ENFORCING C++ TYPE INTEGRITY

WITH FAST DYNAMIC CASTING, MEMBER FUNCTION PROTECTIONS AND AN EXPLORATION OF C++ BENEATH THE SURFACE

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Sadie J. Macintyre-Randall
December 2021

Abstract

The C++ type system provides a programmer with modular class features and inheritance capabilities. Upholding the integrity of all class types, known as type-safety, is paramount in preventing type vulnerabilities and exploitation. However, type confusion vulnerabilities are all too common in C++ programs. The lack of low-level type-awareness creates an environment where advanced exploits, like counterfeit object-orientated programming (COOP), can flourish. Although type confusion and COOP exist in different research fields, they both take advantage of inadequate enforcement of type-safety. Most type confusion defence research has focused on type inclusion testing, with varying degrees of coverage and performance overheads. COOP defences, on the other hand, have predominantly featured control flow integrity (CFI) defence measures, which until very recently, were thought to be sound. We investigate both of these topics and challenge prevailing wisdom, arguing that: 1. optimised dynamic casting is better suited to preventing type confusion and 2. enforcing type integrity may be the only defence against COOP.

Type confusion vulnerabilities are often the result of substituting dynamic casting with an inappropriate static casting method. Dynamic casting is often avoided due to memory consumption and run-time overheads, with some developers turning off run-time type information (RTTI) altogether. However, without RTTI, developers lose not only secure casting but virtual inheritance as well. We argue that improving the performance of dynamic casting can make it a viable

option for preventing type confusion vulnerabilities. In this thesis, we present MemCast, a memoising wrapper for the dynamic cast operator that increases its speed to that of a dynamic dispatch.

A new variant of the COOP exploit (COOPlus) has identified a weakness in almost all modern, C++-semantic-aware CFI defences. The weakness is that they allow derived class functions to be invoked using corrupted base class instances, specifically where an attacker replaces the object's virtual pointer with one from a derived type object. A CFI defence overestimates the set of target functions at a dispatch site to cover all possible control-flow paths of a polymorphic object. Thus COOPlus takes advantage of the lack of type integrity between related types at dispatch sites. In this thesis, we argue that CFI is an unsuitable defence against COOPlus, and type integrity must be applied. Hence we propose a type integrity defence called Member Function Integrity (MFI) that brings type awareness to member functions and prevents any member function from operating on an invalid object type.

To understand the low-level techniques deployed in MemCast and our MFI defence policy, one has to appreciate the memory layout of the objects themselves and the conventions used by member functions that operate on them. However, in our research, we did not find adequate introductory literature specific to modern compilers. For this reason, we supplied our own self-contained introduction to low-level object-orientation.

This thesis has three contributions: a primer on C++ object layouts, an optimised dynamic casting technique that reduces the casting cost to that of a dynamic dispatch, and a new defence policy proposal (MFI) to mitigate all known COOP exploits.

Acknowledgements

Completing a PhD amid a global pandemic has been a challenging task. So I must express my deepest thanks to all who have supported and aided me.

First and foremost, I'd like to thank my wife, Beth, who has unconditionally supported me throughout this PhD, despite the sleep-disturbing late-night sessions and the mountain of paperwork accumulating in the spare room. Thank you for being my biggest support, greatest encouragement, and loudest cheerleader.

I would also like to express my deepest thanks to my supervisor, Andy King, for allowing me the opportunity to independently pursue my own interests, for supporting me in my scientific writing (perhaps the biggest hurdle of this entire PhD), and for his continued effort and commitment to see my PhD through to the end.

Thanks also to my family for your support, belief and encouragement. But most of all, for distilling in me a level of stubbornness and personal drive to always see things through to the end.

Finally, thanks should also go to a truly dedicated friend, Kelda, who took on the gruelling task of proofreading this thesis, despite it being entirely outside her expertise.

Contents

Abstract ii

Acknowledgements iv

Contents v

List of Tables xi

List of Figures xiii

0 Introduction 1

I Behind Object Abstraction 7

1 The Standards Behind the Language 8

1.1 Introduction . . . 9

1.2 The C++ Language - A Brief History . . . 9

1.3 C++ Standard . . . 10

1.3.1 The C++ Standard Specification . . . 10

1.3.2 The Standard Library . . . 11

1.4 Compiler Interoperability . . . 11

1.5 Application Binary Interfaces . . . 12

1.5.1 The Generic ABI . . . 13

1.5.2 The Processor Specific ABI 13
1.5.3 The C++-ABI 20
1.6 Object Layouts in Other Compilers 21
1.7 Concluding Discussion 21

2 Binary Representation of Objects 24
2.1 Introduction 24
2.2 Representing Inheritance 25
2.2.1 Primary Class 26
2.2.2 Single Inheritance 26
2.2.3 Multiple Inheritance 28
2.2.4 Virtual Inheritance 28
2.2.5 Other Class Keywords and Templates 30
2.3 Polymorphism and the Type Systems 32
2.3.1 Types 32
2.3.2 Polymorphic Variables and Object Address-Points 33
2.3.3 Type Checking 35
2.3.4 Casting 36
2.3.5 Member Functions 38
2.4 Dynamic Objects and Supporting Data 41
2.4.1 Object Layouts with Virtual Pointers 41
2.4.2 Virtual Table Layout 44
2.4.3 Virtual Member Functions 50
2.4.4 Run-Time Type Information 52
2.5 MSVC Object Comparison 58
2.6 Concluding Discussion 61

3 Assembly-Level Object Operations 62
3.1 Introduction 62

3.2 Member Functions 63
3.2.1 Function Bodies 63
3.2.2 Non-Virtual Member Functions Calls 63
3.2.3 Virtual Member Function Calls 64
3.3 Object Construction 66
3.3.1 Primary Class Construction 66
3.3.2 Derived Class Construction 67
3.3.3 Virtual Inheritance Class Construction 68
3.4 Cast Operations 70
3.4.1 C-style Cast 71
3.4.2 `static_cast<target>(variable)` 72
3.4.3 `reinterpret_cast<target>(variable)` 74
3.4.4 `dynamic_cast<target>(variable)` 74
3.4.5 Compiler Casting Optimisations 79
3.4.6 Custom RTTI Solutions 82
3.5 Concluding Discussion 83

II Object Vulnerability and Exploitation 84

4 Type Confusion Vulnerabilities 85

4.1 Introduction 85
4.2 Type Confusion Defense Strategies 88
4.3 Type Inclusion Testing 92
4.4 What about Dynamic Casting? 94
4.5 Concluding Discussion 100

5 Memoised Casting 101

5.1 Introduction 101
5.2 Cast Stability and Deal.II Analysis 105

5.2.1 Cast Stability . . . 106
5.2.2 Deal.II Experiments . . . 107
5.2.3 Deal.II Results . . . 109
5.3 Design and Implementation . . . 111
5.3.1 MemCache Objects . . . 112
5.3.2 Dynamic Cast Wrapper . . . 116
5.4 Experimental Results . . . 124
5.4.1 The True Cost of Casting . . . 124
5.4.2 Evaluation of MemCast's Capabilities . . . 130
5.4.3 Deal.II Revised . . . 139
5.4.4 OMNet++ . . . 143
5.4.5 Antlr4 . . . 146
5.5 Related Work . . . 150
5.6 Future Work . . . 152
5.7 Concluding Discussion . . . 155

6 Object-Oriented Code-Reuse 158

6.1 Introduction . . . 158
6.2 Counterfeit Object-Orientated Programming . . . 164
6.2.1 The COOP Exploit . . . 164
6.2.2 COOP Defenses . . . 167
6.2.3 COOPlus . . . 169
6.3 CFIXX Under the Microscope . . . 172
6.3.1 Object Type-Integrity . . . 172
6.3.2 CFIXX Implementation . . . 172
6.3.3 CFIXX Vulnerabilities . . . 174
6.4 Member Function Integrity . . . 182
6.4.1 Defence Policy . . . 183
6.4.2 Implementation Proposal . . . 184

6.4.3 Converting Bitype's Encoding Scheme 184
6.4.4 Benefits of MFI 194
6.4.5 Scalability 197
6.5 MFI Proof of Concept 201
6.5.1 Example Program Design and Vulnerability 201
6.5.2 Exploit 1: COOPlus vptr Overwrite 204
6.5.3 Exploit 2: Adjacent Vtable Access 206
6.5.4 Exploit 3: Unprotected Library 207
6.5.5 MFI - Source-Base Implementation 210
6.5.6 Defence Comparison 213
6.6 Future Work 216
6.7 Concluding Discussion 219

III Reflection 221

7 Concluding Discussion and Future Work 222
7.1 Conclusion 222
7.1.1 Low-level C++ Implementation 222
7.1.2 MemCast 223
7.1.3 MFI 224
7.2 Future Work 225
7.2.1 MemCast 225
7.2.2 MFI 226

Bibliography 229

A Behind Object Abstraction 249
A.1 RTTI Hierarchy 249
A.2 Full Virtual Inheritance Constructor Call 250

B Deall.II Full Results **251**

B.1 Dynamic Down-Cast Locations 251

B.2 Stability of Each Dynamic Down-Cast Site 254

C OMNet++ Full Results **259**

C.1 Dynamic Down-Cast Locations 259

C.2 Stability of Each Dynamic Down-Cast Site 261

D Antlr4 Full Results **264**

D.1 Dynamic Down-Cast Locations 264

D.2 Stability of Each Dynamic Down-Cast Site 266

List of Tables

1 Data Sizes and Alignment Requirements for AMD64 Platforms . . 14
2 Calling Convention Requirements for AMD64 Platforms 17
3 Vtable Entries Explained 44
4 RTTI Classes and Attribute Uses 54
5 Exploit Mitigation vs Sanitisers 89
6 Stability Calculation Example 107
7 Cast Sites Stability 109
8 Deal.II Cast Site Stability Results 110
9 Type Specifications of the Dynamic Cast Operator 116
10 Virtual Pointer Flag Meanings 120
11 Cost of Casting Results 127
12 Minimum Stability and Executions for MemCast 138
13 Deal.II with MemCast 142
14 OMNet++ with MemCast 144
15 Antlr4 with MemCast 148
16 Vtable Interleaving Example Data 169
17 Adapted Bitype Encoding Scheme for MFI 186
18 MFI Type Inclusion Test Example 1 192
19 MFI Type Inclusion Test Example 2 194
20 MFI Type Inclusion Test Example 1 200

21 MFI, CFIXX, and Clang CFI comparison against three different exploits 214
22 Deal.II Dynamic Down-Casts Locations and Visits 251
23 Stability Results for Deal.II Step-x Programs 254
24 OMNet++ Dynamic Down-Casts 260
25 Stability Results for OMNet++ Programs 261
26 Antlr4 Dynamic Down-Casts 265
27 Stability Results for Antlr4 Programs 266

List of Figures

1 The Effects of Padding in Data Structures 16
2 Instruction-level Examples of Calling Conventions 18
3 Different Compiler Layouts 22
4 Primary Class Object 26
5 Single Inheritance Objects 27
6 Multiple Inheritance 28
7 Diamond Problem 29
8 Virtual Inheritance 29
9 Template Expansion 31
10 Class Template Object 32
11 Assigning Objects to Variables 34
12 Object Address-Points 34
13 Sub-object Address-Points 35
14 Access Rights to Functions 40
15 Dynamic Object Layouts and Source Code 42
16 Dynamic Object Layouts 43
17 Vtable Layout 44
18 Vtable Layout Single Inheritance 45
19 Vtable Layout Multiple Inheritance 47
20 Vtable Layout Virtual Inheritance 1 48
21 Vtable Layout Virtual Inheritance 2 49

22 Access Rights to Virtual Functions 51
23 RTTI Object Layout 53
24 RTTI Data Member 55
25 RTTI Objects and Relationships 57
26 MSVC Object Layout 59
27 MSVC vs g++ 60
28 Member Function Call 63
29 The Dynamic Dispatch Mechanism 65
30 Basic Constructor Call 66
31 Nested Constructor Call 68
32 Virtual Inheritance Constructor Call 69
33 Cast Example 70
34 C-style Casting 71
35 Static Cast Operation 72
36 Static Cast Type Confusion & Virtual Base Casting 73
37 Reinterpret Casting 74
38 Dynamic Cast Operation 75
39 The Global Dynamic Cast Functions 76
40 Types of Casting 81
41 Type confusion listing and depiction 87
42 CVE Type Confusion Vulnerabilities 88
43 Fast Dynamic Cast Example 97
44 Dynamic Cast Hot Path with Depiction 103
45 Virtual pointers vs RTTI pointers 114
46 MemCache Objects 116
47 MemCast_Resolver 118
48 Virtual Pointer Bit Flags 120
49 MemCast Function for Pointer Targets 122

50 MemCast's default Dynamic Cast Wrapper 123
51 MemCast Function for lvalue References Targets 123
52 Hierarchy used for Cast Testing 126
53 Stability and The Estimate Cost of MemCasting 133
54 Minimum MemCast Visits 135
55 Run-time Savings With MemCast 136
57 COOP Example 1 165
58 COOP Example 2 166
59 CFI Targets 167
60 Interleaved Vtable Layout in Clang CFI 168
61 COOPPlus Exploit 170
62 A COOPPlus Attack 171
63 CFIXX MetaData Table 173
64 CFIXX MDT Over Population 175
65 Over Population MDT and Ghost Objects 176
66 Forced Type Confusion Under CFIXX 177
67 Dynamic Casting used in a COOP-Style Dispatch 179
68 Adjacent Vtable Calls 181
69 Mapping Techniques in COOP Defences 183
70 Bitype Example Hierarchy 185
71 Adapted Bitype Object Tracking for MFI 188
72 Object Tracking for MFI 190
73 MFI Example 194
74 MFI Proction for Partial Coverage 195
75 Scalable MFI Encoding Scheme 198
76 MFI Proof of Concept 1 202
77 Example Main Menus 203
78 MFI Proof of Concept 2 203

79 Exploit 1 COOPlus 205

80 Exploit 2 Adjacent Vtable 206

81 Unprotected Dispatch in Unprotected Library 208

82 Exploit 3 Unprotected Library 209

83 MFI Code Implementation 211

84 MFI Safe Set Implementation 212

85 RTTI Inheritance Hierarchy 249

86 Virtual Inheritance Constructor Call 250

Chapter 0

Introduction

C++ is an object-oriented (OO) language that provides multiple orthogonal layers of abstraction. The rationale behind abstraction is to reduce the cognitive load on the programmer by freeing them from consideration of the low-level machine. The most fundamental of abstraction is the concept of class. This construct allows a programmer to define their own data types (realised as objects) and specify functions that operate on them. This coupling of data and functions enables a programmer to encapsulate different program components into distinct modules, known as types, and then use these types to organise the program into hierarchies. Enforcing segregation of these modules, that is, only allowing an object to be accessed by its member functions, is known as type-safety and is critical to securing OO programs from fault or exploitation. With this emphasis on abstraction, it is unsurprising that most developers do not give low-level implementation a second thought. However, to guard against common C++-specific bugs and exploits, low-level understanding is vital. Despite this, the seminal text [74] on low-level object implementation is over 25 years old and focuses on the (now discontinued) Cfront compiler. Online blogs [2, 19, 48, 97, 108, 109, 115, 121] offering commentary on modern C++ compilers are, at best, partial. In this thesis, we dive beneath all layers of abstraction and provide an in-depth explanation of how objects are

realised in memory, specifically for the GNU C++ compiler. We also disassemble well-known operations, like casts, constructors, and member function calls and explain how these operations interact with objects at the assembly level.

A compiler will enforce type-safety by identifying any illegal type operations and reporting them to the programmer for correction. However, due to inheritance and polymorphism, a compiler's ability to sanitise code for type vulnerabilities is limited, as a single textual object reference can refer to different object types at run-time. After compilation, all high-level abstractions are lost, objects become nothing more than a collection of bits in an allocated memory area, and functions can indiscriminately execute on data passed to them. Conceptually, however, these memory regions and functions still have defined types, but enforcing type-safety at this level becomes more difficult. In fact, the lack of type-awareness at the binary level provides an environment for some security vulnerabilities and exploitation techniques unique to OO. This thesis examines one of each, a security vulnerability called type confusion and an exploitation technique known as Counterfeit Object-Orientated Programming [112].

Type confusion is a common and dangerous vulnerability found in many C++ programs. It occurs when a program incorrectly interprets an object's type at run-time and then accesses data and functions that would have been previously inaccessible. Access to illegal data and functions can lead to memory corruption and control-flow hijacking attacks when abused by an attacker. One way to avoid this is to use the `dynamic_cast` operator to securely and correctly cast an object before interaction. However, this technique is deemed (for some) to have unacceptable performance overheads and is often omitted (like in Google Chrome [22]). There has been plenty of research into defensive strategies to combat type confusion vulnerabilities, almost all of which design a new method of testing type correctness, but very few seek to optimise dynamic casting itself. In this thesis, we introduce MemCast, a memoising wrapper function for the dynamic cast operator,

which reduces the cost of dynamic casting to that of dynamic dispatch.

Counterfeit object-orientated programming or COOP [112], is an exploitation technique that injects counterfeit objects (designed by an attacker) into memory and uses these objects to call illegal functions that interact with the attacker's counterfeit data. This is known as a control-flow hijack attack, as the attacker's goal is to take control of a program's control-flow by invoking a chain of functions that the attacker has carefully selected. The most prominent defensive technique against COOP is control-flow integrity (CFI), a defence policy that guarantees the control-flow of a program falls within a set of predetermined control-flow paths. However, a recent variant of the COOP exploit (called COOPPlus [21]) has bypassed almost all modern CFI defences. In this thesis, we break away from these popular CFI defences and introduce a novel defence policy called Member Function Integrity (MFI). MFI is a type-integrity defence that brings type-awareness to all member functions, meaning that functions can verify the objects they receive are valid types. We present a thorough and detailed proposal of how MFI could be implemented and demonstrate its capabilities against COOP and COOPPlus with a proof of concept.

Thesis Structure We have taken an unusual approach to structuring this thesis and have separated it into parts. The first two parts present the main body of the work, each with its own goals and contributions, and the final part draws these themes together. A common theme throughout is object types, and despite the segregation of parts, each chapter builds on its predecessor (if there is one).

Part 1 Behind Object Abstraction is a deep dive into the low-level implementation of C++ objects. It is both an introduction to the topic and a contribution. Descriptions of modern-day object layouts are scattered across white papers, scientific papers, books, forums, blogs, and even patents, and most do not adequately explain variations between compilers and machines. As we could not

find adequate introductory literature, we provided it instead.

Chapter 1 discusses compilers, their implementation of formal standards and platform specifications, and how they impact object layout and member function interactions.

Chapter 2 discusses the object representation of inheritance, how polymorphism is represented and implemented, and explains the role and layout of the supporting auxiliary data structures such as virtual tables and run-time type information.

Chapter 3 explains how various operations are realised, namely member functions, constructor functions, and cast operations.

Part 1 provides a body of knowledge helpful in accessing Part 2.

Part 2 Object Vulnerability and Exploitation explains the impact of type confusion vulnerabilities and reviews the defences and mitigation strategies developed to protect against them. Dynamic casting is an inbuilt mechanism that can prevent type confusion at cast sites, but it is often omitted from a program because it is considered prohibitively slow. Despite its reliability, few have attempted to counter this problem by optimising its performance. So we did just that! We designed a memoising cast that optimises for speed. The technique is realised as a wrapper function called MemCast, which can reduce the cost of a dynamic cast to that of dynamic dispatch. This drastic reduction in performance cost will hopefully break the stigma of expensive dynamic casting and compel more developers to use it, albeit in the form of MemCasting.

Beyond the discussion of type confusion vulnerabilities, we also discuss COOP and a new variant COOPlus that threatens all CFI defences posed thus far. In light of this, we propose a new security defence, MFI, to prevent such exploits.

Chapter 4 provides an overview of type confusion mitigations, sanitisers, and the wide use of type inclusion testing as a solution.

Chapter 5 provides a detailed explanation of the design of MemCast, an evaluation of its performance against dynamic casting, and an assessment of its overall performance benefits when applied to several real-world libraries.

Chapter 6 introduces the Member Function Integrity (MFI) policy, provides a detailed implementation proposal, and demonstrates its capabilities against COOP-style exploits in a proof of concept.

Part 3 Reflection contains a single chapter finalising this thesis with a concluding discussion. The chapter discusses opportunities for future work and reflects on the main contributions of the thesis.

Supporting Figures The topics discussed in the thesis bridge the gap between high-level code and low-level memory, which can be particularly difficult to articulate with discussion alone. For this reason, almost all low-level discussions are supplemented with visual depictions and examples. Many of these supplementary figures took longer to design and create than the text itself. We let the reader draw their own conclusion as to whether this investment was worthwhile.

Contributions This thesis makes the following contributions:

- We present a modern and thorough insight into low-level object-oriented implementation without abstraction. This includes: how objects, virtual tables, and run-time type information are realised in memory; how and why different compilers and platforms can apply different variations of these structures; and how common operations, such as member functions, casts, and constructors, interact with these data-structures at the assembly level.
- We introduce a novel optimisation technique for dynamic casting called MemCast. We demonstrate that MemCast can outperform dynamic casting, reducing its cost to that of dynamic dispatch, and show that replacing dynamic casting with MemCast can improve the overall performance of large

programs.

- We present a new and novel defence policy, Member Function Integrity (MFI), that brings type-awareness to member functions. We give an implementation proposal with this policy, detailing the mechanisms and type testing methods required for its deployment. From this proposal, we demonstrate how MFI can mitigate all known COOP and COOPLUS attacks and back up this claim with a proof of concept.

Part I

Behind Object Abstraction

Chapter 1

The Standards Behind the Language

In object-oriented (OO) programming, one might ask, what does an object actually look like in memory? How are the positions of each data member determined? Furthermore, what auxiliary data members support run-time operations? Part 1 answers these questions while providing a body of knowledge that supports the rest of the thesis. In this chapter, we answer the question, why? Specifically, why do different compilers (on different platforms) produce different object layouts and binary instructions, even when compiling identical C++ source code? The answer depends on many factors, many of which sit within the broad topics of language specification and compilation. The pertinent go-to documents are language specifications and technical interface protocols, nearly all of which are large and dry. To maintain a rapid pace of delivery, we focus on object layout and function calling conventions, as this serves as a filter for distilling the essence from these documents. The narrative given in this chapter presents only the essential (and, dare we say, intriguing) information and provides insight into how C++ is realised at the lowest level.

1.1 Introduction

Why? Why do object layouts differ for different C++ compilers? For different machines? For different operating systems (OS)? Unfortunately, these questions have no simple answers because each machine and OS determine how the compiler, linker, libraries and the run-time system function as a whole. To operate as a consistent whole, the processor, OS, and language itself, must agree on how data is realised and organised in memory but also exchanged between components. Compiler vendors are then responsible for bringing these agreements together to produce binary code compatible with the underlying platform and library modules it may be linked against. If these vendors want to produce code compatible with others, then further agreements must be made between them. Each agreement has a particular impact on object layout. Exactly which combination of agreements is brought together by a compiler will determine the final object layout used within a binary.

1.2 The C++ Language - A Brief History

The first known language to implement OO constructs was Simula 67 [26], which later inspired Bjarne Stroustrup [126] in his initial design of the C++ language, back in 1979 [127]. The original version of C++, known as "C with Classes" [127], added Simula-like facilities to the C language [126]. C with Classes, first publicised in 1982 [124, 125], required a preprocessor (called Cpre [127]) to translate OO code into C code before passing the source to a C compiler.

By 1983, C with Classes had been renamed to C++ [127] and developed into its own separate language with it own standalone compiler. The first C++ compiler front-end, developed by Stroustrup himself, called Cfront, was released in 1985 [6]; but standardisation was not achieved until 1990, by which point, multiple vendors had developed their own C++ compilers [127]. Each vendor had their

own interpretation of the language, often using Cfront as a defacto standard. A standardisation came with the publication of the annotated C++ reference manual [36]. The International Organization for Standardization (ISO) later used this manual as the basis for the C++ ISO standard, which was published in 1998 [122]. Since then, this standard has received six major revisions to date and a seventh currently under development. C++ is now over 40 years old and one of the most popular languages worldwide.

