. 2019. Efficient fail-fast dynamic subtype checking. In *VMIL*. https://doi.org/10.1145/3358504.3361229
[56] Piotr Padlewski, Krzysztof Pszeniczny, and Richard Smith. 2020. Modeling the Invariance of Virtual Pointers in LLVM. arXiv:2003.04228
[57] Krzysztof Palacz and Jan Vitek. 2003. Java Subtype Tests in Real-Time. In *ECOOP*. https://doi.org/10.1007/978-3-540-45070-2_17
[58] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. 2018. Mapping to Bits: Efficiently Detecting Type Confusion Errors. In *ACSAC*. https://doi.org/10.1145/3274694.3274719
[59] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. 2018. Mapping to Bits: Efficiently Detecting Type Confusion Errors. In *ACSAC*. https://doi.org/10.1145/3274694.3274719
[60] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.* 30, 1 (Nov. 2007), 4–es. https://doi.org/10.1145/1290520.1290524
[61] Constantin Cezar Petrescu, Sam Smith, Rafail Giavrimis, and Santanu Kumar Dash. 2023. Do names echo semantics? A large-scale study of identifiers used in C++'s named casts. *Journal of Systems and Software* 202 (2023). https://doi.org/10.1016/j.jss.2023.111693
[62] Patrick W. Sathyanathan, Wenlei He, and Ten H. Tzen. 2017. Incremental whole program optimization and compilation. In *CGO*. https://doi.org/10.1109/CGO.2017.7863742
[63] L. K. Schubert, M. A. Papalaskaris, and J. Taugher. 1983. Determining Type, Part, Color, and Time Relationships. *Computer* 16, 10 (Oct. 1983), 53–60. https://doi.org/10.1109/MC.1983.1654198
[64] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P*. https://doi.org/10.1109/SP.2015.51
[65] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. 2012. Open and efficient type switch for C++. In *OOPSLA*. https://doi.org/10.1145/2384616.2384686
[66] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *SP*. https://doi.org/10.1109/SP.2019.00010
[67] Benoît Sonntag and Dominique Colnet. 2014. Efficient compilation strategy for object-oriented languages under the closed-world assumption. *Softw. Pract. Exper.* 44, 5 (May 2014), 565–592. https://doi.org/10.1002/spe.2174
[68] Amitabh Srivastava and David W. Wall. 1994. Link-time optimization of address calculation on a 64-bit architecture. In *PLDI*. https://doi.org/10.1145/178243.178248
[69] Bjarne Stroustrup. 2020. Thriving in a crowded and changing world: C++ 2006–2020. *Proc. ACM Program. Lang.* 4, HOPL, Article 70 (June 2020). https://doi.org/10.1145/3386320
[70] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security*. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf
[71] Jan Vitek, R. Nigel Horspool, and Andreas Krall. 1997. Efficient type inclusion tests. In *OOPSLA*. https://doi.org/10.1145/263698.263730
[72] Chenyu Wang, Bihuan Chen, Yang Liu, and Hongjun Wu. 2019. Layered Object-Oriented Programming: Advanced VTable Reuse Attacks on Binary-Level Defense. *IEEE Transactions on Information Forensics and Security* 14, 3 (2019), 693–708. https://doi.org/10.1109/TIFS.2018.2855648
[73] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210. https://doi.org/10.1145/103135.103136
[74] Zhefeng Wu, Zhe Sun, Kai Gong, Lingyun Chen, Bin Liao, and Yihua Jin. 2020. Hidden inheritance: an inline caching design for TypeScript performance. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 174 (Nov. 2020). https://doi.org/10.1145/3428242
[75] Yizhuo Zhai, Zhiyun Qian, Chengyu Song, Manu Sridharan, Trent Jaeger, Paul Yu, and Srikanth V. Krishnamurthy. 2024. Don't Waste My Efforts: Pruning Redundant Sanitizer Checks of Developer-Implemented Type Checks. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity24-zhai.pdf
[76] Peng Zhao and J.N. Amaral. 2005. Function outlining and partial inlining. In *SBAC-PAD*. https://doi.org/10.1109/CAHPC.2005.26

Received 2024-11-15; accepted 2025-03-06