1.3 C++ Standard

The ISO C++ Standard [57], specifies the semantics of the C++ language with the use of code snippets and technical commentary. It has two parts: a specification for all C++ facilities and functionalities, explaining how the language should function; and the C++ Application Programming Interface (API), known as the Standard Library.

1.3.1 The C++ Standard Specification

The C++ standard specification provides all formal definitions of the semantics and facilities within the C++ language. In terms of objects, it describes how they exist in memory; how they are created, destroyed, referenced, accessed, and manipulated. It defines terms such as sub-objects, complete objects, and most-derived objects. It also defines the rules and properties for C++ classes, including such facilities as types, class members, inheritance, and polymorphism. We discuss all of these concepts in Chapter 2, but what is important to note is that the standard does not dictate how they are realised at the binary-level; the compiler implementation and platform specifications govern this.

1.3.2 The Standard Library

The Standard Library is a collection of generic classes, functions, and facilities (including all keywords such as `new`, `delete`, `dynamic_cast`) used to support the C++ language and developers with their everyday programming tasks. Within this API are highly technical expectations of the functionality of each component, but again, no details of low-level implementations. Low-level implementation is not included in the Standard Library because many facilities require interaction with an OS's system calls. As each OS provides its own system calls, no universal standardisation can be applied.

The C++ standard library has many different implementations by many different developers. Some implementations are provided with a given platform; others are downloaded as an additional component to a compiler. In either case, to a programmer, all standard library implementations should be functionally homogeneous but may differ at the instruction level.

1.4 Compiler Interoperability

Before the standardisation of C++, compiler vendors relied on their own interpretations of the C++ language to produce executables. With different interpretations came different implementations, conventions, and, most notably, different object data layout structures. This variety made many executables, compiled with different compiler vendors, utterly incompatible with one another. Compatibility (better known as interoperability) of binaries, generated from multiple compilers, is crucial to today's software development and the fundamental premise behind library packages. To achieve full interoperability, compiler vendors must conform to a family of strict binary interfaces known as application binary interfaces (ABIs). An ABI family (expanded on in Section 1.5) specifies the binary expectations for the OS, for the architecture, and the realisation of language semantics.

If a compiler vendor conforms to an ABI family, its binaries will be fully interoperable with any other conforming compilers' binaries. One of the results of this interoperability is that their methods of realising data, including object layouts, will be identical. However, aspects unspecified by the ABI (i.e. not required for communication) are left to the vendor's interpretation, which again leads to instruction-level variations, despite ABI conformance. Compiler vendors with different ABI conformance or no formal ABI specification will likely have different conventions and data layout structures.

1.5 Application Binary Interfaces

In the most general sense, an ABI is a boundary in which two or more separate binary components exchange information. How this exchange is achieved is platform-specific. Some platforms publish formal documented ABIs, allowing multiple compilers to adopt and conform to that exact specification. Other platforms deploy their own compilers to act as the standard for that platform rather than publishing their own ABIs. An ABI specifies how a program should interact with the OS, the processor, or libraries and separate source files. For many platforms, each of these three aspects is defined in its own separate (documented) ABI; collectively, they come together to define a family of ABIs, to which a program must conform as a whole. Compilers enforce conformance of a compiled program against the ABI family, which in turn ensures compatibility with the underlying platform and code it is linked against. A given platform is composed of the OS and architecture (the processor). A platform-specific ABI is, therefore, a composition of two ABI specifications: a generic ABI (gABI), specific to the OS, and a processor-specific ABI (psABI) supplement, specific to the processor and the OS. Although these two parts are orthogonal, the psABI is typically presented as sub-chapters of different chapters within the gABI document. Together, these

documents specify the C components of C++. As C++, by its very design, is an extension to the C language, the psABI also requires an extension in the form of a C++-specific ABI. The C++-specific ABI ensures compatibility against the OO constructs of C++ code.

Many Unix-based systems, such as Linux, BSD and Mac OS X conform, at least in part, to the System V ABI. The System V ABI consists of the gABI [7] and supporting psABI, which again is processor-specific. There is a psABI for every popular micro-architecture family: the Itanium psABI [54], the AMD64 psABI [79], the PowerPC psABI [129] and the ARM psABI [4]. On the other hand, the Microsoft Windows OS does not have a single unified ABI publication, as Microsoft opts to deploy their own compiler (MSVC) to act as the standard for Windows OS platforms. Some aspects of the Microsoft Windows ABI requirements are published on their online library system, Microsoft Docs [86], which continues to grow. However, without complete transparency, other compiler vendors (outside of Microsoft) may be forced to reverse-engineer ABI information to achieve interoperability, as was the case (historically) for calling conventions [42].

1.5.1 The Generic ABI

The gABI defines the binary requirements of a compiled program to be compatible with the target OS. These requirements include file formats and file headers, which provide the prerequisite information for program execution, linking and relocation. The gABI also describes the linking mechanisms, alongside a list of libraries that any conforming system must support. Of all the ABIs, the gABI has the most negligible impact on OO facilities, conventions, and object layout.

1.5.2 The Processor Specific ABI

A psABI is a supplementary set of requirements used in conjunction with a gABI. The psABI plays a significant role in object layout, as it governs both size and

Type	Data Type	Microsoft Windows		System V	
		Size	Alignment	Size	Alignment
Integral	bool	1	1	1	1
	char (signed & unsigned)	1	1	1	1
	short (signed & unsigned)	2	2	2	2
	int (signed & unsigned)	4	4	4	4
	long (signed & unsigned)	4	4	8	8
	long long (signed & unsigned)	8	8	8	8
Pointer	any-type *	8	8	8	8
Floating-Point	float	4	4	4	4
	double	8	8	8	8
	long double	8	8	16	16

Table 1: A comparison of data type sizes and alignment requirements for the AMD64 architecture hosting different OS.

alignment requirements of all primitive data types. The size determines the number of bytes a primitive data type can occupy, whereas the alignment determines the possible positions in memory that data can take. As well as this, the psABI defines the function calling conventions (or function calling sequence) for that processor. Standardised function calling conventions ensure interoperability between two code modules, as it standardises the way parameters are passed between function calls. Many System V supplementary psABIs are available online[1], but for the remainder of this chapter, we will focus on the AMD64 psABI.

1.5.2.1 Data Size

A psABI standardises all primitive data types, such as ints, booleans, pointers, etc., for a given platform. Table 1 compares data type sizes specified in the Microsoft Windows AMD64 psABI [81] verses the System V (UNIX based OS) AMD64 psABI [79].

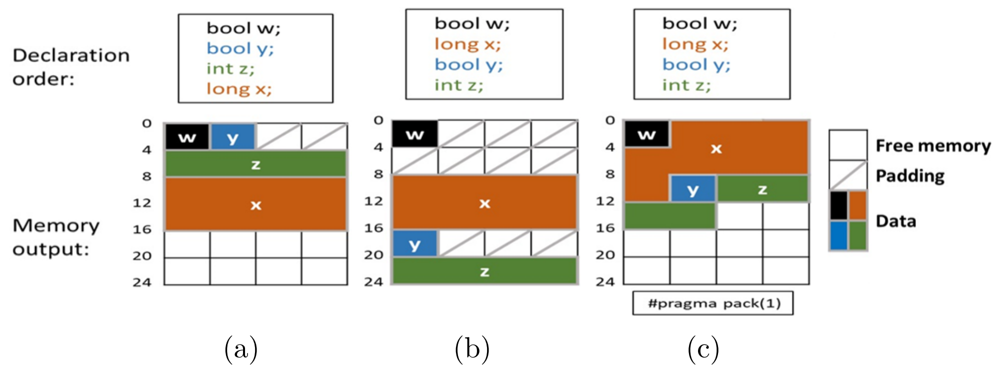
[1] https://www.uclibc.org/docs/

1.5.2.2 Data Alignment

Data is transferred in fixed-sized blocks between caches at consecutive levels in the memory hierarchy. Transfers between different consecutive levels employ different fixed-sized blocks. These blocks are indivisible units of data transfer, so when a datum straddles two blocks at one or more levels of the hierarchy, more memory accesses are required. This motivates aligning data items to block boundaries and is fulfilled using data alignment values.

Alignment requirements dictate that the address of any data type must be divisible by its alignment value. The alignment value is specified within the psABI, typically 1, 2, 4 or 8 bytes, as illustrated in Table 1. Objects will often contain padding to ensure their data members are correctly aligned. Padding is simply memory deliberately left empty or unassigned between consecutive data members. Objects themselves also have an alignment value equal to the greatest alignment value of all the data members. An object's size must be divisible by its alignment value, which is often achieved by adding more padding. This requirement ensures that the object and all its data members are always correctly aligned, even when objects are contiguously allocated in a data structure, such as an array.

Figure 1 reflects the effects of padding and data member ordering within an object. The data member order within an object is not ad hoc but reflects the order of declaration within the source code. Figure 1 compares the declaration order of four data variables: two booleans of size and alignment (1), an integer (4) and a long (8) while demonstrating the effect of alignment on memory consumption. Figure 1a illustrates the smallest possible object constructed from these variables whilst adhering to the alignment requirements. Figure 1b, on the other hand, demonstrates a possible increase in memory consumption, which can occur accidentally when a programmer neglects consideration of alignment values and declaration order. Figure 1c, depicts the result of an alternative technique used to reduce memory consumption by using a `#pragma pack(n)` in-code instruction.



(a) (b) (c)

Figure 1: A visual representation of padding and packing techniques. Each square represents one byte of data.

The `#pragma pack(n)` instruction is used to "pack" data together by aligning them at `n` bytes (or lower) intervals.

Pragma instructions are not part of any language standard but instead are macro operations supplied by some (not all) compiler vendors. Pragmas allow developers more flexibility in how their code is compiled as they override the default setting of the compiler. The `#pragma pack(n)` instruction, for example, overrides the default alignment requirements, and although this can optimise for memory space, it also breaks ABI conformance. Programs that break ABI conformance are not compatible with those that conform, which could be problematic when linking to dynamic libraries.

The `#pragma pack(n)` instruction is not the only technique available for overriding alignment requirements; there are several packing techniques, all compiler dependant, that could affect object layout; these include in-line code instructions, whole file instructions, and compiler optimisation flags. Although packing often improves memory space, it can also incur a performance hit due to increased cache misses. The hardware designed to aid cache memory is most efficient when data is correctly aligned. When a datum is unaligned and straddles two memory blocks of size 2^n (like with `long x` in Figure 1c), it may straddle across two cache lines as well, making unaligned memory reads more expensive than aligned ones.

<table>
<tr><th>Usage</th><th>Microsoft Windows</th><th>System V</th></tr>
<tr><td>1st argument to functions</td><td>RCX</td><td>RDI</td></tr>
<tr><td>2nd argument to functions</td><td>RDX</td><td>RSI</td></tr>
<tr><td>3nd argument to functions</td><td>R8</td><td>RDX</td></tr>
<tr><td>4th argument to functions</td><td>R9</td><td>RCX</td></tr>
<tr><td>5th argument to functions</td><td rowspan="3">passed via the stack</td><td>R8</td></tr>
<tr><td>6th argument to functions</td><td>R9</td></tr>
<tr><td>$\geq 7^{th}$ argument to functions</td><td>passed via the stack</td></tr>
<tr><td>return value register</td><td>RAX</td><td>RAX</td></tr>
</table>

Table 2: A comparison of calling conventions for the AMD64 architecture hosting different OS.

Programmers rightly tread cautiously before employing such techniques.

1.5.2.3 Function Calling Convention

A function calling convention is a standardised procedure for calling functions. At the lowest level, the psABI will describe the conventions for passing data parameters to and from subroutines. Data parameters are passed either via the stack or via registers. In the case of registers, each register has an assigned role and calling sequence defined within the psABI. Which registers are used, the order they are used in, or whether they are used at all depends upon the parameter list of that subroutine.

Registers in the AMD64 machine have different purposes, which we will generalise here. General-purpose registers are used to calculate data and store addresses, control registers monitor programs and store status flags, and AVX (advanced vector extension) registers are used for vector and floating-point data. The intricate usages and register roles can be explored outside of this thesis [67], but for now, our focus is how registers are used within the context of OO function calls. To ensure examples are easy to follow, we will use only integer and pointer parameters, which require general-purpose registers only.

Table 2 presents some of the general-purpose registers and their usage as part of a function calling sequence, as defined within the AMD64 psABI for Microsoft

Simple Addition

```
int add(int a, int b, int c){
  return a+b+c;
}
int x = add(1,2,3);
```

MSVC Assembly on Windows OS

```
1 mov r8d, 3
2 mov edx, 2
3 mov ecx, 1
4 call int add(int,int,int)
5 mov DWORD PTR x$[rsp], eax
6
```

g++ Assembly on Windws OS

```
mov r8d, 3
mov edx, 2
mov ecx, 1
call add(int, int, int)
mov DWORD PTR [rbp-4], eax
```

g++ Assembly on System V OS

```
1 mov edx, 3
2 mov esi, 2
3 mov edi, 1
4 call add(int, int, int)
5 mov DWORD PTR [rbp-4], eax
6
7
```

Clang Assembly on System V OS

```
mov edi, 1
mov esi, 2
mov edx, 3
mov dword ptr [rbp - 4], 0
call add(int, int, int)
xor edx, edx
mov dword ptr [rbp - 8], eax
```

Figure 2: Instruction-level example of calling convention for the AMD64 architecture hosting different OS.

Windows [81] and System V platforms [79]. These general-purpose registers are used to pass integer and pointer values to or from a function call[2]. Some general-purpose registers have important roles to play in program execution, most notably: `RSP`, the stack pointer; `RBP`, often used as a frame pointer (points to the base of a stack frame); and `RIP`, the instruction pointer. Other general-purpose registers may be used as temporary registers or callee-preserved registers. `RBX` is also a notable register, as it will appear in many later examples; this register is often used as a callee-preserve register, which means it is assigned data prior to a function call and is guaranteed to store that same data upon the return of that function.

Conformance with a platform's calling conventions enables abiding compilers to produce interoperable subroutines. These subroutines are interoperable because their parameters are passed in identical ways via the stack and registers. Figure 2 presents some examples of the calling conventions produced by different compilers on both a Windows OS and System V OS. Firstly, notice that the use of registers follows the conventions outlined in Table 2 for each OS. Secondly, notice that the GNU g++ compiler produces different calling conventions based on the platform

[2]Other data types, like floats and doubles, are assigned to specialist register sets, all of which are outlined in the psABI

it is compiling for; the GNU g++ compiler is a cross-platform compiler, meaning it can adapt its psABI implementation based on its target platform. Thirdly, notice that the LLVM Clang compiler (which adheres to the same family of ABIs as g++ on the System V OS) produces the same calling conventions as g++, albeit slightly differently. This is an excellent example of two compilers that produce interoperable code but still differ at the instruction level.

We note that the functions presented in Figure 2 examine one of the most straightforward cases; three `int` parameters. Things can get more complex when different primitive data types are passed or if full structs are passed by value. In the case of passing structs by value, Microsoft will always pass them via the stack, whereas for System V (using the g++ compiler), the size of the struct matters. Large structs will be passed entirely via the stack, whereas small structs (one that contains two `int`s, for example) will be passed via registers. Additional optimisations are made when passing structs via registers; for example, on a 64-bit System V machine, our struct of two `int`s is passed via one register (`RDI`). This is possible because the `int`s in the struct are aligned and are four bytes each; therefore can be efficiently passed via one eight-byte register.

We further note that the Microsoft Windows ABI calling conventions listed in Table 2 are the default convention for the AMD64 system. This convention is known as the `__fastcall` calling convention and will always try to pass the first four parameters via registers [81]. For x86 (a 32-bit system), the Microsoft Windows ABI has several different calling conventions, each managing function parameters and the stack in various ways [82]. These conventions all come with a keyword (`__cdecl`, `__clrcall`, `__stdcall`, `__fastcall`, `__thiscall` and `__vectorcall`), enabling the programmer (if they wish) to override the default convention. These keywords are compatible with other versions of the Microsoft compiler, but if they are not supported by that version, the default convention

will override the requested convention. For now, we will focus only on the default conventions for the AMD64 system and, for simplicity, stick with examples that pass only integer and pointer values as parameters. To experiment with other function parameters and their calling conventions on various compilers, we recommend using Compiler Explorer [46] for easy code-to-assembly comparisons.

1.5.3 The C++-ABI

A C++-ABI specification details many of the low-level implementations of the OO constructs outlined in the C++ standard. It is targeted at compiler designers, presenting conventions that stipulate the transformation of a high-level C++ program to a binary executable. The generated executable satisfies both the functionalities defined in the C++ standard and the low-level requirements of the platform (gABI and psABI). If a compiler follows a formally documented C++-ABI, then this document provides the low-level implementation of data member ordering within an object, but not their data sizes or alignments, which falls to the psABI. The document will also provide details on supporting run-time mechanisms for OO constructs such as polymorphism and dynamic dispatch, which require additional auxiliary data structure and object data members to determine object types at run-time. These auxiliary data structures, which includes virtual tables and run-time type information, also have a strict layout requirement defined in the C++-ABI.

The Itanium C++-ABI [23] is the most widely used C++-specific ABI; initially designed for the Itanium architecture, it is now a non-processor specific specification for the System V OS. The data structures and conventions defined in this specification align themselves with the requirements of the ISO C++ standard and the System V gABI. Where processor specifics are required, the target architecture's psABI acts as a supplement document, making the Itanium C++-ABI applicable to any System V platform.

For Microsoft Windows platforms, there is no single unified ABI publication, but parts of Microsoft's C++-ABI can be found in their online documentation [86] alongside platform-specific specifications. Without a complete ABI publication, other compiler vendors (who wish to produce for a Windows platform) try to produce code compatible with the MSVC compiler instead, making the MSVC compiler, at least in part, a de facto standard for the Windows platform. Many aspects of the MSVC ABI is purposely unpublished, as Microsoft wish to leave latitude for changes [84] and even have a history of breaking binary compatibility between major updates of their compiler [83].

1.6 Object Layouts in Other Compilers

Figure 3 displays the different object layouts that can occur when using different compilers. Although the contents of these objects might not be clear yet, we invite the reader to return to this section after completing Part 1 of this thesis, once the different components of objects have been fully explained. For now, these five objects demonstrate the variety of object layouts, strategies for representing inheritance relationships, and the auxiliary data members (vptrs and vbptrs), that can occur when compiling identical source code with different compilers.

1.7 Concluding Discussion

The object layouts and calling conventions realised by a compiler are predictable if that compiler conforms to the C++ standard while adhering to a platform's strict ABI family. At the centre of an ABI family is the OS and its gABI. Although the gABI does not directly influence object layouts, it does dictate which supplementary psABI and C++-ABI is supported. The psABI, supplied by the architecture, defines the data sizes and alignment requirements, impacting object

```
1  class  Mortal {
2    int age, lifespan;
3    virtual void die() {...}
4    ... };
5  class Humanoid : public Mortal {
6    int language;
7    virtual void speak() {...}
8    ... };
9  class Human : virtual public Humanoid {
10   int humanDNA;
11   virtual void getHumanDNA() {...}
12   ... };
13 class Navi : virtual public Humanoid {
14   int naviGenetics;
15   virtual void getNaviGenetics() { ... }
16   ... };
17 class Avatar : public Navi, public Human {
18   int hybridGenetics;
19   virtual void remotelyControl() { ... }
20   ... };
```

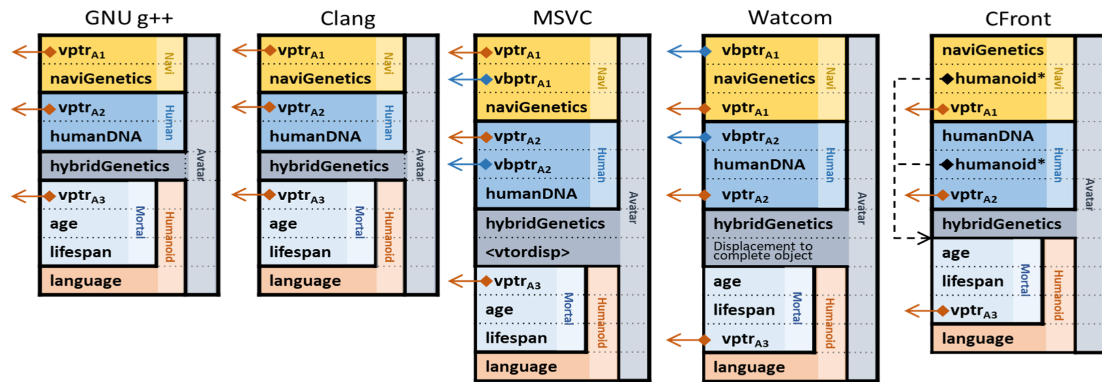


Figure 3: Object layouts produced from identical source code on five different compilers

sizes, data members sizes and padding. It also supplies the calling conventions, dictating how member functions interact with their object instances. The C++-ABI defines the layouts of objects and their supporting auxiliary data structures, meaning the arrangement of the data they store, not their size or alignment. The auxiliary data structures used to aid run-time mechanisms are linked to run-time objects using pointers. How many pointers are required and what positions they take within an object are outlined in the C++-ABI. However, this is only helpful if a compiler follows a well-documented ABI, like GNU g++ and LLVM Clang. The MSVC compiler acts as a defacto standard for any Microsoft Windows platform, and although their online documentation is growing, historically, many aspects of Microsoft's interfaces were not openly available. So, to understand the C++-ABI for Microsoft Windows, reverse engineering may still be required to bridge the gaps in their documentation.

Finally, if a compiler vendor wants to create code that is interoperable with other vendors and pre compiled libraries, it is essential they conform to the C++ standard and all ABI specifications for the platform. However, it is important to note that compiler vendors are not compelled to conform. Some vendors may have different goals other than interoperability, like performance or low memory consumption, which may be better achieved by breaking the standardised ABI. In fact, beyond the major compiler vendors, many are not up to date with the C++ standard, which evolves over time. Furthermore, even those committed to adhering to strict specifications often offer overriding features that allow programmers to break that conformance. So if one requires knowledge of exact object layout and data member positioning, especially for a program they did not compile themselves, reverse engineering is essential.

Chapter 2

Binary Representation of Objects

We now know why different compiler vendors on different platforms can produce different object layouts. However, we also know that the major compiler vendors often conform to a family of ABI specifications and therefore produce identical object layouts for interoperability. So, let us look at one of these major compiler vendors on a specific platform and talk through its object representation.

2.1 Introduction

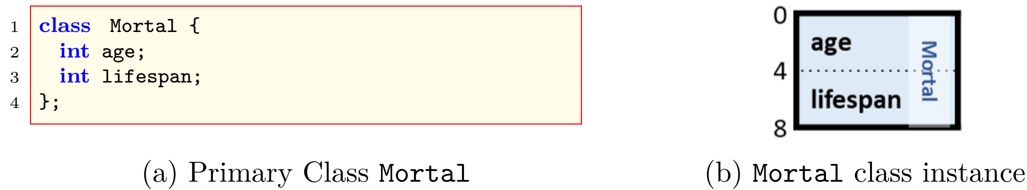
Finding modern introductory literature on the low-level C++ object model layout and the mechanism used to support it was surprisingly difficult. While researching all topics within this thesis, we looked at many scientific papers that discuss low-level C++ implementation. Most of these papers [13, 16, 21, 24, 37, 38, 101, 104, 112, 138, 141, 144] explain a particular aspect of OO implementation relevant to their work but do not include any reference to further reading (outside of ABI documentation). The few papers that did [35, 140] pointed to a standard reference book on C++ object layout [74], which is now over 25 years old. The book reflects on early compilers, in particular, Cfront [6] (released in 1985), discussing topics which were then of debate in the C++ compiler community. However, Cfront

was abandoned sometime after its final release in 1993 [80], and since the book's release, object layout representation has stabilised, thanks to standardisation and formal ABIs specifications. The lack of modern references in this space suggests a need to revisit the topic of low-level C++ within a modern-day compiler.

In this chapter, we will provide a modern look into the C++ object model layout, looking specifically at the GNU C++ Compiler (g++) on an AMD64 System V platform. We do not assume any prior knowledge of low-level C++ concepts. Concepts will be broken down from start to finish, starting with basic inheritance, moving towards the management of polymorphism and ending with a discussion of the layouts and mechanisms that support polymorphism, run-time type information, virtual inheritance and the dynamic dispatch mechanism. By discussing the specifics of the g++ compiler on an AMD64 System V platform, we, by default, are discussing the requirements of the Itanium ABI and all the relevant supplementary psABIs. So, although we are specifically looking at g++, thanks to strict ABIs, the same object layouts will be seen in any Itanium ABI conforming compiler (on an AMD64 System V platform). Before the chapter concludes, we will briefly compare object layouts found in the MSVC ABI to those we have seen in the Itanium ABI. This comparison serves as an example of the different layouts one might find in the wild, depending on the systems they use.

2.2 Representing Inheritance

This section demonstrates precisely how the g++ compiler realises each type of object for each hierarchical circumstance. Each hierarchical circumstance will be introduced gradually, with commentary and a small source code snippet. Each source code snippet will add a new class or attribute to the class hierarchy and is presented alongside a new object representation, demonstrating how the new aspect to the class hierarchy is represented at the binary-level.



(a) Primary Class `Mortal` (b) `Mortal` class instance

Figure 4: Primary class object

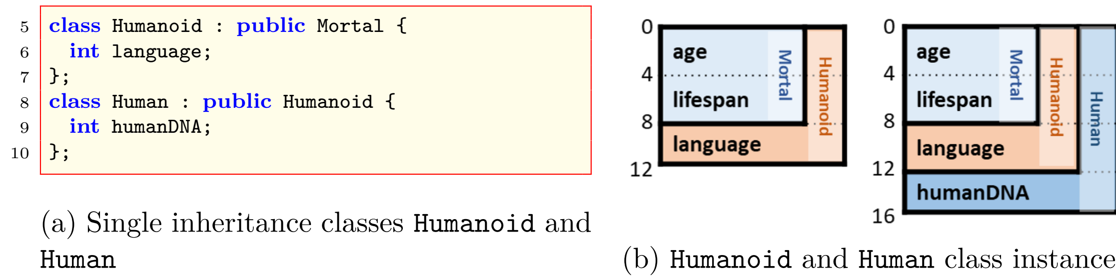
2.2.1 Primary Class

A **primary class** does not inherit from any other class. Figure 4a introduces the source code of a primary class called `Mortal` alongside its object representation in Figure 4b. The `Mortal` object contains two integer attributes, `age` and `lifespan`, known as that object's **data members**. These data members reflect the exact ordering of the attributes defined within the class. As both members are integers, they occupy 4 bytes of data and are aligned to 4-byte offsets due to the integer size requirements defined in the AMD64 psABI (Table 1). Notice that the total size of a `Mortal` object is 8-bytes, so no padding was added to the object. The largest alignment value of the object's data members is 4 bytes, so the object itself must be aligned to 4 bytes as well. This means that `Mortal` objects will always exist at a memory location divisible by 4 (any location ending in `0x0`, `0x4`, `0x8`, or `0xc`) as long as no overriding compiler commands (like pragmas) have been used during compilation.

2.2.2 Single Inheritance

Inheritance allows for a parent-child relationship between classes, where the child class (known as the **derived class**) inherits all attributes (and functions) of its parent class (known as its **base class**).

Figure 5a introduces the source code of two classes called `Humanoid` and `Human`.



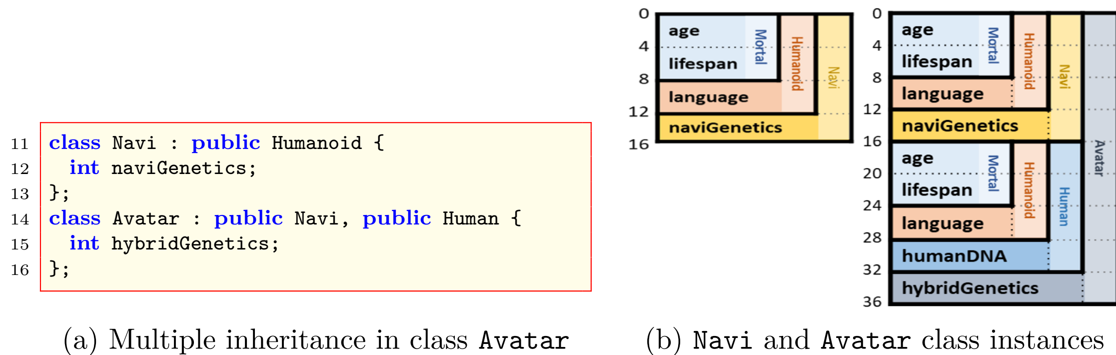
(a) Single inheritance classes `Humanoid` and `Human`

(b) `Humanoid` and `Human` class instances

Figure 5: Single inheritance objects

Along with the mortal class in Figure 4a, these classes make up a single inheritance hierarchy. We can describe the relationships between these classes in multiple ways: the `Mortal` class is a **base class** of both the `Humanoid` and `Human` classes, but is also a **direct-base class** of `Humanoid`; the `Humanoid` class is both a base class and direct-base class of `Human`, and both a **derived class** and **direct-derived class** of `Mortal`; the `Human` class is a derived class of both the `Mortal` and `Humanoid` classes but is also a direct-derived class of `Humanoid`. The `Human` class can also be described as the **most-derived** class of the hierarchy, meaning no other classes inherit from it. With these descriptions, we can define single inheritance as a hierarchy in which every class has a maximum of one direct-base class.

Derived classes inherit all the attributes of their base classes. In memory, this inheritance relationship between base and derived classes is represented using **sub-objects**. A sub-object is the complete object representation of a base class inside the memory region of a derived class object. Sub-objects mirror the inheritance hierarchy within the source code, where direct-base classes are represented as **direct-sub-objects**. Figure 5b demonstrates how the base classes `Mortal` and `Humanoid` are represented as sub-objects within their derived class instances, but more specifically, it shows how `Mortal` is a **direct-sub-object** of `Humanoid` and an **indirect-sub-object** of `Human`.




```
11 class Navi : public Humanoid {
12   int naviGenetics;
13 };
14 class Avatar : public Navi, public Human {
15   int hybridGenetics;
16 };
```

(a) Multiple inheritance in class `Avatar` (b) `Navi` and `Avatar` class instances

Figure 6: Multiple inheritance

2.2.3 Multiple Inheritance

In a multiple inheritance hierarchy, at least one class has two or more direct-base classes. Figure 6a introduces an example of multiple inheritance with the `Avatar` class. The `Avatar` class inherits from a newly introduced `Navi` class and the previously discussed `Human` class from Figure 5a. These two classes are presented in an inheritance list within the `Avatar` class declaration; this list is known as a class's **base-specifier-list**. The order direct-base classes appear in a base-specifier-list is reflected in the order their sub-objects appear within the inheriting class instance. Figure 6b depicts both a `Navi` and `Avatar` instances, demonstrating how classes with multiple direct-bases contain multiple direct-sub-objects.

2.2.4 Virtual Inheritance

The hierarchy built so far is a classic example of the notorious diamond problem [78], where one or more base classes are inherited multiple times by a derived class (see Figure 7). In the `Avatar` class example, the `Humanoid` class is indirectly inherited twice via the `Navi` and `Human` classes, resulting in an `Avatar` object with duplicate sub-objects (`Humanoid` and `Mortal` appear twice in the complete `Avatar`

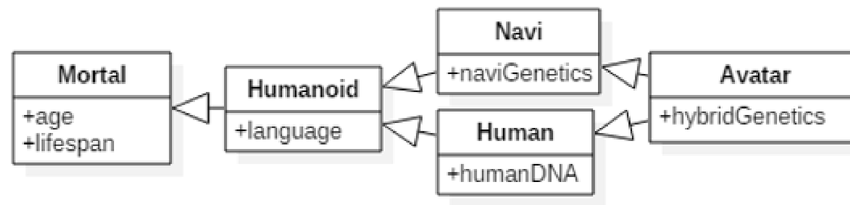
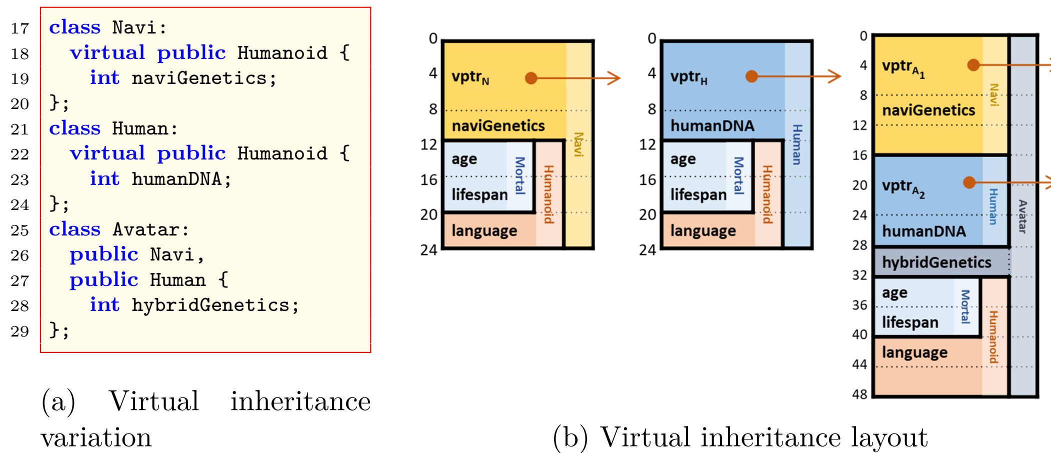


Figure 7: UML diagram of current hierarchy demonstrating the diamond problem



```
17 class Navi:
18   virtual public Humanoid {
19     int naviGenetics;
20 };
21 class Human:
22   virtual public Humanoid {
23     int humanDNA;
24 };
25 class Avatar:
26   public Navi,
27   public Human {
28     int hybridGenetics;
29 };
```


(a) Virtual inheritance variation

(b) Virtual inheritance layout

Figure 8: Virtual inheritance

object depicted in Figure 6b). This duplication can cause ambiguities when attempting to access data members with multiple instantiations. For example, if we want to increment an `Avatar`'s age, which age attribute is incremented? Without due care, duplication of data members can often result in compile-time errors because a compiler cannot decipher which data member to access.

Virtual inheritance prevents the ambiguities found in the classic diamond problem. Any virtually inherited class is guaranteed to appear once, and only once, as a sub-object in any derived class instance. Figure 8a provides a revised version of our multiple inheritance hierarchy, where the `Navi` and `Human` classes virtually inherit from `Humanoid`. Because `Humanoid` is virtually inherited, its sub-object representation is no longer constructed inline (at the top) with their inheriting class instances but instead resides at the bottom of the most-derived class instance, as seen in Figure 8b.

Virtual inheritance requires additional run-time mechanisms for data member access, which is not required in other types of inheritance hierarchies. Why this is required is covered in a later section (Section 2.4), but for now, just know that additional mechanisms are needed. The g++ compiler supports these additional mechanisms using an auxiliary data structure called a **virtual table** (Section 2.4.2). The virtual table (vtable) is addressed from the object itself, using a **virtual pointer (vptr)**. The vptr is an implicit[1] data member positioned at the top of any complete object or sub-object that virtually inherits from another. The vptr is a pointer data type and, due to psABI requirements (Table 1), is both 8-bytes in size and alignment. The addition of an implicit 8-byte data member increases both the size and the alignment of the `Navi`, `Human`, and `Avatar` class instances (Figure 8b). All these objects now have an alignment of 8 and a size that must be divisible by that alignment value (Section 1.5.2.2). To satisfy these size and alignment requirements, padding is used within the `Avatar` object to align vptr data members and expand the object's overall size.

2.2.5 Other Class Keywords and Templates

One might find other keywords within their class definitions, such as `const`, `public`, `private`, and `protected`. These terms tell the compiler how to deal with specific data and facilitate compile-time safety checks. For example, the compiler will flag an error if an attempt was made to alter a `const` variable at compile-time. The same is true for illegal access to `private` and `protected` class members. However, when these keywords are used to define class attributes, they do not affect how data is realised at the binary-level and therefore do not affect object layout.

[1] meaning automatically generated by the compiler.

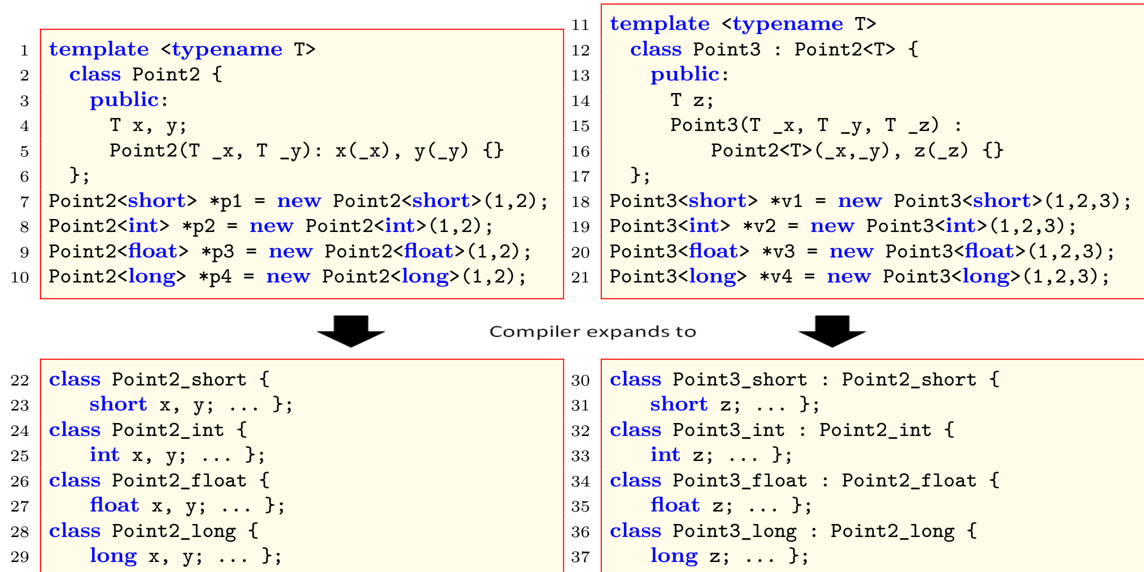


Figure 9: Template expansion

Static When a class attribute is declared as `static`, it means that the attribute should be common to all instances of that class, i.e., a single copy should be shared between all instances of that class at run-time. Because a static attribute is shared between all instances, it cannot appear in all instances. Instead, the compiler will place this attribute elsewhere in memory and address it directly during execution. So, class attributes declared as `static` will never appear in object instances.

Templates The ABI will treat template classes the same as non-template classes. The reason is that template classes are ultimately expanded into non-templated code by the compiler. For example, in Figure 9 (lines 1-21), we have declared two class templates, `Point2` and `Point3`, that are part of a single inheritance hierarchy. Both of these class templates are instantiated using four different template parameters: `short`, `int`, `float`, and `long`. These instantiations will be expanded at compile-time, creating eight class specialisations similar to those listed in lines 22-37. After expansion, there is no longer any templated code, and therefore no

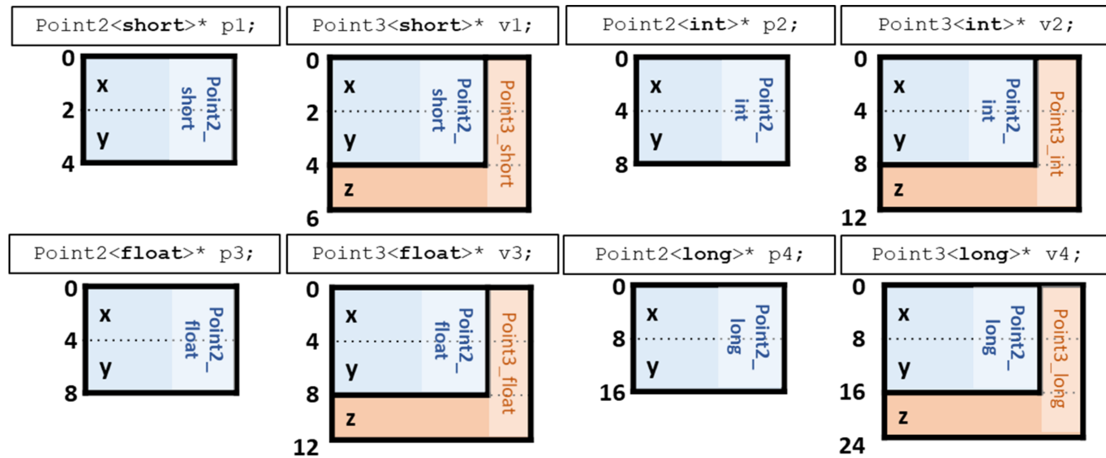


Figure 10: Objects generated from class templates

need to treat these classes differently. Figure 10 depicts the object layouts generated from each class specialisation. Notice that the lengths of the data members of each instance reflect the lengths outlined in the Itanium ABI (Table 1).

2.3 Polymorphism and the Type Systems

Polymorphism, in the general sense, refers to something that can take several different forms. In the context of OO, polymorphism refers to variables, objects and functions that can take several different run-time forms from within one inheritance hierarchy. This section will explain both high-level and low-level polymorphism in C++.

2.3.1 Types

All run-time objects have a **type**, where the type of an object is defined by its class. Classes can have derived classes, and similarly, types can have **derived-types** (also known as sub-types). For example, the type of an object created from the `Mortal` class is `Mortal`. The `Mortal` class has a derived class called `Human`, where any `Human` class instance is a `Human` type but is also a derived-type

of `Mortal`.

It is common to describe a run-time object as "type X" rather than "an instance of class X". These two phrases mean the same thing, but there is a difference between types and classes. Classes are a method of creating new data types, and these types are a property assigned not just to the objects those classes create but also to the variables that store those objects.

2.3.2 Polymorphic Variables and Object Address-Points

Static Types At the source level, objects are assigned to variables and variables are declared with a given type. The expression

```
Mortal obj = Mortal();
```

creates a variable called `obj` of `Mortal` type that stores a `Mortal` object at the point of construction. The type a variable is declared (at the source-level) is called that variable's **static type** and is known at compile-time. The static type of a variable can also be defined as a pointer type (`Mortal*`) or reference type (`Mortal&`).

Dynamic Type A variable can address an object that matches its static type or any derived-types of its static type. For example, the expression:

```
Mortal* objPtr = new Human();
```

creates a variable called `objPtr` of `Mortal*` type that, at the point of construction, will address a `Human` object. Variables, therefore, have both a static and **dynamic type**, where the dynamic type is the type of the object a variable addresses at a given moment in run-time execution. We say a given moment because the dynamic type of a variable can change throughout program execution. This is the nature of polymorphism and **polymorphic variables**.

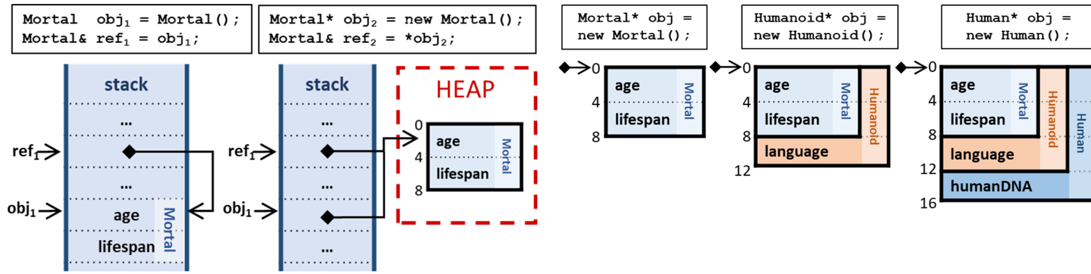


Figure 11: Different object assignments

Figure 12: Object address-points

Variables at the Binary-Level A source-level variable is realised as a stack placeholder at run-time. That variable might be a placeholder for a complete object instance (`Mortal obj`$_1$) or a placeholder for an address to lookup the object instance in memory (`Mortal* obj`$_2$). Figure 11 demonstrates these types of variables as well as reference types (`Mortal& ref`$_i$). Reference types are semantically different at the source-level but are realised in the same way as pointer variables at the binary-level. Examples throughout this section will focus on heap objects, as the heap allows for dynamic memory, which permits all polymorphic capabilities.

Object Address-Points A variable that addresses an object will address a specific location within that object, which we will call an object's **address-point**. In single inheritance, objects have a single address-point at the zero offset of any complete object, as seen in Figure 12 (address-points depicted with black arrows). In multiple and virtual inheritance, objects have multiple address-points, one at a zero offset to the complete object and others at zero offsets from its sub-objects.

The static type of a variable will dictate the address-point of an object at run-time, as the **variable must always address an object of its declared type**. This means multiple and virtual inheritance objects may be addressed by a sub-object instead of the complete object.

Figure 13 shows several instances of the `Avatar` object, demonstrating it has two address-points, one at the zero offset and another at offset 16 (or zero offset of the `Human` sub-object). The figure also shows all seven possible static types a

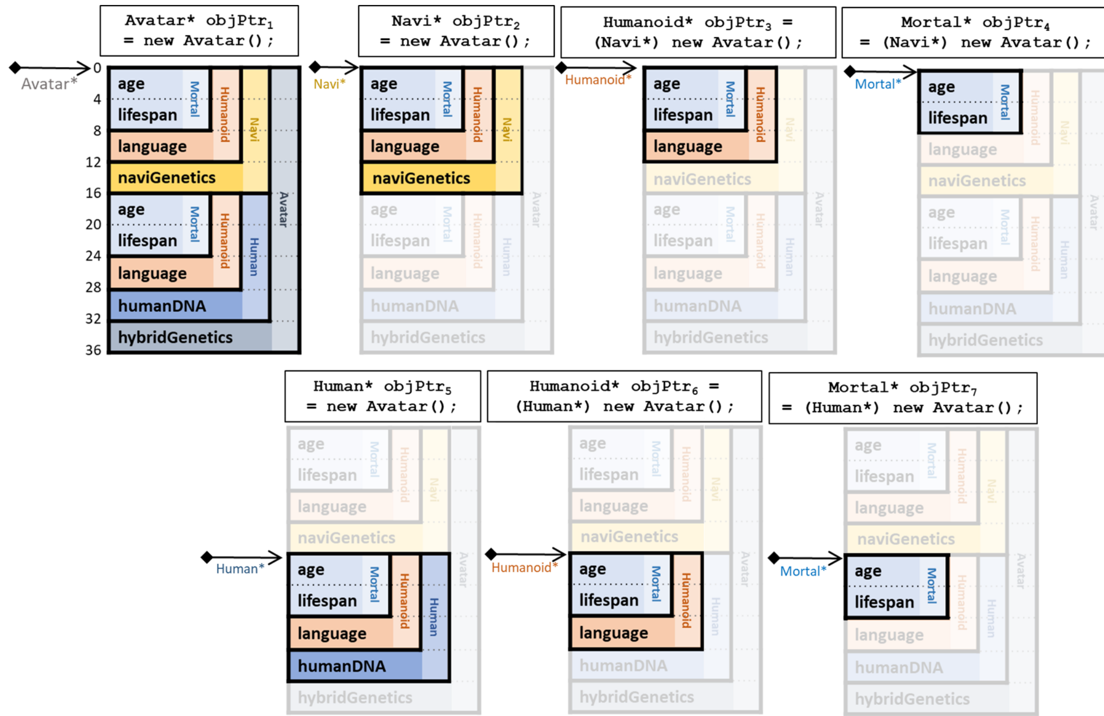


Figure 13: Derived-type assignment and corresponding sub-object address-points

variable can be assigned (`objPtr`$_{1\text{-}7}$) when addressing an `Avatar` object and the resulting address-points in each case.

2.3.3 Type Checking

During compilation, the compiler performs a static type check that verifies a program's type-safety at the source level. Static type-checking is applied to all variables. Object variables, when referenced in the source code, are often interacting with either a data member or member function (Section 2.3.5). Data members and member functions (or just members) are declared within a specific class and, therefore, have an associated class type at compile-time. When interacting with these members through a variable, the compiler has a set of expected static types that the variable can be (i.e. any class type that declared or inherited those members). Static type-checking, therefore, verifies that the static type of a variable

matches one of the expected types associated with the members it is interacting with. In other words, we can say that an object variable has **access rights** only to the data members and member functions of its static type at compile-time due to the type-safety mechanisms employed by the compiler.

For example, consider the pointer variable $\texttt{objPtr}_5$, which was defined as $\texttt{Human* objPtr}_5\texttt{ = new Avatar()}$ in Figure 13. This variable has access rights to the data members of the `Human` object and its sub-objects. However, any attempt to access data members outside of the `Human` sub-object (i.e. $\texttt{objPtr}_5\texttt{->hybridGenetics}$ an `Avatar` attribute) will result in a compilation error, even though that member exists within the complete `Avatar` object. This happens because of the static type-checking employed by the compiler, which sees $\texttt{objPtr}_5$ as a `Human` object (its static type) and not an `Avatar` object (its run-time type). Figure 13 highlights all the accessible data members for each static variable type and fades those inaccessible at compile-time due to static type-checking.

2.3.4 Casting

When a variable's static and dynamic types differ, access to members of the dynamic type (without compilation error) can only be achieved through explicit casting. Casting is the act of changing a variable's perceived type (the **source type**) to another type in the class hierarchy (the **target type**). Although casting does not alter the data or type the variable addresses, it does change the compiler's perception of that variable. After casting a variable, the compiler will associate it with the target type of the cast and, in turn, grant that variable access to target type data members and functions.

For example, consider both variables $\texttt{objPtr}_2$ and $\texttt{objPtr}_5$ from Figure 13. Both variables will address a run-time `Avatar` type but statically appear as base types `Navi` and `Human`. Suppose a programmer attempts to access the `Avatar` attribute `hybridGenetics` directly (i.e. $\texttt{objPtr}_{2/5}\texttt{->hybridGenetics}$); then the

programmer will receive a compile-time error in both cases, thanks to static type-checking. This is because the compiler can only associate those variables with their declared static types, and those types do not have a `hybridGenetics` attribute. To fix this, the programmer must explicitly cast these variables before accessing `Avatar` data (i.e. `(Avatar*)objPtr`$_{2/5}$`->hybridGenetics`). This action changes the compiler's perception of the `objPtr`$_2$ and `objPtr`$_5$ variables, granting them access to `Avatar`'s data members and functions.

Things are different at the binary-level, because we are handling physical memory. Casting will take a variable's assigned address-point and offset it to the address-point associated with the target type. For example, the expression `((Avatar*)objPtr`$_5$`)->hybridGenetics` will first perform the cast `((Avatar*) objPtr`$_5$`)`, which adds `-16` to the address stored in `objPtr`$_5$, returning a pointer that addresses the complete `Avatar` object. With this new address-point, `Avatar`'s data members become accessible. Casting must be performed because the compiler only stores limited information about where attributes exist in memory. In fact, the compiler only knows the location of class attributes relative to the address-point of their own class instances. So the compiler knows that the data member, `hybridGenetics`, exists at a `+32` offset from an `Avatar`* address-point but will never know its location from a sub-object address-point. Put simply, the compiler cannot resolve the `(objPtr`$_5$`)->hybridGenetics` data access without casting because it cannot deduce the offset required to access the `hybridGenetics` attribute from a `Human`* address-point.

We note that casts between source and target types that share the same address-point (i.e. `((Avatar*)objPtr`$_2$`)`) are treated no differently. In this case, the offset adjustment made to `objPtr`$_2$ is `+0`. However, the compiler optimises such adjustments out of the final assembly. Thus these casts will not appear at the binary level.

```
1  class  Mortal {
2    int age, lifespan;
3    void incrementAge() {...}
4    void die() {...}
5  };
6  class Humanoid : public Mortal {
7    int language;
8    void speak() {...}
9    void move() {...}
10 };
11 class Human : public Humanoid {
12   int naviGenetics;
13   void speak() {...}
14   void getHumanDNA() {...}
15 };
```

Listing 2.1: Member functions

```
16 int main() {
17 Mortal   *m  = new Mortal();
18 Humanoid *h  = new Humanoid();
19 Human    *p  = new Human();
20 Humanoid *hp = new Human();
21
22 m->move();   //compile error
23 h->move();   // fine
24 p->move();   // fine
25
26 m->speak();  //compile error
27 h->speak();  //Humanoid::speak()
28 p->speak();  //Human::speak()
29 hp->speak(); //Humanoid::speak()
30 }
```

Listing 2.2: Calling member functions

2.3.5 Member Functions

A **member function** is declared and defined within a class; it is described as a member of that class and any object created from that class. Like data members, member functions are inherited and are members of both the inheriting classes and their instances. Member functions are invoked using an object variable and only pass compile-time static type-checking if that function is a member of the variable's static type. For example, Listing 2.1 introduces several new functions to our hierarchy; the `Humanoid::move()` function is a member of the `Humanoid` class and the `Human` class by inheritance. This function can be invoked by instances of the `Humanoid` and `Human` classes (lines 23 and 24 of Listing 2.2), but not by an instance of the `Mortal` class (lines 22 of Listing 2.2) of which it is not a member.

How Member Functions Interact with Objects At the source level, we define a member functions with zero to n parameters:

$$\texttt{ClassType::funcName(Type}_1\ \texttt{p}_1\texttt{, ..., Type}_n\ \texttt{p}_n\texttt{)\{...\}}$$

and call that function using an object variable

$$\texttt{obj->funcName(p}_1\texttt{,...,p}_n\texttt{)\{...\}}.$$

At the binary-level, the compiler realises a function call by passing the invoking object to the function as its first parameter:

$$\texttt{funcName(obj,}p_1\texttt{,...,}p_n\texttt{)\{...\}}$$

and is known as a member function's **implicit object parameter**. Member function definitions can therefore be thought of as having $n + 1$ parameters after compilation, taking the form:

$$\texttt{ClassType::funcName(ClassType* obj, }Type_1\ p_1\texttt{,..., }Type_n\ p_n\texttt{)\{...\}}$$

where the implicit object parameter (`obj`) can be of type `ClassType` or any of its derived-types. This object parameter can be referenced within the body of the function definition using the `this` keyword. For this reason, it is common to refer to a function's implicit object parameter as its `this-pointer`.

Implicit Casts The body of a member function is realised solely to interact with the object layouts of its own type. So when a member function is invoked using a variable of a derived-type, an implicit cast must be performed. An **implicit cast** is a cast performed by the compiler, which in this case, ensures the implicit object passed to the function is of the same type as the function's `this-pointer`. For example, if a variable of `Human` type is used to invoke the `Mortal::incrementAge()` function, then the `Human` variable will first be implicitly cast to a `Mortal` object before the function is called. In this particular case, we know that `Mortal` and `Human` share the same address-point (Figure 12) due to single inheritance; the implicit cast will therefore add a zero offset to the `Human` variable. At the binary-level, this operation would be redundant, so the g++ compiler omits such adjustments. For multiple and virtual inheritance, on the other hand, an implicit cast may result in an address-point adjustment at the binary-level, if the source and target types have different address-points.

Function Overriding Inherited member functions can be redefined as part of an inheriting class. Redefining functions in derived classes is called **function overriding**. When a member function is overridden, multiple implementations of that function will exist, which to the programmer appear to share the same

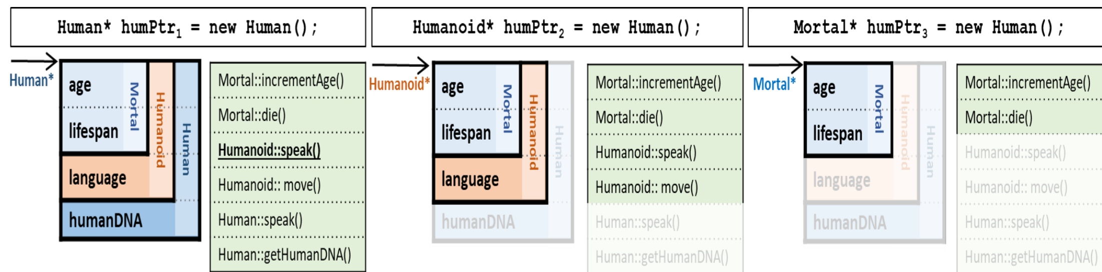


Figure 14: Static variable access rights at compile-time

function signature. However, to the compiler, each has its own implicit object parameter, so which function implementation is executed when called will depend on the static type of the object used to invoke it. In Listing 2.1, line 8 defines a new function, `speak()`, within the `Humanoid` class. This function is then overridden in Line 13 of the `Human` class. Lines 27-29 of Listing 2.2 demonstrate which instance of the `speak()` function is invoked based on the static type of the variable used to call it. Of course, as the function is not a member of the `Mortal` type, calling this function using a `Mortal` object (line 26 of Listing 2.2) produces a compile-time error. Figure 14 provides a visual example of the compile-time access rights (to both data members and member functions) of each possible static variable type used to store a `Human` object[2].

Virtual Functions To invoke a function based on the dynamic type of the invoking object variable, the function needs to be declared as virtual. Non-virtual functions, like those seen so far, are realised as direct function calls, as they are resolved at compile-time, based on the static type of the variables used. A virtual function is realised as an indirect function call, using a mechanism called **dynamic dispatch**. Dynamic dispatch determines which version of a function is called during run-time execution, based on the dynamic type of the variable used.

Like virtual inheritance, virtual functions use the same vtables to aid the

[2]It is still possible for a `Human` objects to call the `Humanoid::speak()` function, but it must be called directly, i.e. `humPtr`$_1$`->Humanoid::speak()`. This is why the function is underlined in the figure.

dynamic dispatch mechanism. Therefore, the use of virtual functions will add the same virtual pointer (vptr) found in objects with virtual inheritance. These vptrs are added to all object types with virtual member functions, which alters the layout of these objects. Any object that has a virtual pointer is called a **dynamic object**.

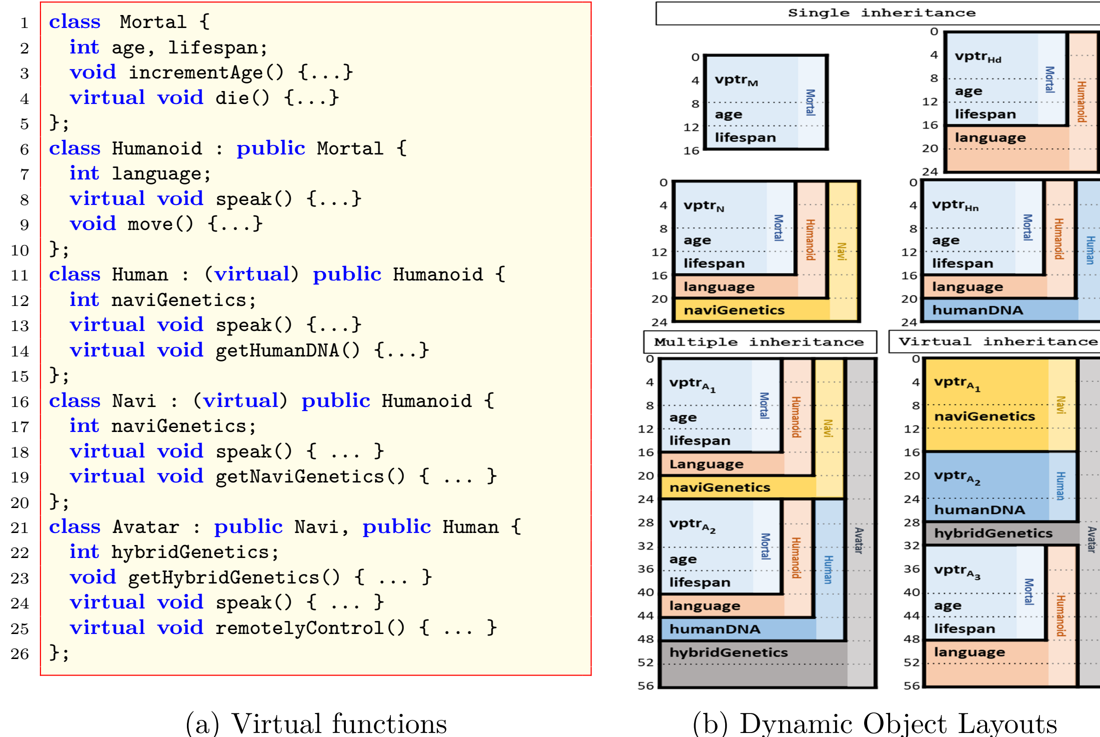
2.4 Dynamic Objects and Supporting Data

A dynamic object is simply an object that contains a vptr. Classes that produce objects with a vptr are also known as **dynamic classes**. A class is dynamic if it declares a virtual function, inherits a virtual function, or is part of a virtual inheritance hierarchy. This section will discuss dynamic classes, the layout of their dynamic objects, the vtables associated with these classes, and the dynamic dispatch mechanism.

2.4.1 Object Layouts with Virtual Pointers

Every dynamic class will have its own unique vtable. Vtables support several run-time mechanisms associated with that class and its instances. To access this support at run-time, every instance of a dynamic class will store an implicit virtual pointer (vptr) that addresses that class's unique vtable. In cases of inheritance, objects will store the vptr(s) of the most-derived object's class, giving them access to run-time mechanisms specific to the run-time type of the object. These mechanisms will be discussed in Sections 2.4.3 and 3.4.4.

Figure 15a alters our hierarchy so that every class now contains a virtual function, making every class dynamic. Note that we have placed the `virtual` inheritance keyword in parentheses to avoid repetitive code listings. Assume that this `virtual` keyword is only present in the case of virtual inheritance. Figure 15b depicts the object structures of the new hierarchy listed in Figure 15a. As every



(a) Virtual functions (b) Dynamic Object Layouts

Figure 15: Dynamic object layouts with corresponding source code

class in this hierarchy is dynamic, every instance of these classes (complete or sub) has an implicit vptr data member associated with the complete object's type. Vptrs reside at the top of an object or sub-object structure, i.e. at a zero offset from a (sub-)object's address-point. Where (sub-)objects share an address-point, they also share a vptr (discussed further in Section 2.4.2). Remember that vptrs have a size and alignment of 8 (Table 1), so for some class instances, padding is needed to meet the size and alignment requirements of the object and its data members.

A Mix of Dynamic and Non-Dynamic Classes We previously stated that the order direct-base classes appear in a base-specifier-list is reflected in the order that their sub-objects appear within their inheriting class instance. When classes inherit from both dynamic and non-dynamic base classes, this is no longer guaranteed. Figure 16 depicts three object layouts (and their address-points) of class

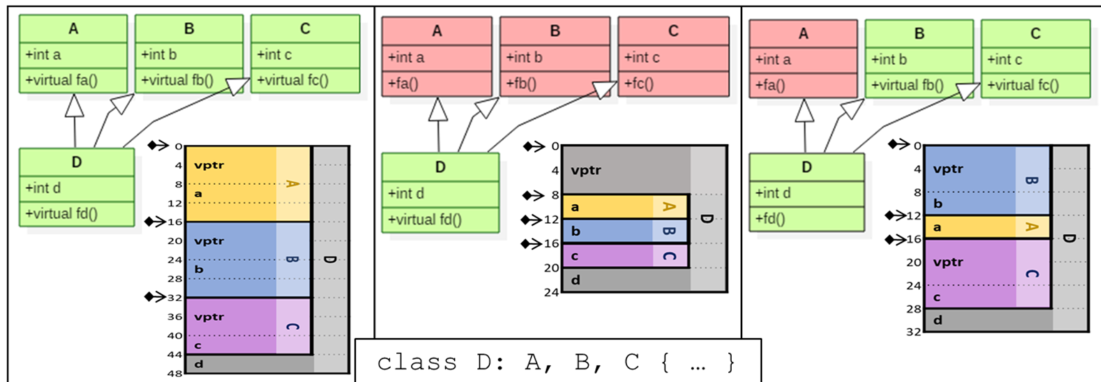


Figure 16: Dynamic object layouts and dependencies

D, where D inherits from three other classes A, B, and C, in that order. Each version of D is dynamic (through virtual function declaration or by inheriting a dynamic class), but the dynamic status of its base classes change with each depiction. For quick reference, any class highlighted in green is dynamic, and those highlighted in red are non-dynamic. The first D object (from left to right) is generated when all base classes are dynamic, which is no different from the objects we have seen thus far. The second D object is generated when D is the only dynamic class in the hierarchy, inheriting solely non-dynamic base classes. In this case, none of the sub-object require a vptr, so they are positioned after D's vptr. The final D object is generated when classes B, C and D are dynamic, but A is non-dynamic. To achieve better run-time efficiency in the compiled code, the g++ compiler will position the first dynamic base class (from the base-specifier-list) as the first sub-object within the complete object. This means, for the final D object, the compiler deviates from the declared order of the base-specifier-list and forces B and D to share a vptr. Vptr sharing reduces the number of address-points an object has, which in turn reduces the number of possible address-point adjustments that can occur during execution. By forcing B and D to share a vptr, the (non-dynamic) sub-object A is shifted down and appears as D's second sub-object after B.

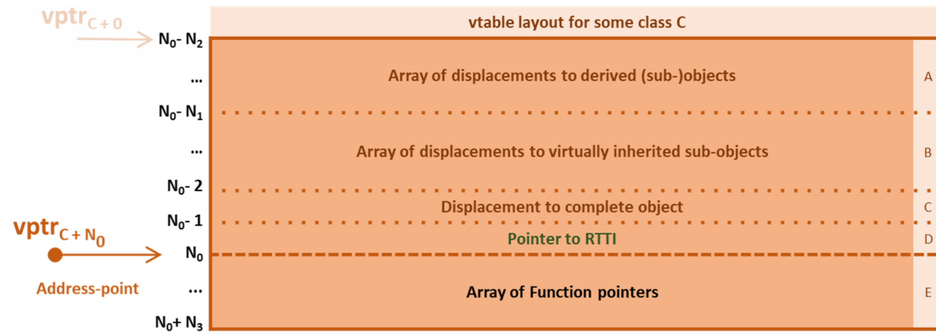


Figure 17: (Sub-)vtable layout as defined in the Itanium C++ ABI [23].

Vtable Entry	Description	Use
A	Array of displacements	This entry is only in the sub-vtables of a virtually inherited sub-object. It is an array of displacements used to perform an implicit cast to a derived object as part of an overridden virtual function call.
B	Array of displacements	This entry is only in vtables of classes with virtual inheritance. It is an array of displacements to each virtually inherited sub-object from the location of the specific vptr used to address the vtable.
C	Complete object displacement	This entry stores a single displacement value to the complete object (offset zero) from the location of the specific vptr used to address the vtable.
D	Pointer to RTTI	This entry stores the address of the class's run-time type information object (Section 2.4.4). This entry is always the first negative entry from the vptr address-point.
E	Array of virtual function pointers	This entry is an array of virtual function pointers, storing only the functions appropriate to the associated class type. This array is used to determine which version of an overridden virtual function is invoked at run-time. Function pointers appear consecutively and in the order they were declared.

Table 3: Vtable Entries Explained

2.4.2 Virtual Table Layout

Every dynamic class has a unique vtable. A vtable stores data that aids run-time mechanisms, such as dynamic dispatch and dynamic casting. To gain run-time access to a class vtable, every instance of that class stores a vptr addressing that vtable. A class's vtable, like its objects, has specific address-points (pointed to by a vptr). In cases of multiple and virtual inheritances, a complete vtable may consist of multiple sub-vtables.

Figure 17 depicts the layout of all (sub-)vtables. Unlike objects, a (sub-) vtable's address-point (pictured here as $\texttt{vptr}_{\texttt{C+N}_0}$) is not the lowest address of the data set but is positioned at a specific offset ($\texttt{N}_0$) so that certain data is accessible

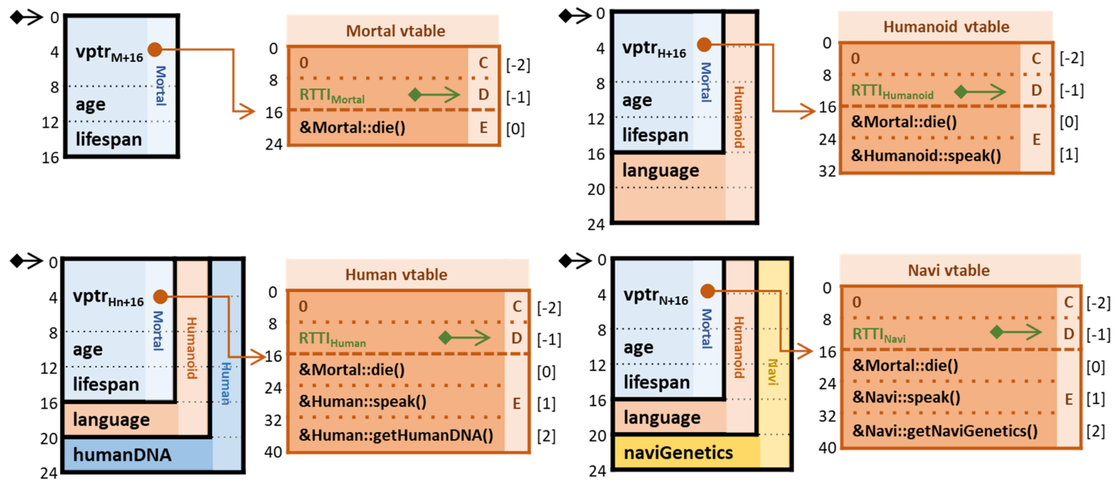


Figure 18: Vtable layout for single inheritance

at both positive and negative offsets from that location. We have labelled each vtable data entry A-E for ease of reference, and Table 3 further explains the use of each vtable entry. As these uses are not intuitive, the following examples should clarify further. All examples of vtable layouts were produced by compiling the source code listed in (Figure 15a).

Single Inheritance Example Figure 18 depicts the vtables for all our single inheritance objects. Notice that the first entry of each vtable (**labelled C**) is zero. This is the displacement value to the complete object's address-point, i.e. the offset adjustment required to get to the complete object's address-point from the position of the vptr used to access the vtable. In all cases, these objects have a single vptr positioned at the zero offset of the complete object, so no adjustment is needed; hence all entries are zero. The displacement value to the complete object is always present within a vtable at a negative offset from the vptr. Think of the vptr as a pointer to an array of 8-byte data chunks (containing integers and pointers); then, the displacement value will always exist at `vptr[-2]`. We have labelled the vtables in our example with array offsets as well as address offsets for ease of understanding.

Each of our vtables also contains an RTTI pointer (**labelled D**). This is a pointer to run-time type information, which will be discussed in full in Section 2.4.4. An RTTI pointer will always exist at the `vptr[-1]` entry of a vtable. We note that it is possible to exclude RTTI from compilation; in such cases, the RTTI pointer is set to null (`0`).

Each vtable pictured in Figure 18 has an array of virtual functions (**labelled E**). Notice that all derived class vtables contain a subset of inherited virtual function pointers within their own arrays. For example, the function `&Mortal::die()`, which was defined in the `Mortal` class, exists in every derived vtable due to inheritance. Virtual functions are given a specific entry within the virtual function pointer array, which is consistent across all derived class vtables. For the `&Mortal::die()` function, this entry is `vptr[0]`. When a derived class overrides a function, the function pointer in the derived vtable is also overridden. This can be seen with the `speak()` function that occupies the second array entry (`vptr[1]`). This function was declared within the `Humanoid` class but overridden by the `Human` and `Navi` classes. The different versions of the speak() function can be seen in each of the vtables, except for Mortal's vtable, as the `speak()` function is not a member of that class.

Multiple Inheritance Example Figure 19 depicts an example of a multiple inheritance vtable. In this case the object contains two vptrs ($\texttt{vptr}_{\texttt{A+16}}$ and $\texttt{vptr}_{\texttt{A+62}}$) addressing two distinct sub-vtables, at offsets 16 and 62, within the complete `Avatar` vtable. Notice here that the displacement value (labelled C) addressable from the second vptr ($\texttt{vptr}_{\texttt{A+62}}\texttt{[-2]}$) is `-24`. This is the displacement from the location of $\texttt{vptr}_{\texttt{A+62}}$ (within the object instance) to the zero offset. Also, notice that virtual function pointers accessible to each vptr are the virtual functions associated with the (sub-)objects that share that vptr. For example, the `&Human::getHumanDNA()` function is only accessible from the vptr within the

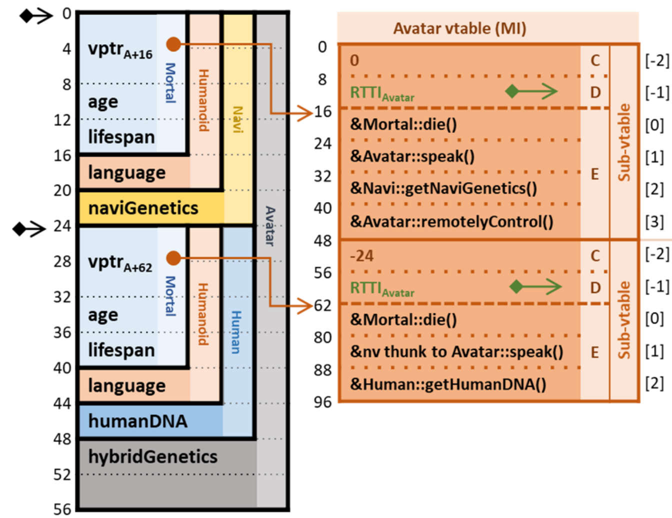


Figure 19: Vtable layout for multiple inheritance

`Human` sub-object. The same is true for the `&Navi::getNaviGenetics()` function, which is only accessible from the vptr within the `Navi` sub-object.

The `speak()` function is one that was defined in the `Humanoid` class and has been overridden by `Human`, `Navi`, and `Avatar`. This means that the `speak()` function should be accessible from both vptrs and exist in both sub-vtables, which it does at $\texttt{vptr}_{\texttt{A+16}}$`[1]` and $\texttt{vptr}_{\texttt{A+62}}$`[1]`. Notice that this function pointer exists at the second entry of the function array in both cases; even in sub-vtables, virtual functions still have the same position in the virtual function array. Also notice that the `speak()` function at $\texttt{vptr}_{\texttt{A+62}}$`[1]` is different to the one accessible at $\texttt{vptr}_{\texttt{A+16}}$`[1]`. In the first instance, `&Avatar::speak()` is a direct address to the `Avatar::speak()` function, whereas the second instance, listed as `& nv thunk to Avatar::speak()` is an address to a 'non-virtual thunk'. A thunk [52] (also known as a trampoline) is a small snippet of instruction code that performs an operation before jumping to another set of instructions. In this case, the code snippet is an implicit cast to the complete `Avatar` object before a direct jump to the `Avatar::speak()` function. This is because the `Avatar::speak()` function expects to receive an `Avatar` object, so we must cast to an `Avatar` address-point

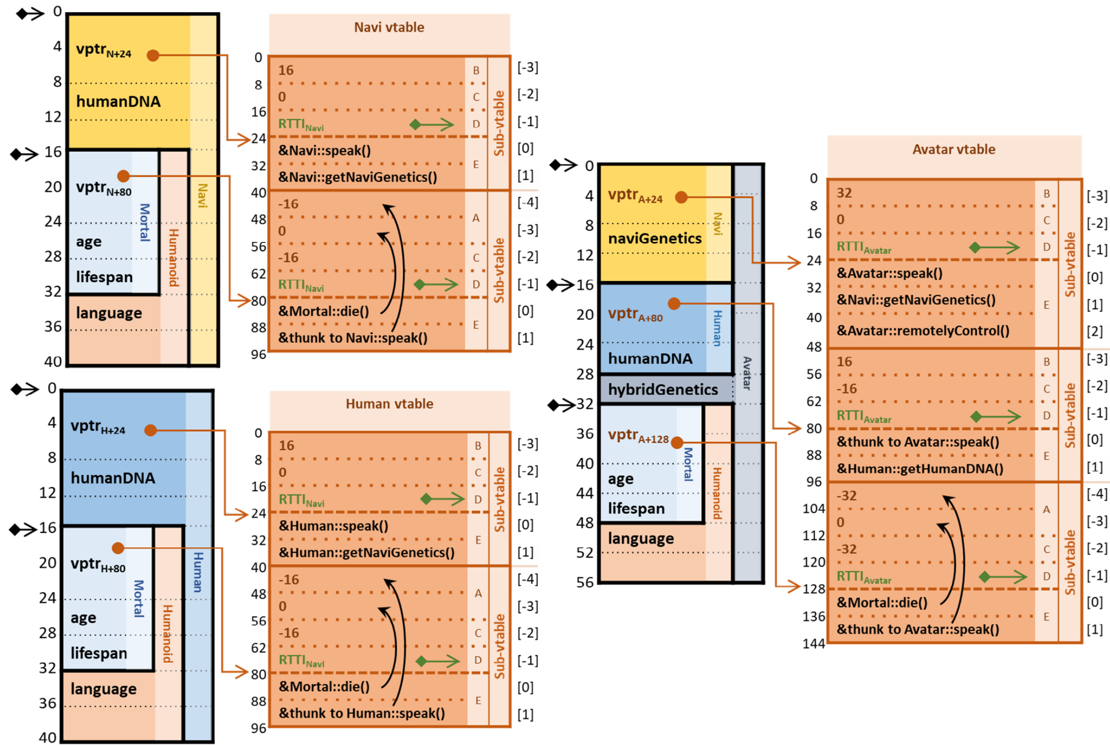


Figure 20: Vtable layout for virtual inheritance with virtual functions

before executing the function. In this case, the thunk is defined as 'non-virtual', which means that the offset adjustment performed during the implicit cast was known at compile time.

Virtual Inheritance with Virtual Functions Example Figure 20 depicts an example of several virtual inheritance vtables. In these vtables, we again have displacement values to the complete objects (labelled C) but also the displacement values to the virtually inherited sub-object (**labelled B**). For example, using the `Navi` vptr ($\texttt{vptr}_{\texttt{N+24}}$), we can access the displacement value to the virtual base at $\texttt{vptr}_{\texttt{N+24}}$`[-3]`. This displacement value is `+16`, which is the offset from the complete `Navi` object address-point to the virtually inherited `Humanoid` address-point. Notice that in the `Avatar` vtable, both the `Navi` and `Human` sub-objects can access a displacement to the `Humanoid` sub-object at the same vptr array entry `[-3]`, i.e. $\texttt{vptr}_{\texttt{A+24}}$`[-3] = +32` and $\texttt{vptr}_{\texttt{A+80}}$`[-3] = +16`, but the offsets stored

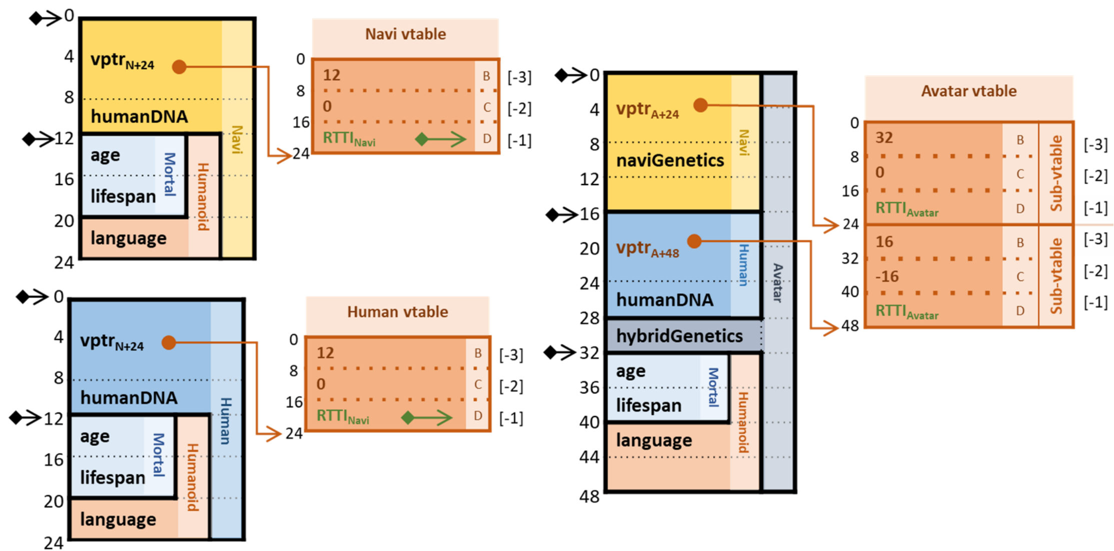


Figure 21: Vtable layout for virtual inheritance without virtual functions

are relative to the vptr used to access it.

Notice we again have an address to a thunk rather than a direct address to the `speak()` function in several vtables. In this case, these thunks are all virtual thunks, which means that the offset adjustment needed as part of the implicit cast is not known at compile time because it is object dependent. This is where the displacement values to derived (sub-)objects are utilised from the vtable (**labelled A**). This array stores offset values specific to the virtual functions of virtually inherited classes. So if the thunk is called from the `Avatar` vtable, then the implicit cast will use the `-32` offset at $\texttt{vptr}_{\texttt{A+128}}$`[-4]`, whereas if the thunk is called from the `Navi` vtable, then the implicit cast will use the `-16` offset at $\texttt{vptr}_{\texttt{N+80}}$`[-4]`. We drew extra arrows on these tables to show which functions map to which offsets. We will discuss the low-level execution of these types of function calls in Section 2.4.3.

Virtual Inheritance Finally, Figure 21 depicts the same virtual inheritance objects as Figure 20 but demonstrates what these vtables would look like if no virtual functions existed in the hierarchy. Notice that the virtually inherited class

```
1  class Humanoid : Mortal {
2    ...
3    virtual void speak() {...}
4    ... };
5  class Human : Humanoid {
6    ...
7    virtual void speak() {...}
8    ... };
9  class Avatar: Navi, Human {
10   ...
11   virtual void speak() {...}
12   ... };
```

Listing 2.3: Virtual member functions

```
13 int main() {
14 Humanoid *h1  = new Humanoid();
15 Humanoid *h3  = new Human();
16 Humanoid *h3  = new Avatar();
17
18 h1->speak();  //Humanoid::speak()
19 h2->speak();  //Human::speak()
20 h3->speak();  //Avatar::speak()
21
22 Mortal   *m   = new Humanoid();
23 m->speak();   //compiler error
24 ... }
```

Listing 2.4: Calling virtual member functions

`Humanoid` has no vptr or sub-vtable in this case because, without virtual functions, this class (and its object instance) is no longer dynamic.

2.4.3 Virtual Member Functions

Virtual member functions, unlike non-virtual functions, are resolved at run-time, where the exact implementation of the function executed is determined by the dynamic type of the invoking object. For example, Listing 2.3 highlights three implementations of the virtual function `speak()` in three different classes in the hierarchy. In Listing 2.4, three different objects are constructed (lines 14-16), each with a static type of `Humanoid`, but a different run-time type in each case. Each object is then used to invoke the `speak()` function (lines 18-20), where each invocation results in the execution of three different `speak()` function implementations, each determined by the dynamic types of the objects used. Of course, this is only possible if the source code passes all static type-checking performed by the compiler. In line 23 of Listing 2.4, a compile-time error occurs as we have tried to invoke the `speak()` function using a variable with a static type of `Mortal`. As `Mortal` does not have a member function called `speak()` (introduced in `Humanoid`), this invocation does not pass compiler static type-checking and causes a compile-time error.

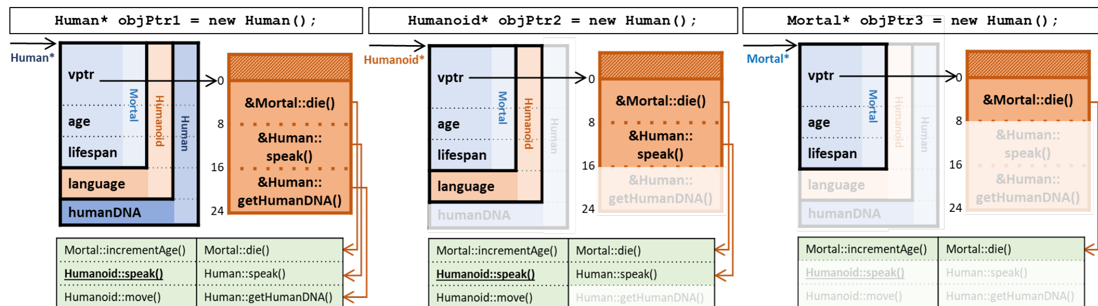


Figure 22: Access rights of dynamic objects

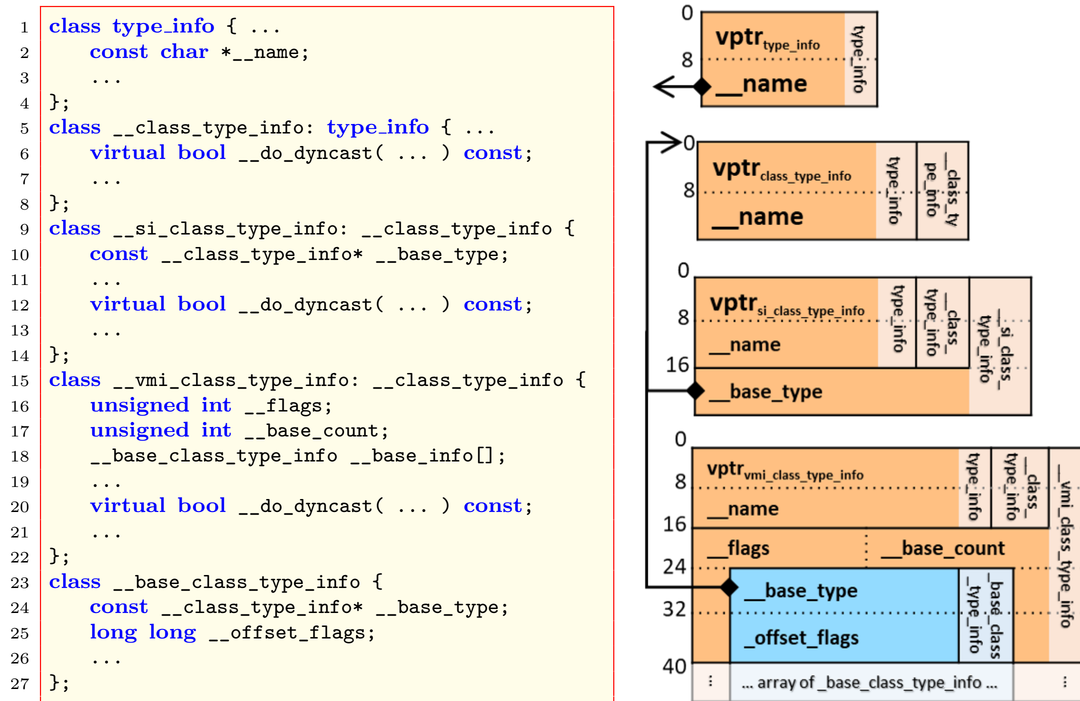
Dynamic Access Rights Figure 22 expands on the concepts of compile-time variable access rights (discussed in Section 2.3.5) of dynamic class instances. The figure depicts the access rights of each variable type used to address a run-time `Human` object. Since each run-time object is of `Human` type, their vptrs addresses the `Human` vtable, but it is the object's static type that determines the access rights to that vtable at compile-time. Object pointers with a base class type, such as `Humanoid`* and `Mortal`*, have access rights to a subgroup of virtual function pointers from the `Human` vtable; this subgroup reflects the functions available to the static type of the variable. For example, the `Humanoid` vtable, depicted in Figure 18, has two function pointer entries that address the `die()` and `speak()` functions. Similarly, the `Humanoid`* pointer, depicted in Figure 22, also has access to these same functions through the `Human` vtable, meaning that the pointer has access to the dynamic class versions of these virtual functions (if they were overridden).

Virtual Functions and Implicit Casts At the binary level, the implementation of a virtual function is no different to a non-virtual function; they are just called by different methods; non-virtual functions are called directly, whereas virtual functions are called indirectly via a vtable. So, Section 2.3.5's rules regarding member functions also hold true for virtual member functions. In particular, virtual member functions still receive an implicit object parameter, which must be

the same type as the function's defining class. However, thanks to how vtables are set up, only vtable entries linked to thunks perform implicit casts. For example, consider the `Avatar` object and vtable in Figure 20 again. The first set of virtual functions in the vtable (`&Avatar::speak()`, `&Navi::getNaviGenetics()`, and `&Avatar::remotelyControl()`) can only be invoked using the vptr ($\texttt{vptr}_{\texttt{A+24}}$), which, in turn, can only be used if we are addressing the object from the zero offset. The zero offset address-point can be one of two types, an `Avatar`* or a `Navi`*, matching the types that those three functions can use. Therefore, there is no need to perform an implicit cast. Now consider the other address-points, each with its own vptr and virtual function array. The functions called directly from the vtable do not require an implicit cast, as the address-point used to access them is already of the correct type. However, notice that the functions called via a thunk are not the same type as the address-point used to access them, hence the need to invoke a thunk and perform the implicit cast.

2.4.4 Run-Time Type Information

Run-time type information (RTTI) is a feature of the C++ language; it is an auxiliary data structure that encapsulates the type information of a class and is accessible to that class's instances during run-time execution. RTTI is defined within the C++ standard library and is simply another inheritance hierarchy structure. This hierarchy is used solely by the compiler, producing a unique RTTI object for every complete class (a class with a complete definition) that requires a vtable. The unique RTTI object is accessible from a class's vtable and stores information about that class, including its name, hierarchy type (single, multiple, or virtual), and a list of pointers to base class RTTI objects with accompanying offset information to relevant sub-object positions. The RTTI object itself is used as a type identification key, i.e. the physical address of an RTTI object is often used to identify an object's type at run-time, particularly in operations such as



```
class type_info { ...
    const char *__name;
    ...
};
class __class_type_info: type_info { ...
    virtual bool __do_dyncast( ... ) const;
    ...
};
class __si_class_type_info: __class_type_info {
    const __class_type_info* __base_type;
    ...
    virtual bool __do_dyncast( ... ) const;
    ...
};
class __vmi_class_type_info: __class_type_info {
    unsigned int __flags;
    unsigned int __base_count;
    __base_class_type_info __base_info[];
    ...
    virtual bool __do_dyncast( ... ) const;
    ...
};
class __base_class_type_info {
    const __class_type_info* __base_type;
    long long __offset_flags;
    ...
};
```


(a) Partial RTTI hierarchy (b) RTTI layouts

Figure 23: RTTI hierarchy and object layouts in libstdc++

`dynamic_cast` (Section 3.4.4), which only operates on complete class instances.

Different Implementations of RTTI The RTTI hierarchy and data fields are specified in the Itanium C++ ABI [23]; however, the ABI does not specify the virtual functions of RTTI classes, except for the destructor function. Therefore, it is essential to note that the RTTI classes discussed in this section are specific to GNU's C++ standard library implementation [44], which is called libstdc++. Other implementations, such as LLVM's libcxxabi [75], contain different virtual function signatures and definitions within its implementation of the RTTI hierarchy.

The Run-Time Type Information Hierarchy Figure 23a lists (partially) the source code of the RTTI hierarchy in libstdc++ [44] and each class's corresponding

Class	Size (bytes)	Class Use	Attributes	Attribute Use
type_info	16	Primary base class for all RTTI instances	**__name**	char* pointer to the mangled name of the type
__class_type_info	16	Linked to objects with no base classes	-	-
__si_class_type_info	24	Linked to objects with only a single, public, non-virtual base class	**__base_type**	A pointer to a (polymorphic) __class_type_info instance. It stores the address of another RTTI object, representing that class's base type
__vmi_class_type_info	24 + (16 for every base class)	Linked to objects with any other base class scenario.	**__flags**	Described the class structure using the __flags_masks enumeration. 0x01: class has non-diamond repeated inheritance 0x02: class is diamond-shaped
			__base_count	Number of direct base classes
			__base_info[]	Array of __base_class_type_info objects, once for every direct base class.
__base_class_type_info	16	Supporting class for __vmi_class_type_info, used to store base class information	**__base_type**	A pointer to a base class RTTI object
			__offset_flags	Stores offset to base class subobject or offsets to vtable entries. Uses two lowest bits as flag information about base class. 0x1 Base class is virtual 0x2 Base class is public

Table 4: RTTI classes and attribute uses

object layout in Figure 23b. In this partial hierarchy we have included only class attributes and a single virtual function definition called `__do_dyncast`, which is relevant to a later Chapter.

The primary class of this RTTI hierarchy (listed in Figure 23a) is called `type_info`, and every RTTI class inherits from it. As defined in the Itanium C++ ABI [23], the `type_info` class has ten derived classes (eight direct), which are used to represent all C++ data types for any ABI conforming program (full inheritance hierarchy in Figure 85 in Appendix A.1). One direct derived class from `type_info` is `__class_type_info`, which is an RTTI type that represents all programmer defined class types. From the `__class_type_info` class, there are two further directly derived classes, `__si_class_type_info` and `__vmi_class_type_info`, each used to represent classes with specific hierarchical scenarios, as described in Table 4. We will focus on these three RTTI classes as their instances represent all dynamic class types at run-time. An explanation of each RTTI class and the attributes

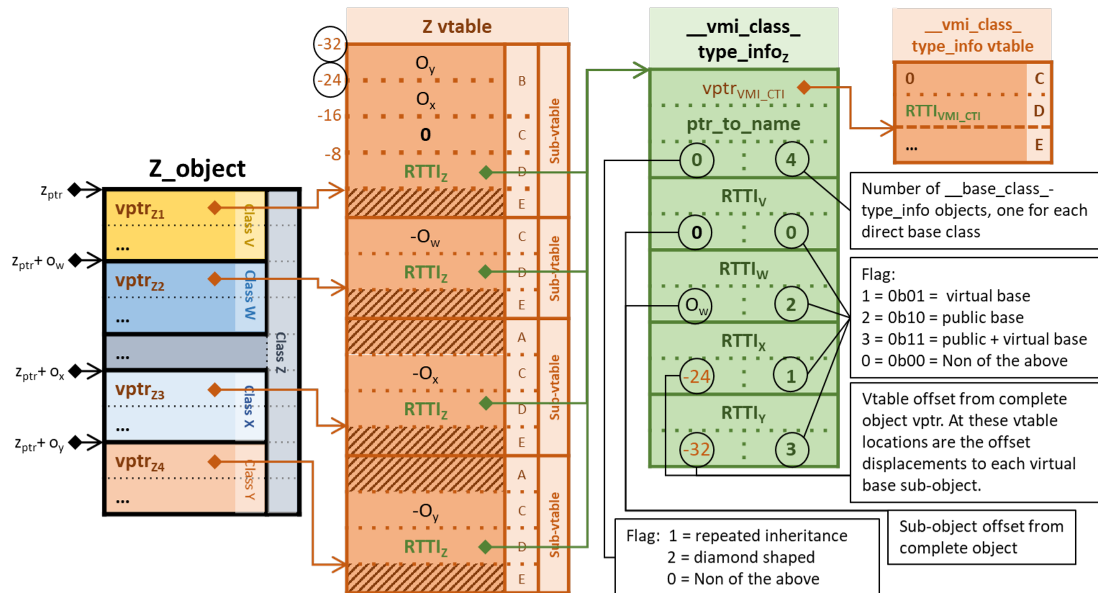


Figure 24: RTTI data members explained. See Figure 23 for data member names and Table 4 for uses.

they introduce (listed in Figure 23) is presented in Table 4.

The final class listed in Figure 23a is the `__base_class_type_info` class. The `__base_class_type_info` class is another primary class with no descendants but is used within an array attribute of the `__vmi_class_type_info` class. As an attribute, at least one complete instance of the `__base_class_type_info` class exists inside the complete `__vmi_class_type_info` object instance. This is known as a **compound object**, an object instance that exists within another but is not inherited. As it is not inherited, it is not a sub-object but a compound object.

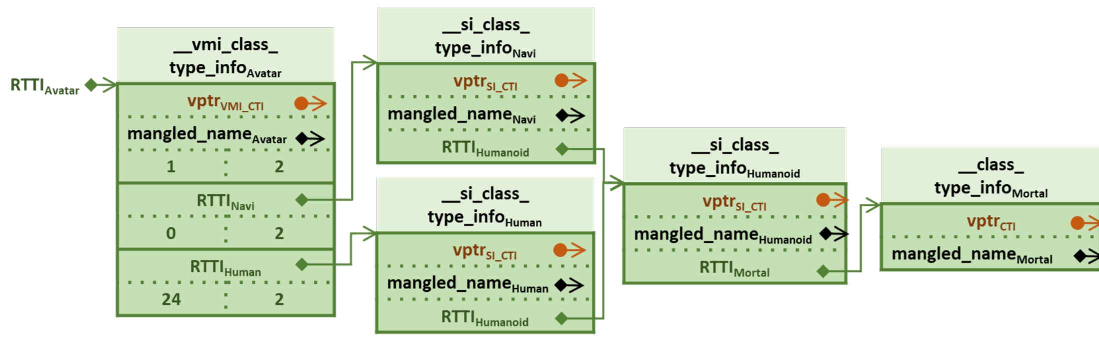
Before providing the RTTI object examples for our Avatar hierarchy, we will first look at a generic `__vmi_class_type_info` object instance to provide a clear picture of the purposes of each data member. Figure 24 displays a multiple inheritance object `Z` which inherits from classes `V` and `W`, as well as virtually inheriting from classes `X` and `Y`. This object has four address-points, each labelled as z_{ptr}+O_i where z_{ptr} is the address-point of the complete `Z` object, and +O_i is the offset displacement to the `i` sub-object. The `__vmi_class_type_info`

RTTI object generated for `Z` has four compound `__base_class_type_info` objects, one for each inherited class, storing a pointer to that class's RTTI object. Each data member of the `__vmi_class_type_info`$_z$ object is labelled[3] in the figure, but note that offset information for virtually inherited objects (`X` and `Y`) is not stored in their respective compound `__base_class_type_info` objects. Instead, the `__base_class_type_info` object stores an offset (`-24` or `-32`) to the object displacement values (`O`$_x$ or `O`$_y$) within the vtable. This is because the offset locations to virtually inherited sub-objects, unlike non-virtual sub-objects, are not consistent in all derived-class instances. By placing offset data in the vtable, the `__vmi_class_type_info`$_z$ object can represent the `Z` class in all derived class RTTI data structures, which keeps it unique to the `Z` class. If virtually inherited offsets were stored in RTTI objects, a new `__vmi_class_type_info`$_z$ object would have to be generated for every derived class of `Z` to compensate for the new offset locations of the virtually inherited sub-objects `X` and `Y`.

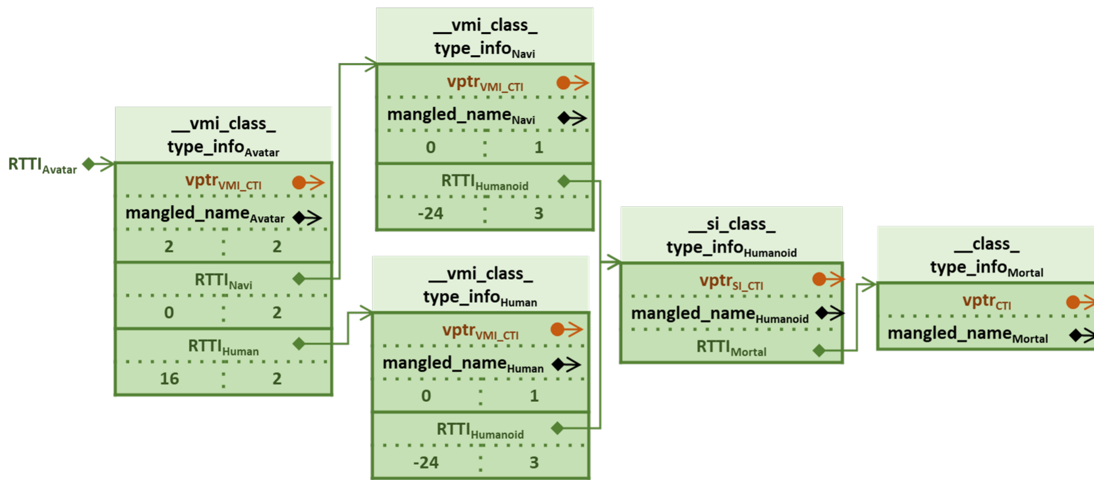
Figure 25 now presents the tree structure of linked RTTI objects that represent the `Avatar` hierarchy for both the virtual and non-virtual inheritance examples. Each object in this data structure is addressed from the vtables depicted in Figures 18, 19, 20, and 21.

Memory Overheads RTTI can add significant memory overheads to a program. A technical report on C++ performance [47] estimated that a typical RTTI object adds 40 bytes of data per class. For our `Avatar` hierarchy, RTTI representation adds a total of 272 bytes for the virtual inheritance hierarchy in Figure 25b (176 for the RTTI objects, plus 96 for mangled class names and alignment padding) and 240 bytes for the non-virtual hierarchy in Figure 25a

[3] Note that the flag and offsets values stored in the `__base_class_type_info` objects is actually a single data member, not two. The data member, called `__offset_flags` (Figure 23), uses the last two bits of the number stored as a flag and masks and shifts these bits to retrieve the offset value.



(a) Without virtual inheritance



(b) With virtual inheritance

Figure 25: All RTTI objects create the `Avatar` hierarchy

(144 for the RTTI objects, plus 96 bytes for mangled class names and alignment padding), averaging 51 bytes per class overall. Of course, the number of bytes occupied by RTTI depends on the hierarchy itself. In our example, the `Avatar` hierarchy has either multiple or virtual inheritance, which is represented using the `__vmi_class_type_info` class instances, the largest of all RTTI objects (see Table 4). Simple single inheritance hierarchies represented using `__si_class_type_info` objects will occupy less space. Additionally, mangled class names are stored as part of RTTI, which incurs some alignment padding in some cases, increasing the overall memory overheads of RTTI support.

Modern machines have plenty of memory, so an average cost of 51 bytes per class for RTTI data may appear reasonable; after all, most classes and their hierarchies will not be that complicated. However, one must consider class templates; as we saw in Section 2.2.5, template instantiation can easily multiply the number of classes after compilation. Where a program leans heavily on class templates, template instantiation could result in a significantly large number of class specialisations after compilation. This could be problematic, especially for small systems like embedded systems and mobile devices that have significant restrictions on their memory capacity. For these reasons, it is not uncommon for developers to turn off RTTI generation during compilation to avoid this cost. The repercussion of removing RTTI is that programmers lose RTTI-dependent C++ features, like dynamic casting, which we will discuss in Section 3.4.4.

2.5 MSVC Object Comparison

We will now take a brief look at the objects produced by the MSVC compiler on an AMD64 Windows OS platform. The MSVC compiler has several different strategies for object layouts and their supporting auxiliary data structures compared to the g++ compiler on a System V OS. We can demonstrate these differences using the same virtual inheritance hierarchy from earlier examples. Figure 26 depicts two Avatar objects produced by the MSVC compiler. The first is from a virtual inheritance hierarchy with no virtual functions, and the second is from a virtual inheritance hierarchy with a virtual function present in every class. The following list describes the key differences between MSVC (using using the Microsoft C++ ABI) and g++ (using the Itanium C++ ABI) objects and virtual tables:

More padding: MSVC has stricter alignment requirements than g++ resulting in larger objects from multiple and virtual inheritance hierarchies.

Virtual base table pointers: Objects contain virtual base table pointers (vbptr)

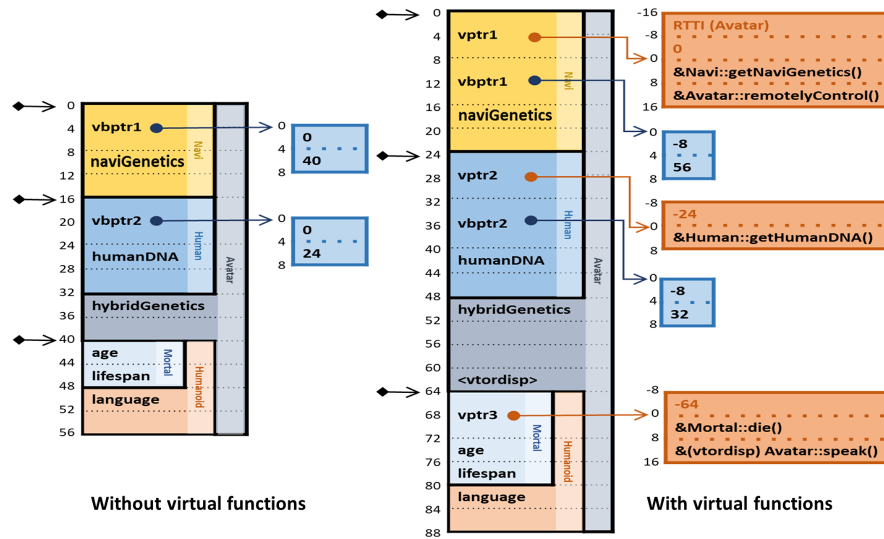


Figure 26: Virtual inheritance object layout for MSVC

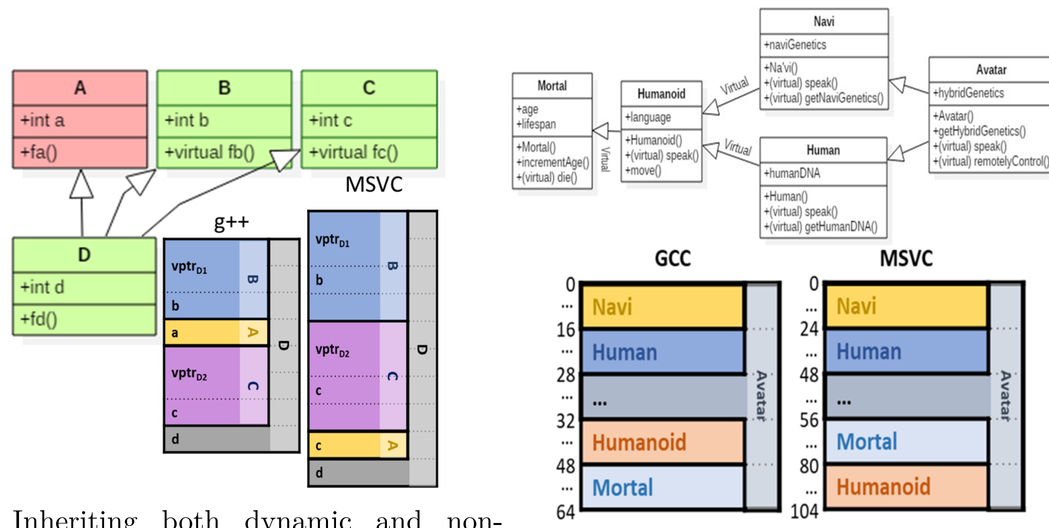
used to address virtual base displacement tables.

Virtual base displacement table: MSVC does not store displacement information to virtual base sub-objects in its vtables. These values are instead stored in a separate table, called a virtual base displacement table. The virtual base displacement table contains a displacement value to the address-point of the current (sub-)object (use to address the table) and an array of virtual base displacements (offsets to virtual base sub-objects).

Different vtable layout: The RTTI pointer and displacement value to the complete object appear in a different order than seen in g++ vtables.

Only one RTTI pointer: Sub-vtables addressed by sub-objects do not have an RTTI pointer. Instead, only one RTTI pointer exists in a complete vtable, and it is addressable only from the complete object's vptr.

Virtual function pointer array: Sub-vtables only store virtual function pointers declared within the object types that address them. For example, the function `speak()` was declared in the `Humanoid` class, so it exists only in the sub-vtable address by the `Humanoid` sub-object. The version of this function still correlates with the most derived-type and hence addresses the



(a) Inheriting both dynamic and non-dynamic classes

(b) Multiple virtually inherited classes

Figure 27: Differences in sub-object ordering

`Avatar::speak()` implementation.

New vtordisp data member: Sub-vtables addressed by virtually inherited sub-objects (like `Humanoid`) do not contain displacement information for their virtual functions. Instead, MSVC uses a data member called vtordisp to aid in implicit casting for these virtual functions. The vtordisp data member is stored just above the relevant virtual sub-object.

MSVC also applies different methods for sub-object ordering, which can be seen in Figure 27. In Figure 27a the `D` class inherits from `A`, `B`, and `C` in that order, where `B` and `C` are dynamic. As seen earlier, g++ will place the first dynamic class within the base-specifier-list as the first sub-object. MSVC, on the other hand, place all dynamic sub-objects first, then follows with non-dynamic sub-objects.

Figure 27b presents an example of multiple virtually inherited classes. In this case, we use the same `Avatar` hierarchy but virtually inherit both the `Mortal` and `Humanoid` classes. If the hierarchy is thought of as a tree graph, g++ has a bottom-up approach to virtually inherited sub-object placement, whereas MSVC has a top-down approach.

2.6 Concluding Discussion

This chapter looked closely at the object layouts produced from the Itanium C++ ABI on an AMD64 System V platform. This overview saw object layouts from single, multiple, and virtual inheritance hierarchies. We also looked at dynamic classes and saw virtual function pointers' impact on the layouts of dynamic objects. We also discussed the layouts of vtables and RTTI structures and briefly touched on some mechanisms that interact with these data structures.

To complete this chapter, we briefly discussed the object layouts produced by the MSVC compiler on an AMD64 Windows platform. This section provided a more rounded picture of the different strategies employed by major compiler vendors and how two compilers that conform to the C++ standard can produce different object layouts in memory.

Understanding these concepts will be beneficial for discussing more advanced topics in Part 2 of this thesis.

Chapter 3

Assembly-Level Object Operations

We now know the fundamental ideas behind object layout, vtable layout and RTTI data structures. But, what about the member functions and operations that interact with them? We will now look at how member functions, constructors and cast operations interact with different object instances at the assembly level.

3.1 Introduction

This chapter continues from the last, expanding the examples of object memory layouts to the low-level operations that construct and interact with them. We will explore how member functions, constructor functions, and cast operations are realised in assembly and how they interact with object instances and their supporting auxiliary data structures. We follow the same method in explaining these concepts as previous chapters, using examples. All examples were produced by the GNU C++ Compiler (g++) using libstdc++ (GNU's Standard C++ library implementation [44]).

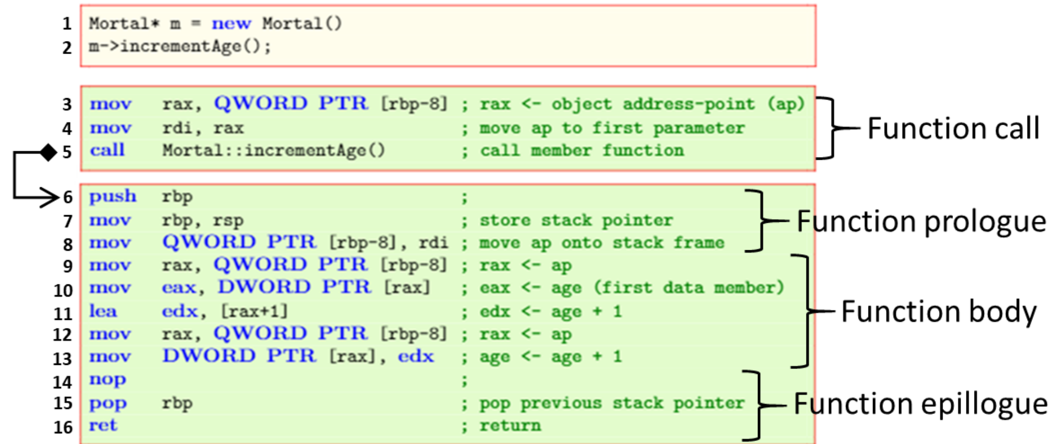


Figure 28: `Mortal::incrementAge()` member function call

3.2 Member Functions

How member functions interact with objects has already been discussed in Sections 2.3.5 and 2.4.3, but we will now look at what is happening at the assembly level.

3.2.1 Function Bodies

Any function defined in a class is a member function; this includes constructors and destructors. At the assembly-level, member functions are realised in three parts, a prologue, a function body, and an epilogue. The prologue allocates and initialises the stack frame, the function body reflects the function definition in the source code, and the epilogue releases the stack frame and returns control flow to the call site. An example of these three parts can be seen in Figure 28, which shows the assembly of the `Mortal::incrementAge()` function.

3.2.2 Non-Virtual Member Functions Calls

Non-virtual functions are realised as direct function calls. Figure 28, lists the non-virtual call of the `Mortal::incrementAge()` function in line 2. This call is realised in the assembly in lines 3-5. We know from Section 2.3.5 that all member

functions receive an object address as their first parameter; this can be seen in lines 3 and 4 of our function call example. Specifically, in line 3, an address is copied from the stack before being placed into the first parameter register in line 4. As the `Mortal::incrementAge()` function has no other parameters (other than the object itself), no more parameters are staged, and the function is called directly in line 5.

The function body of `Mortal::incrementAge()` is simple, containing only one line of source code (`age++`) that increments the `Mortal` attribute `age` (the first data member in a `Mortal` object). Line 6-16 lists the assembly code of this function. The function's prologue resides in lines 6-8 and assigns a stack frame before placing the object's address in that frame in line 8. The body of the function exists between lines 9-13. The body goes through a process of retrieving the object address from the stack (line 9), accessing the first data member (line 10), incrementing its value (line 11), then writing it back to the object (lines 12 and 13). The function concludes with an epilogue (lines 14-16) which resets the stack (line 15) before returning control to the call location (line 16).

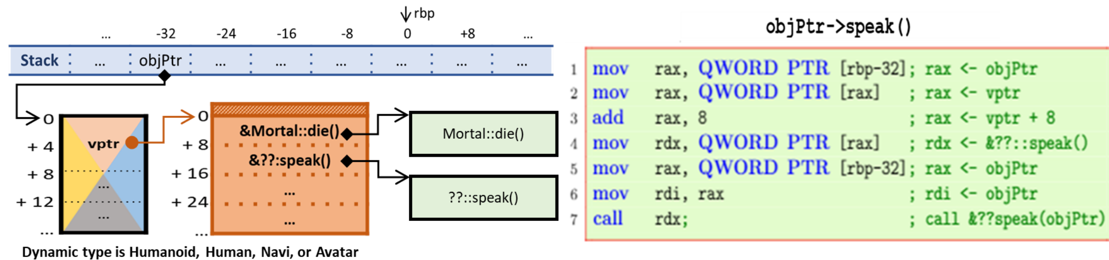
3.2.3 Virtual Member Function Calls

Virtual functions are realised as indirect function calls, which are performed using a mechanism called dynamic dispatch. **Dynamic dispatch** is a low-level mechanism that will call a virtual function based on the run-time type of the invoking object. This mechanism is supported by the vtables, which house the virtual function pointers of a variable's dynamic type and is accessible through the invoking object's vptr.

Dynamic Dispatch Mechanism Figure 29 provides a visual example of the dynamic dispatch mechanism, where Figure 29a presents the source code, Figure 29b the object layout in memory, and Figure 29c the assembly output. In

```
1 Humanoid *objPtr; //dynamic type could be Humanoid, Human, Navi, or Avatar
2 ...
3 objPtr->speak();
```

(a) Dynamic dispatch as the source level



(b) Memory representation (c) Dynamic dispatch in assembly

Figure 29: The dynamic dispatch mechanism

line 1 of the assembly, the object address-point is retrieved from the stack (Figure 29b). Using this address, the vptr is accessed (line 2) and adjusted by +8 (line 3). The adjusted vptr now points to the second virtual function pointer in the vtable. This pointer is retrieved (line 4) and called (line 7) with the object variable (`objPtr`) passed as a first parameter (lines 5-6).

The dynamic dispatch listed in Figure 29 will always call the virtual function in the second entry of the vtable. So, which function is invoked depends on which vtable is accessed. Which vtable is accessed depends on the run-time type of the object used. In this example, the object is either a `Humanoid`, `Human`, `Navi`, or `Avatar` type and will invoke either `Humanoid::speak()`, `Human::speak()`, `Navi::speak()` or `Avatar::speak()` respectively (see their vtables in Figures 18, 19, 20, and 21).

If the run-time object is part of a multiple or virtual inheritance hierarchy, then the second vtable entry from the vptr may not directly address the `speak()` function but instead address a thunk [52] (as seen in Figure 19 and 20). Recall that a thunk (also known as a trampoline) is a small snippet of instruction code that performs an operation before jumping to another set of instructions. In this case, it performs an implicit cast to the complete object before invoking the

```
1 mov   edi, 16            ; size of Mortal object
2 call  operator new       ; new operator call
3 mov   rbx, rax           ;
4 mov   rdi, rbx           ;
5 call  Mortal::Mortal()   ; Mortal constructor call
```

Mortal *obj = new Mortal();

```
6  push  rbp                          ; \
7  mov   rbp, rsp                     ;  |- prologue
8  mov   QWORD PTR [rbp-8], rdi       ; /
9  mov   edx, vtable for Mortal+16    ; vtable address
10 mov   rax, QWORD PTR [rbp-8]       ;
11 mov   QWORD PTR [rax], rdx         ; vptr assigned to zero offset
12       ...                          ; constructor body
13 nop                                ; \
14 pop   rbp                          ;  |- epilogue
15 ret                                ; /
```

Figure 30: `Mortal` constructor call

appropriate version of the `speak()` function. This operation uses a displacement value also stored in the vtable to find the complete object address-point. These displacement values can be seen in Figure 20.

3.3 Object Construction

Objects are initialised using a class constructor. A constructor function is responsible for initialising all data members, including all vptrs if the objects are dynamic. Constructors from primary classes, derived classes, and virtual inheritance classes, are realised slightly differently in each case. So this section will explore each case separately.

3.3.1 Primary Class Construction

The most straightforward constructor functions are generated from primary classes, as they need only initialise their own data members. For example, the following expression:

```
Mortal *obj = new Mortal();
```

invokes the constructor of the primary class `Mortal`, which generates the assembly in Figure 30. The `Mortal` class has a predetermined size requirement for its instances, 16 bytes in this case. The value, 16, is stored in the first parameter register (`rdi` in line 1), before calling the `new` operator in line 2. Given a class's size requirements, the `new` operator is responsible for finding and allocating a region of memory of that size. The address of the assigned memory region is returned

via the `rax` register in line 3. The returned address is then passed to the first parameter register in line 4, ready for the invocation of the `Mortal` constructor in line 5. The address passed between the `new` operator and the `Mortal` constructor is the address-point of the `Mortal` object and will eventually be stored in the `obj` variable. The `obj` variable itself is simply a stack location that stores the pointer to the constructed object.

The `Mortal` constructor, like all constructors, is a member function, so it contains a prologue and an epilogue (Figure 30 lines 6-8 and 13-15). After the constructor's prologue, the vtable address is stored in register `edx` (line 9) and then assigned as the first data member of the object (line 11), initialising the vptr. The vptr is the first data member initialised in any dynamic object. After the vptr is initialised, the body of the `Mortal` constructor is executed. From this point onwards, all function prologue and epilogue will be omitted from assembly code listings, to simplify examples.

3.3.2 Derived Class Construction

A derived class constructor is responsible for initialising not only its own data members but also its immediate sub-objects. To do this, the derived class constructor will invoke the constructors of its immediate base classes, which in turn will do the same, until a primary class constructor is reached; this creates a set of nested constructor calls. For example, the following expression:

```
Human *obj = new Human();
```

will generate the assembly in Figure 31. Like in the previous example, the new operator is called (line 2), but this time has a memory size request of 24 bytes (line 1). Upon its return (line 3), the memory address retrieved from the new operator is passed to the `Human` constructor (line 4-5).

The first action a derived class constructor will perform is to invoke its base class constructors. We can see this in line 10 of the `Human` constructor and line 20

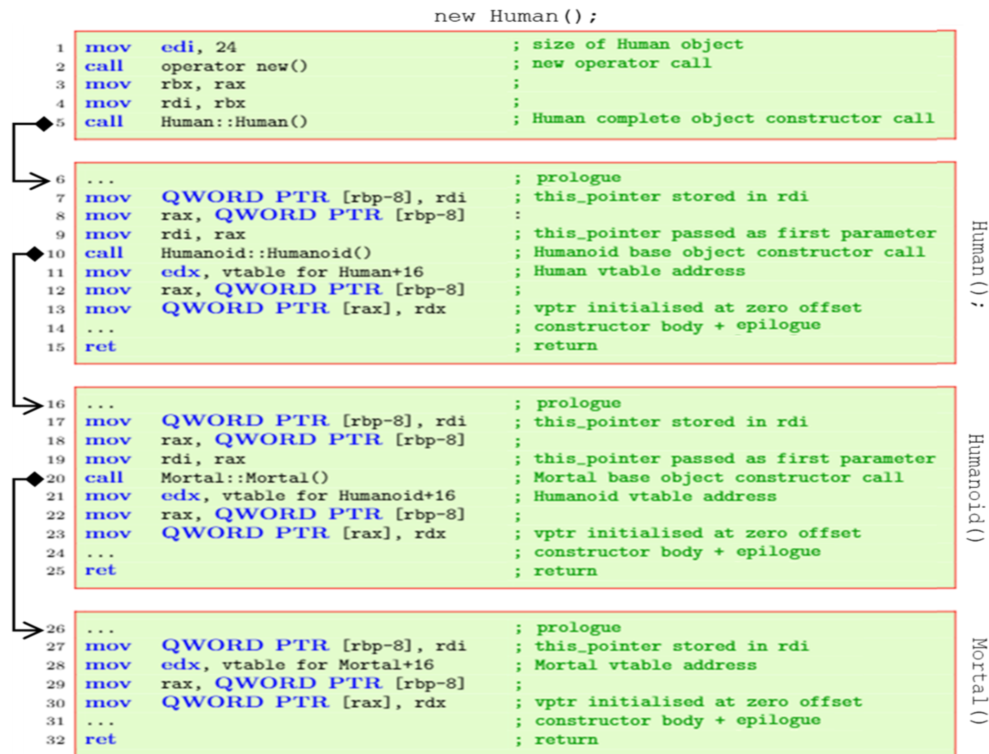


Figure 31: `Human` nested constructor call

of the `Humanoid` constructor. Once all base class sub-object instances are constructed, vptrs are assigned (lines 11-13, 21-23, and 28-30). Note that each base class constructor will assign its own vptr, which will be immediately overwritten upon return of the constructor call. Following vptr assignment is the constructor body, reflecting the programmer's implementation at the source level (lines 14, 24, and 31). Once the constructor body has been executed, control flow is returned to the callee: here, lines 32 and 25 return control flow to derived class constructors (line 32→21 and line 25→11), whereas line 15, the last return of the nested call, completes the construction of the `Human` object.

3.3.3 Virtual Inheritance Class Construction

Any virtually inherited class is guaranteed to appear once, and only once, as a sub-object in any inheriting class instance. This guarantee is achieved with a slight

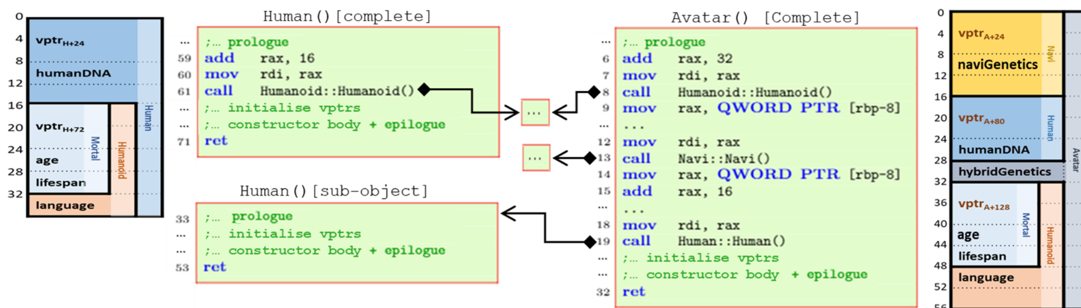
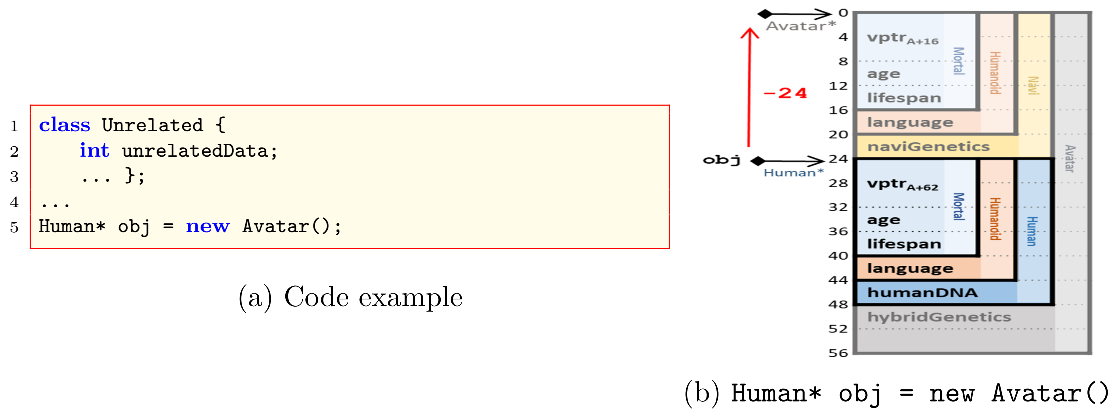


Figure 32: `Avatar` nested constructor call with virtual inheritance (full assembly in Figure 86 Appendix A.2)

change to the nested constructor calls seen so far. When constructing an object from a virtual inheritance hierarchy, the responsibility of calling the virtually inherited class constructor no longer falls to the class that directly inherited it but instead the most-derived class. By passing this responsibility, the virtually inherited class constructor will be invoked directly within the most-derived class constructor but not repeated in any nested calls; resulting in a single sub-object instance within the complete object.

Figure 32 depicts the construction of a complete `Avatar` and `Human` object. Each class virtually inherits from `Humanoid`, where `Human` inherits directly and `Avatar` indirectly. As `Humanoid` is virtually inherited, it is the responsibility of both the complete object constructors (`Human()` and `Avatar()`) to invoke the `Humanoid` constructor (lines 8 and 61). The `Avatar` object contains an instance of the `Human` class as a sub-object. When `Human` appears as a sub-object, it does not contain the virtually inherited `Humanoid` base class instance. However, when `Human` appears as a complete-object (most-derived), it does contain a virtually inherited `Humanoid` sub-object. To accommodate both scenarios, the `Human` class generates two constructor[1] functions, one for complete object construction

[1]This is Itanium ABI specific. MSVC produces a single constructor with an internal 'most-derived class' check to determine which constructors will invoke the virtually inherited constructor(s).




```
1 class Unrelated {
2     int unrelatedData;
3     ... };
4 ...
5 Human* obj = new Avatar();
```

(a) Code example

(b) `Human* obj = new Avatar()`

Figure 33: Cast Example

(lines 59-71), which includes virtual sub-objects, and one for sub-object construction (lines 33-53), which omits virtual sub-objects.

3.4 Cast Operations

We briefly discussed explicit casting in Section 2.3.4 and implicit casting in Section 2.3.5. In this section we discuss the four types of explicit cast operators - C-style cast, `static_cast`, `reinterpret_cast`, and `dynamic_cast`, and how they are realised in assembly.

For all cast examples, we will use an `Avatar` object addressed by a `Human`* variable, called `obj` (Figure 33a line 5). This `obj` variable stores the address-point at offset `+24` within the complete `Avatar` object, as seen in Figure 33b. In all examples, two casts will be performed on the `obj`, followed by a data member assignment. The first cast will be to an `Avatar`* type, which should adjust the pointer stored in `obj` by `-24`, to retrieve the complete object address-point. The second will be an illegal cast to an `Unrelated` type (Figure 33a line 1) to help demonstrate the vulnerabilities of different cast operations.

```
1 ((Avatar*)obj)->hybridGenetics = 0;
```

```
2 mov    rax, QWORD PTR [rbp-24]
3 sub    rax, 24
4 mov    DWORD PTR [rax+48], 0
```

(a) Cast to `Avatar`*

```
5 ((Unrelated*)obj)->unrelatedData = -1;
```

```
6 mov    rax, QWORD PTR [rbp-24]
7 mov    DWORD PTR [rax], -1
```

(b) Cast to `Unrelated`*

Figure 34: C-style Casting

3.4.1 C-style Cast

The C-style cast was inherited from the C language; it is a compile-time operation with the following expression:

`(target)variable`

A C-style cast is performed statically, meaning the compiler can infer the result of a cast at compile-time. If the result of a cast is known at compile-time, it means that the pointer adjustment for the cast is fixed and known and can therefore be realised directly in the assembly. This can be seen in the following example.

A C-style cast is performed in Figure 34a line 1. At the source level, the `obj` variable is temporarily cast to an `Avatar`* type, granting it access to the `Avatar::hybridGenetics` data member before setting it to 0. Lines 2-4 is the assembly generated from that cast and data member assignment. Line 2 retrieves the `obj` pointer from the stack (positioned at `rbp-24`) and moves it to the `rax` register. Line 3 subtracts `24` (the known adjustment) from the pointer stored in rax, creating a new pointer that addresses the zero offset of the `Avatar` object; this line completes the temporary cast. Line 4 adjusts the pointer in `rax`, to address the `hybridGenetic` data member at offset `+48` and moves the zero value into that location, setting that data member to zero. After the assignment, the cast pointer is dropped, and the original `object` pointer stored on the stack goes unchanged. This is why we describe this as a temporary cast, as it does not alter the address stored in the variable `obj` itself but uses it to return the address-point of the cast's target.

```
1 static_cast<Avatar*>(obj)->hybridGenetics = 0;

2 mov   rax, QWORD PTR [rbp-24]
3 sub   rax, 24
4 mov   DWORD PTR [rax+48], 0
```

(a) Cast to `Avatar*`

```
5 static_cast<Unrelated*>(obj)->unrelatedData =
  -1; // compile time error
```

(b) Cast to `Unrelated*`

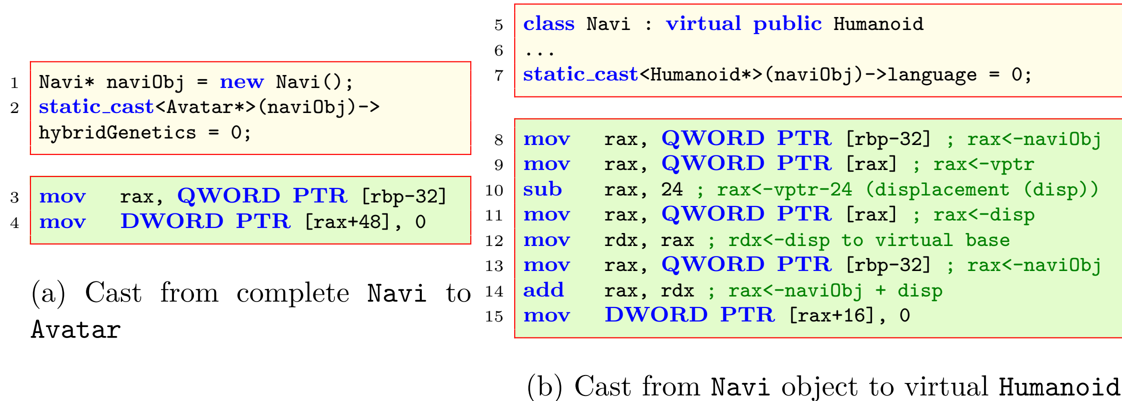
Figure 35: static_cast

Figure 34b line 5 attempts a similar cast but targets an `Unrelated` type. The C-style cast allows such code to be compiled, resulting in the assembly in lines 6 and 7. As these types are unrelated, the compiler has no hierarchical information and applies no adjustments to the pointer stored in `obj`. Instead, the pointer is assumed to be the `Unrelated` type, and execution continues as if that is the case. This is known as a **type confusion vulnerability** (discussed further in Section 5.1). Type confusion vulnerabilities are problematic, as we can see in line 7 of our assembly. At the assembly level, the object address is assumed to be an `UnrelatedData` object and therefore allows the first data member of this object to be overwritten and assigned the value `-1`. However, what has actually happened in this type confusion is that the `vptr` of the `Human` sub-object (within the complete `Avatar` object) has been overwritten to `-1`, which would likely result in a segmentation fault if a virtual function is invoked.

3.4.2 `static_cast<target>(variable)`

The `static_cast` operator, like C-style, is a compile-time operation, but unlike C-style, it benefits from static type-checking. When using this type of cast, the compiler will ensure that the source and target types are part of the same hierarchy before generating the assembly code.

Figure 35a shows that the binaries generated from the `static_cast` operator is identical to the C-style cast in Figure 34a. However, in Figure 35b, when a cast is attempted with an unrelated target type, the `static_cast` operator will



(a) Cast from complete `Navi` to `Avatar`

(b) Cast from `Navi` object to virtual `Humanoid`

Figure 36: Type confusion using static_cast & casting to virtual base

prompt a static type relationship check, resulting in a compile-time error.

Despite static type-checking, `static_cast` can still suffer from type confusion vulnerabilities. In Figure 36a line 1, we create a complete `Navi` object stored in the variable `naviObj`, before casting it to an `Avatar` object in line 2. As static casting only ensures these types are related at compile-time (which they are), the dynamic type of `naviObj` is not considered, and the cast is allowed, resulting in a type confusion vulnerability. In this scenario, the compiler assumes that `naviObj` addresses a `Navi` sub-object within an `Avatar` object; as the `Navi` sub-object exists at a zero offset in a complete `Avatar` object, the generated assembly makes no alterations to the pointer stored in `naviObj`. The following instruction (line 4) reassigns a data member at offset `+48` from the pointer in `naviObj`. As the `Navi` object layout is only 24 bytes in size, the cast actually allows data to be altered outside of the object's boundaries, potentially causing a data corruption error.

In Figure 36b line 5, we have temporarily altered the `Navi` class to virtually inherit from `Humanoid`, then performed a static cast to that `Humanoid` type from the complete `Navi` object (line 7). This is the only type of `static_cast` that is performed dynamically, as it is encoded to access the object's vtable to retrieve the displacement offset to the `Humanoid` sub-object (line 10 & 14). This displacement offset is positioned at `-24` from the `vptr`, as seen in line 10 and the `Navi`

```
1 reinterpret_cast<Avatar*>(obj)->
  hybridGenetics = 0;

2 mov   rax, QWORD PTR [rbp-24]
3 mov   DWORD PTR [rax+48], 0
```

(a) Cast to `Avatar*`

```
4 reinterpret_cast<Unrelated*>(obj)->
  unrelatedData = -1;

5 mov   rax, QWORD PTR [rbp-24]
6 mov   DWORD PTR [rax], -1
```

(b) Cast to `Unrelated*`

Figure 37: reinterpret_cast

vtables depicted in Figure 20 and 21. We should note that this cast is performed dynamically but does not perform any type checks, so it can still cause a type confusion vulnerability.

3.4.3 `reinterpret_cast<target>(variable)`

The reinterpret cast operator returns a copy of a variable with a reassigned type, which occurs without any type-checking or offset adjustments. It is a direct transformation of types irrespective of hierarchical information and is ignored by all compiler static type-checking mechanisms. Figure 37 demonstrates that the `reinterpret_cast` operator generates no additional assembly instructions but simply reinterprets a pointer as the target type.

3.4.4 `dynamic_cast<target>(variable)`

The `dynamic_cast` operator is a run-time mechanism that can verify the dynamic type of a variable before casting. This additional check makes `dynamic_cast` the most secure casting technique and will avoid any potential type confusion vulnerabilities.

Figure 38a depicts how the dynamic cast operator is realised in assembly. Unlike earlier casting techniques that are composed of a straightforward offset adjustment to an object's pointer, dynamic casting is resolved using a separate (global) function called `__dynamic_cast` (line 7). The `__dynamic_cast` function takes the object address (lines 2 and 6) and the type information (RTTI objects)

```
1  dynamic_cast<Avatar*>(obj)->hybridGenetics
    = 0;
```

```
2  mov    rax, QWORD PTR [rbp-24]
3  mov    ecx, 24 ; static hint
4  mov    edx, typeinfo for Avatar
5  mov    esi, typeinfo for Human
6  mov    rdi, rax
7  call   __dynamic_cast
8  mov    DWORD PTR [rax+48], 0
```

(a) Cast to `Avatar*`

```
9  dynamic_cast<Unrelated*>(obj)->unrelatedData
    = -1; // run-time error
10 dynamic_cast<Avatar*>(naviObj)->
   hybridGenetics = 0; // run-time error
```

(b) Run-time errors for illegal casts

Figure 38: dynamic_cast

of the target type (line 4), and the object's static type (line 5), as parameters. The `__dynamic_cast` function will check the object's type using its RTTI information, and the appropriate cast will be performed. The mechanics of the `__dynamic_cast` function are discussed in the following section.

Figure 38b attempts a cast using an `Unrelated` type (line 9) or a complete base class type (line 10). In both cases, these casts will fail, resulting in either a null pointer return or throwing a `bad_cast` exception.

The `__dynamic_cast` function The global `__dynamic_cast` function is listed in Figure 39a. It takes four parameters: the source object address (`src_ptr`), the source's type information (`src_type` an RTTI pointer), the target type information (`dst_type` another RTTI pointer), and a number representing a static hint (`src2dst`). This function signature is defined within the Itanium C++ ABI [23], but the function body implementation is not. Different implementations of the standard C++ library will have different implementations of the `__dynamic_cast` function, handling dynamic casting slightly differently in each case. Here we examine the GNU's Standard C++ library implementation [44] (libstdc++).

In its most expensive form, dynamic casting is a recursive traversal through an entire object's RTTI data structure. As this can be expensive, the g++ compiler will attempt to bypass this traversal by providing the `__dynamic_cast` function

```
1  extern "C" void * __dynamic_cast (
2     const void *src_ptr,                      // object address-point
3     const __class_type_info *src_type,        // source RTTI object
4     const __class_type_info *dst_type,        // target RTTI Object
5     ptrdiff_t src2dst)                        // static hint
6  {
7    ...                                        // gather data from src_ptr vtable
8    if (src2dst >= 0 && src2dest == -whole_disp  // test for faster cast to avoid __do_dyncast
9       && *whole_type == *dst_type )           // if complete object is the target type
10     { return whole_ptr }                    // return complete-object address-point
11   whole_type->__do_dyncast (...);            // whole_type = RTTI object from src vtable
12   ...                                        // Check __do_dyncast return and complete cast
13 }
```

(a) Global `__dynamic_cast` function

```
14 bool __class_type_info::__do_dyncast (...) const { ...
15          /* no recursive call */ ... }
16 bool __si_class_type_info::__do_dyncast (...) const { ...
17          __base_type->__do_dyncast(...); ... }
18 bool __vmi_class_type_info::__do_dyncast (...) const { ...
19          __base_info[i].__base_type->__do_dyncast(... ); ... }
```

(b) The overridden `__do_dyncast` function

Figure 39: The global `__dynamic_cast` function and its recursive call of the RTTI virtual function `__do_dyncast`

with a static hint (`src2dst`) parameter. The `src2dst` parameter can take several different values: anything greater or equal to zero is a predicted displacement from the source address-point to the target address-point; whereas anything less than zero is a special case used to describe the static relationship between the source and target types. These special cases are as follows (as determined by the GNU library implementation [44]; other implementations may differ):

-1: no hint

-2: source type is not a public base of the target type

-3: source type is a multiple public base type but never a virtual base type

These values can help to speed up the `dynamic_cast` mechanism in certain circumstances by traversing shorter control-flow paths within the `__dynamic_cast` function call.

The global `__dynamic_cast` function is listed in Figure 39a. This function begins (line 7) by gathering data about the source object from its vtable (Figure 17).

In particular, the data collected includes the pointer to RTTI information (stored in `whole_type`), the displacement to the complete object (stored in `whole_disp`), and from this, the address-point to the complete object is calculated and stored in `whole_ptr`. In lines 8-10, the `__dynamic_cast` has a conditional if statement that, if true, will avoid the costly recursive call in line 11 (the `__do_dyncast` function). The conditional if statement is testing for a simple cast case, where the most derived-type of the source object is the target type of the cast. It does this with three conditional statements:

1. Checks if `src2dst` is greater or equal to zero. When true, it means that the `src2dst` stores a predicted displacement from the source's address-point to the target address-point.
2. Checks if `src2dst` is equal to the negation of `whole_disp`. If true, then the displacement hint matches the displacement to the complete object described in the vtable.
3. Checks if `*whole_type` is equal to `*dst_type`. If true, then the target type is the same as the source object's dynamic type.

When all three conditions are true, then the cast is attempting to cast the source object to its most derived-type. This type of cast is straightforward and does not need RTTI recursion, so instead, the calculated `whole_ptr` is returned in line 10. However, if just one condition is false, that RTTI will be traversed, which is achieved using the recursive `__do_dyncast` function.

The `__do_dyncast` function is invoked in line 11 of Figure 39a using the source object's RTTI pointer (stored in `whole_type`). This function is a virtual function, and has a different implementation depending on the dynamic type of the RTTI object being used. Figure 39b lists all three implementations of this function for each `__class_type_info` specialisation. We have redacted the majority of the functions' code as we are only focused on the recursive nature of each implementation. The implementation used at run-time depends on the dynamic type of the RTTI

object addressed by `whole_type`. If this RTTI object is a `__si_class_type_info` type (representing single inheritance) or `__vmi_class_type_info` type (representing multiple or virtual inheritance), then the `__do_dyncast` implementation will be recursive. Each call to the `__do_dyncast` function checks its `this-pointer` (the current RTTI object in the traversal) against the target RTTI pointer. If a match is found, then a cast is performed. If no match is found, `__do_dyncast` will continue its traversal until a match is found or it has exhausted all RTTI objects. Notice that in the case of `__vmi_class_type_info`, the recursive `__do_dyncast` function is performed on an array of `__base_class_type_info` structures, which represent multiple/virtual inheritance (see RTTI data layouts in Section 2.4.4).

Different Implementations We reiterate that the `__dynamic_cast` function is entirely implementation dependent. For example, in LLVM's libcxxabi standard C++ library [75], the `__dynamic_cast` function traverses the RTTI data structures using two functions (called `search_below_dst` and `search_above_dst` see 'private_typeinfo.cpp' in [75]), compared to GNUs one function (`__do_dyncast`). Another significant difference in implementations is that LLVM's `__dynamic_cast` function does not (at the time of writing) utilise the `src2dst` parameter for optimisation. So it is essential to acknowledge that the dynamic casting cost will differ in different implementations of the C++ standard library.

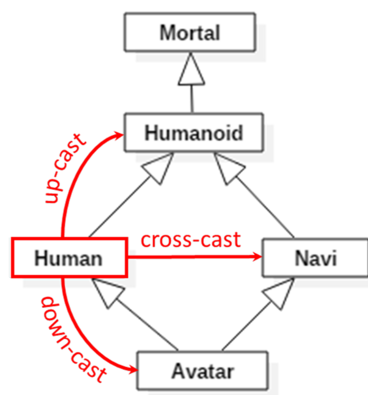
Overheads Because of the recursive nature of RTTI, the executional cost of performing a `dynamic_cast` will be significantly more expensive than a `static_cast`. A `static_cast`, at most, is a simple pointer adjustment embedded in the assembly. In contrast, a `dynamic_cast` can be multiple indirect function calls and comparisons before making a pointer adjustment. In the technical report on C++ performance [47], dynamic casting was estimated to be between 60 and 200 times more expensive than static casting, depending on the compiler and optimisation

level used. We, however, performed our own analysis of the `dynamic_cast` operator (Section 5.4.1) using the GNU C++ standard library and found that dynamic casting is at least 18 times more expensive than static casting, with no measurable upper bound.

The expense of RTTI, both in performance and memory space, makes it undesirable to many developers. To avoid this, some developers omit RTTI from their final binaries, which can be achieved with a single compiler flag. However, when RTTI is omitted, dynamic casting is unavailable, so developers are forced to use less secure casting methods, like static or C-style casting. When deployed without due diligence, insecure casting methods can result in type confusion vulnerabilities, which could pose a significant security threat.

3.4.5 Compiler Casting Optimisations

Given the recursive traversal of RTTI, dynamic casting is undoubtedly more expensive than any other cast operation. So, compiler vendors will optimise a dynamic cast when they can, converting it to a static cast instead. However, compilers must not impede the security of a dynamic cast with their optimisations. This is achieved by categorising casts (within a hierarchy) as one of three types: up-cast, down-cast, or cross-cast:



Up-cast is a cast from a derived-type to a base type, meaning a cast that travels 'up' the hierarchy.

Down-Cast is a cast from a base type to a derived-type, meaning a cast that travels 'down' the hierarchy.

Cross-cast is a cast performed on two class types that do not inherit from one another but are related through a shared derived class. It is a cast that travels 'across' the hierarchy.

For non-virtual hierarchies, an up-cast can always be simplified to a static cast. As seen in Chapter 2, derived class instances always have a base class sub-object instance within their memory region at a specific offset location. Therefore a cast to a base class sub-object is guaranteed to be safe and can be optimised to a `static_cast`. However, this is not true for virtually inherited base classes. The exact offset of a virtually inherited sub-object instance is known only at run-time; therefore, an up-cast to a virtually inherited type must be performed dynamically.

Down-casts and cross-casts, on the other hand, are considered unsafe casts. This is because, unlike an up-cast, there is no guarantee that the cast's target (a derived-type) exists within the source object. Therefore, these types of casts cannot be optimised to a static cast and must be performed dynamically.

Example Consider the code listed in Figure 40; here, we have created two object variables and performed three different casts on each, two down-casts (lines 4-8), two up-casts (lines 10-14) and two cross-casts (lines 16-20), where the result of each cast is stored in a new variable. The first variable, `h1`, is a `Human` pointer that addresses a `Human` object at run-time. The second variable, `h2`, is a `Human` pointer that addresses an `Avatar` object at run-time. We have also depicted these objects in Figure 40, highlighting all the address-points used and by which variables. Notice that this depiction includes the variables created through successful casts, demonstrating the results of each cast.

A notable result from these casts is the `m2` variable defined in line 13. It is noteworthy because this cast is not affected by the ambiguity of having two possible `Mortal` targets from that dynamic cast. The reason is that this cast is an up-cast, and up-casts are optimised to a static cast. Statically this cast believes it is executing solely on a `Human` object (the static type of `h2`). As a result, its dynamic type `Avatar` (and, in turn, its other `Mortal` member) is never considered. So the cast is treated identically to the `h1` up-cast in line 11.

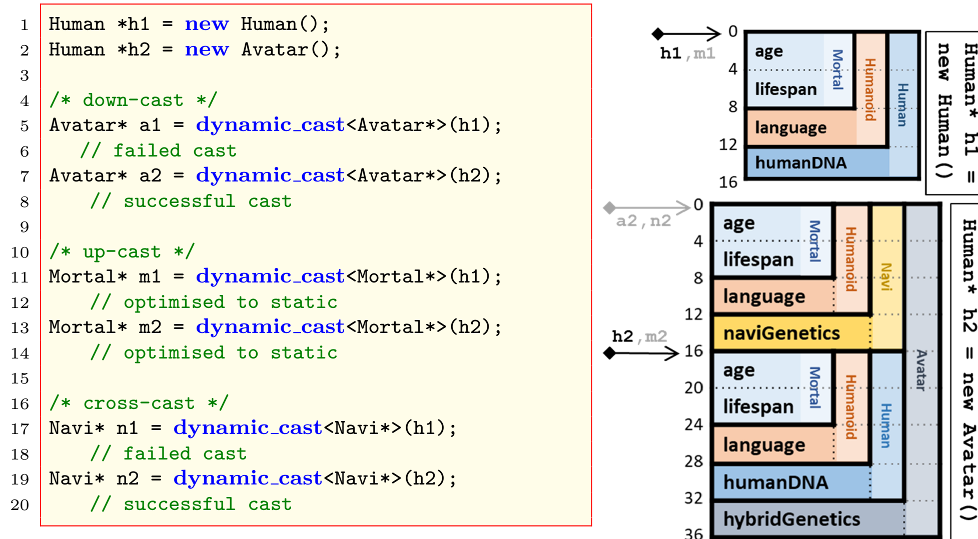


Figure 40: Types of casting example

Other noteworthy casts are the cross-casts performed in lines 17 and 19, as we have not seen these types of casts before. Cross-casts are considered unsafe, so no cast optimisation is considered here, and they must utilise the `dynamic_cast` mechanism at run-time. The `dynamic_cast` mechanism, in both cases, will traverse the RTTI data structures for both run-time objects. As `h2` addresses an `Avatar` object at run-time, the `dynamic_cast` mechanism (performed in line 19) will find a `Navi` RTTI object in `Avatar`'s RTTI data structure and will therefore result in a successful cast. The opposite is true for the cast in line 17. As `h1` addresses a `Human` object at run-time, its RTTI structure will not contain a `Navi` RTTI object, so will rightly fail. We note that such cross-casts would never pass compiler type-safety checks if they were defined using `static_cast` (i.e. `static_cast<Navi*>(h1)`). This is because, on their own, the `Human` class is unrelated to the `Navi` class; they are connected only through a shared derived class (`Avatar`). So, to bypass type-safety checks, an intermediate cast to `Avatar`* would be required (i.e. `static_cast<Navi*>(static_cast<Avatar*>(h1))`). However, this would not only result in a type confusion vulnerability but is exceptionally bad coding practice.

3.4.6 Custom RTTI Solutions

Due to the expense of performing RTTI checks, some developers create their own custom run-time type-checking techniques [72]; this includes LLVM, the developers behind the Clang compiler, who designed their own source-based RTTI mechanism coined LLVM-style RTTI [76]. The LLVM-style RTTI can be divided into two parts: an enumeration-based solution supporting type-checking in dynamic and non-dynamic classes and an extensible RTTI framework that supports open hierarchies with a custom dynamic casting function called `dyn_cast`.

LLVM-style RTTI Enumeration Solution This style of RTTI relies heavily on the programmer to ensure type definitions are accurately defined and instantiated. Each primary class will declare its own enumeration, and every base class will have a value defined in that enumeration. The primary class stores one of the enumeration values as a data member. All base classes will inherit this data member and store the enumeration value associated with the object's dynamic type at run-time. With this, every object, whether dynamic (has a vptr) or not, will store an integer value (enumeration value) linked to its dynamic type. It is this value that is used in LLVM's custom RTTI functions, such as their `isa<C>(obj)`, which will check if the run-time object `obj` is an instance of `C`, and `C::classof(obj)`, which will check if the run-time object `obj` is a derived-type of `C`. This technique allows for fast type checks and extends type-checking to non-dynamic classes but does not support virtual inheritance.

Extensible RTTI This style of RTTI uses class templates to realise RTTI data and establish relationships. In this scenario, all classes in a hierarchy will inherit from a class template called `RTTIExtends`. This template takes two parameters, the type of the inheriting class and the type of the direct base class, creating a link between direct parent and child classes. Every class that inherits from `RTTIExtend`

must define a `static char ID` value, as the address of the static member is used to identify each individual type. For the primary class, the parent type is defined as `RTTIRoot`, an LLVM-defined class whose member functions are responsible for resolving ID checks and are used within LLVM's custom dynamic casting call `dyna_cast`. Although LLVM's Extensible RTTI can perform type-checking much faster than traditional RTTI, it is restricted solely to single inheritance hierarchies.

The Downside to Custom RTTI Although custom RTTI solutions benefit from faster type-checking, they often suffer incomplete coverage by ignoring virtual inheritance hierarchies or being restricted to single inheritance (further discussed in Section 4). In addition, like with LLVM-style RTTI, custom RTTI solutions tend to require heavy manual code modifications [72] and, as a result, may be susceptible to human error.

3.5 Concluding Discussion

This chapter looked closely at the assembly produced by the g++ compiler when realising member functions, constructor functions, and cast operations. This overview discussed the different outcomes of invoking virtual and non-virtual member functions, demonstrating how virtual functions can be dispatched without knowing a variable's dynamic type. We also covered constructor functions and their order of initialisation; specifically, base class constructors are invoked first (creating a nested constructor call), then vptrs are assigned, followed by the execution of the constructor body. Finally, the chapter ended by exploring how cast operations are realised in assembly. Within these explanations, the disadvantages of each cast were explored, be it the performance overheads incurred by dynamic casting or the potential type confusion vulnerabilities of non-dynamic casting.

This concludes Part 1 of this thesis, which provides the prerequisite knowledge required for all later chapters in part 2.

Part II

Object Vulnerability and Exploitation

Chapter 4

Type Confusion Vulnerabilities

In this chapter, we review in greater depth, type confusion vulnerabilities and the security threat they pose. Type confusion can be prevented, either with run-time type testing (added to every cast), or with type confusion detectors (pre-release static/dynamic analysers that flag possible vulnerabilities to developers). We will critically review the strengths and limitations of both the detector and type testing approaches. Many of these approaches are built on top of type inclusion testing. Type inclusion testing is a field of research that develops a hierarchical encoding scheme and accompanying fast type-checking technique; where the type check will ascertain whether or not an inheritance relationship exists between two class types. We will provide an overview of this field, before finally discussing dynamic cast optimisation, a topic that has been largely overlooked within the field of type inclusion testing.

4.1 Introduction

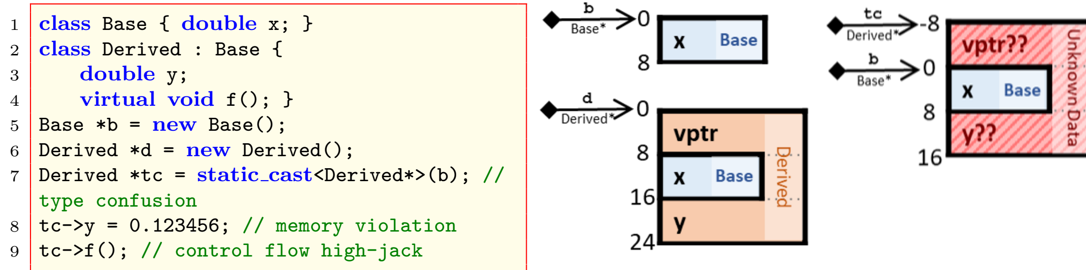
C++ is a large and complex language, with vast capabilities ranging from low-level programming (due to its ties with the C language) to high-level object-orientated (OO) abstraction. Having this range of capabilities (in particular the low-level

access), combined with human error, makes C++ prone to a variety of memory and type-safety vulnerabilities. One such vulnerability (as previously touched upon in Section 3.4) is type confusion.

Type confusion vulnerabilities are typically (but not exclusively [64]) introduced by programming errors. In this thesis we look only at type confusion vulnerabilities caused by casting errors introduced by the programmer; but note that in 2021, a paper was published discussing speculative type confusion attacks [64], a type confusion attack enabled through branch misprediction, within the Linux kernel. Such vulnerabilities are out of scope for this thesis, as they are caused by compiler optimisation techniques and are extremely difficult for a programmer to mitigate against; unlike type confusion vulnerabilities caused by casting errors.

Type Confusion Type confusion vulnerabilities occur when a section of code receives an unexpected object type and, without any type-checking, proceeds to execute that code on the data members of the unexpected object. Such vulnerabilities occur through unsafe down-casting (Section 3.4.5). Down-casting is unsafe when it is performed statically (i.e. at compile-time), relying on the diligence of a programmer to ensure type-safety without the use of run-time type information.

Simple Type Confusion Example Figure 41 illustrates a type confusion vulnerability. Lines 5 and 6 of Figure 41a initialise a `Base` and a `Derived` object to the respective variables `b` and `d` (depicted in Figure 41b). Line 7 performs a static down-cast from the `Base` object, stored in variable `b`, to a `Derived` type. This cast adjusts the pointer stored in `b`, by `-8` (the offset adjustment from a `Base` sub-object to a complete `Derived` object), and stores the result in the new pointer variable `tc` (depicted in Figure 41c). The `tc` variable is a `Derived*` type and is perceived to address a `Derived` object at run-time. This means the variable has access rights (Sections 2.3.3, 2.3.5, and 2.4.3) to a `Derived` object at that location, making it a confused variable. As the `static_cast` operator has



(a) Simple type confusion example (b) Object depiction (c) Type confusion

Figure 41: Type confusion listing and depiction

no run-time type-checking (Section 3.4.2), this cast goes ahead unhindered and execution continues to the next line. Lines 8 and 9 of Figure 41a are executed using the confused variable `tc`, both accessing extraneous data that exists outside the bounds of the original `Base` object. This illegal data access causes both a memory violation and a control flow high-jack, which could potentially execute silently without detection or exception.

Security Threat Type confusion vulnerabilities that involve dynamic objects are particularly dangerous as they grant a confused variable (like `tc` in the above example) access to a foreign vptr data member. The data stored in this foreign vptr can range from a carefully constructed pointer created as part of a malicious attack (discussed further in Chapter 6), to a completely random assortment of bits reinterpreted as a pointer. In either case, the foreign vptr grants the program access to an unforeseen (from the perspective of the programmer) vtable and virtual functions. As virtual functions are dynamically dispatched via the vptr (Section 2.4.3), using a mechanism with no type-safety checks, this could result in an illicit function call or, perhaps at best, a program fault. Without a program fault, the program may continue, silently executing alien functions on foreign data. For these reasons, type confusion vulnerabilities, when utilised by an attacker, can be the catalyst for much greater security threats.

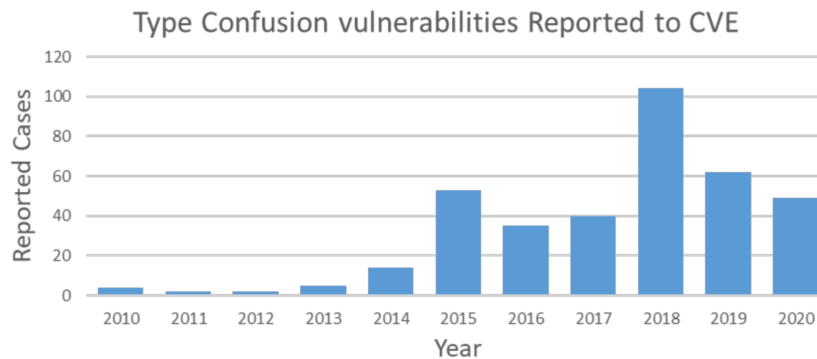


Figure 42: CVE reported type vulnerabilities in the last 10 years

Real World Vulnerabilities Research into type confusion vulnerabilities has gained traction in the last decade, with more than 400 vulnerabilities reported to the CVE (Common Vulnerabilities and Exposures) [25]. Figure 42 shows that the majority of these reports were made between 2015 and 2020, peaking in 2018 with over 100 reports in that year alone. Although there has been a modest dip in CVE reports since then, numbers still remain high (when compared to reports prior to 2015) and vulnerabilities are being uncovered in widely used applications: Google Chrome (22 cases), Firefox (10 cases), and several Adobe software products (25 cases) [25]. With mainstream software being greatly affected, the need to defend and/or mitigate such vulnerabilities is still hugely relevant.

4.2 Type Confusion Defense Strategies

There have been many attempts to try and curb the problem of type confusion, with researchers working hard on analysis tools, bug detection tools (known as sanitisers[1]) and exploit mitigation defences [34, 49, 60, 72, 90, 101, 131, 132, 144].

[1] The 2019 survey paper on C++ sanitisers [120], is a great source for direct comparisons of many sanitiser tools, some of which are discussed in this section.

	Exploit Mitigation	Sanitisers
The goal is to ...	Mitigate attacks	Find vulnerabilities
Used in ...	Production	Pre-release
Performance budget ...	Very limit	Much higher
Policy violations lead to ...	Program termination	Problem diagnosis
Violations triggered at location of bug ...	Sometimes	Always
Tolerance for false positives is ...	Zero	Somewhat higher

Table 5: Exploit Mitigation vs Sanitisers (taken from [120] © 2019 IEEE)

Exploit Mitigation or Analysis Tool? Type confusion defences typically come in one of two forms, a pre-release analysis tool or a run-time exploit mitigation. Each approach has different expectations regarding its functionality and purpose (as summarised in Table 5). Mitigations are run-time mechanisms embedded within a program during production; their goal is to identify an exploit during execution and, once identified using its various safety checks, mitigate that exploit by terminating the program. Their run-time checking techniques must have low-performance overheads with zero capacity for false-positive triggers. However, upon termination, there is no expectation for a mitigation to produce debug diagnostics. Analysis tools, on the other hand, are intended for pre-release bug detection. They aim to identify possible vulnerabilities and produce precise debug diagnostics for developers. As a pre-release tool, high-performance overheads are more accepted (to a degree), and false-positive results are accepted but at a low rate. A higher level of coverage is expected from a static analyser compared to a dynamic analyser (known as a sanitiser) because dynamic analysis is naturally restricted to the control flow paths performed during execution.

Static Analysis and Sanitisers Purely static type confusion analysis is still in its infancy. The only offering in this space, TCD [144], applies a type-aware pointer analysis technique that allows for improved path coverage compared to a purely dynamic approach that considers only a single run. Although pointer analysis can be formulated as an instance of abstract interpretation, pointer analysis rarely comes with a rigorous soundness argument [10, 41] for anything other than a

limited subset of C++. This puts doubts on any purely static type confusion detectors that are realised on an underlying pointer analysis.

Static type confusion analysis can potentially discover new vulnerabilities, as TCD demonstrates with the Qt library [107]. Yet the analysis times are prohibitive (exceeding 7 hours for Qt), and false-positive rates are high (28% on Qt). Consider now the developers using TCD as an analysis tool. They will not only suffer the excessive time of the analyser to evaluate their code, but they must also inspect every warning it generates, filtering for actual errors amongst the false positives. This will inevitably impact development times.

The dynamic type confusion analyser (or Sanitiser), UBSan [132], maps vptrs to types within a hash table and uses that table to check type consistency, thereby identifying possible type confusion errors. However, as others [49, 60, 72, 101, 120, 144] have pointed out, this solution does not support non-polymorphic objects (i.e. objects without a virtual pointer), limiting the applicability of UBSan. Other dynamic approaches (Caver [72], TypeSan [49], HexType [60], Bitype [101]) create their own hierarchical encoding schemes, which are essentially customised type information data/structures (featuring less type information than the standardised RTTI[2]), and accompanying type relationship check mechanism. In these approaches, the customised type information is mapped to a run-time object and stored in a disjoint metadata table. This metadata table is used as part of the custom run-time type relationship checking mechanism. For both Caver and HexType, their type-checking mechanisms incur significant performance overheads on the Firefox browser benchmarks, averaging 64.6% [72] and 60.9% [60] respectively. These high overheads are acceptable for sanitisers, but being dynamic means complete path coverage is not guaranteed.

[2]Encoding schemes will often focus solely on a fast relationship test to determine if two classes are related, so their hierarchical information is limited to these relationships and will therefore often exclude other type information such as: the type of inheritance relationship (virtual/multiple/single), the number of base classes, the name of the type, offset to base classes, etc...

EffectiveScan [34], is perhaps the most comprehensive sanitiser available, offering checks on types, casts, implicit casts, bound checks (for both complete- and sub-object overflows), as well as use-after-free detection. Their prototype attempts to provide an all-in-one sanitiser, detecting a larger variety of possible vulnerabilities and removing the task of running multiple tools. These vulnerabilities are detected using type metadata assigned to every object using low-fat pointers [68]. A fat pointer simply refers to a pointer that stores some metadata on top of an object's address. The metadata makes the pointer larger in size, greater than one word, hence the word 'fat'. A low-fat pointer is, therefore, a fat pointer, with some additional encoding scheme, which limits the amount of 'fat' used around the object address; i.e. utilising any unused bits, like those guaranteed to be zero. Low-fat pointers allow quick and easy access to the metadata and in the case of EffectiveScan, provides all the metadata required for their checking and detection schemes. With such comprehensive checks, it is unsurprising that EffectiveScan has high overheads (even for a sanitiser), averaging 422% [34] on the Firefox browser benchmarks.

Always-On Sanitisers TypeSan [49], Bitype [101], and CastSan [90] market themselves as 'always-on sanitisers' rather than as mitigations. Always-on sanitisers blur the lines between the distinct expectations of both mitigations and sanitisers, as they often both mitigate attacks at run-time, but also find vulnerabilities and produce debug diagnostics. Their expected performance overheads are more lenient than that of true mitigations, which must demonstrate less than 10% performance overheads (considered to be the acceptable slowdown) before even being considered for real-world adoption [120]. These more lenient expectations of performance overheads can be seen in TypeSan's and Bitype's results, respectively achieving on average a 34% and 16% performance increase for the Firefox browser benchmarks. These tools use hierarchical encoding schemes alongside disjoint

metadata tables to perform their type-checking techniques and, although these techniques produce significantly less slowdown compared to other sanitiser works, they are still considered (when viewed as a mitigation) too slow for real-world adoption.

Mitigations Clang CFI [131] is a real world mitigation, offered within the Clang compiler. The Clang Team has not reported on the run-time performance of its CFI mitigation, but it was analysed against the dynamic sanitiser CastSan [90]. According to the CastSan paper [90], Clang CFI applies an average of only 2.04% performance overhead on SPEC CPU2006 benchmarks [50, 123], and CastSan itself reported 1% overhead. Unfortunately, the CastSan paper did not evaluate itself or Clang CFI against the Firefox benchmarks, likely due to the discontinuation of the Octane benchmarks in 2017 [134]. In all other papers discussed, the Firefox benchmarks induced much higher overhead costs compared to SPEC CPU2006 (for example, Bitype's overhead was 1.8% on SPEC CPU2006 compared to 16% on Firefox benchmarks). The SPEC CPU2006 consists of seven C++ benchmark programs, three of which contain no cast operations [49], and out of the four that do, only two (Deal.II and omnetpp) utilise dynamic casting. This brings into question how representative an **average** run-time overhead is, when it is generated from the SPEC CPU2006 benchmarks.

4.3 Type Inclusion Testing

Many of the encoding schemes used in the above sanitisers and mitigation approaches were influenced by the field of type inclusion testing. Type inclusion testing is the act of checking a type is related to another in a hierarchy, with most work focusing on developing a technique of constant time, linear space, and (particularly for work in later years [18, 35, 66, 143]) a compatibility with both multiple and virtual inheritance. In all cases, each of the techniques introduce

their own type encoding scheme to represent type information, alongside a sub-type testing function.

Adding a type encoding facility to determine object types in C++ was first posed by Dmitry Lenkov in 1991 [73]. Lenkov then went on to work alongside Bjarne Stroustrup to design the mechanisms for run-time type identification in C++ [128], which was accepted by the ANSI/ISO committee by March 1993 [127]. But the concept of type inclusion testing preceded C++ and other OO languages. One of the earliest works by Schubert at el. [111], tests type inclusion using range checks. Classes are assigned an interval $[l, u]$, where each number within this range represents a derived class. If one class's interval exists within another, then they are related. This technique is cost-effective but limited only to single inheritance. Later work by Zibin and Gil [143], was able to apply this same range check to their encoding scheme called PQ-Encoding [143]. PQ-Encoding uses PQ-trees [12], a tree based data structure used to represent permutations of a set. The encoded data set, used in the PQ-Encoding scheme, is generated from a PQ-tree, which holds values representing the intervals assigned to each class. Intervals are generated using the PQ-Encoding algorithm, which finds the best permutation of the PQ-tree using some prescribed hierarchical constraints for each resulting interval. These intervals are assigned in such a way that simple type inclusion range tests can be performed not just for single inheritance, but for multiple inheritance as well.

Wirth [139] presented a linked list solution to represent the class hierarchy and a sub-type test that traversed this structure, looking for an equality match. The technique required little additional memory, but only worked for single inheritance. It could be extended with backtracking for multiple inheritance, however, back-tracking added considerable run-time overheads to an already expensive type testing technique, as the cost of testing was directly impacted by the distance of the two types within the hierarchy. A faster technique, and one adopted in some early

compilers [65, 29], was encoding type information in binary matrices. Type inclusion testing, therefore, becomes a simple element check at $Matrix(Target, Source)$. Inclusion testing was fast and efficient, but has quadratic memory space requirements. Techniques to compress the matrix can reduce the amount of required space, but often results in slower verification testing. As seen with Vitek et al. Packed Encoding and Compacted Encoding schemes [137], any form of matrix compression creates a trade-off between space and time.

Cohen was the first practical solution for constant time sub-type testing [24]. It assigned every class a unique type ID and inheritance array. The array stores the type IDs for each class it inherited from, positioned at the index equivalent to their own unique ID. This results in a simple array bounds check and comparison, but was only applicable to single inheritance. After Cohen's paper, attempts were made to reduce the memory capacity of hierarchical encoding, using bit-vectors (also known as bit arrays). Caseau describes a top-down hierarchy encoding scheme [18] that maps the hierarchy to an upper semi-lattice structure (transforming where necessary) and assigns each node a bit-vector, representing each class and its inheritance relationships. Each bit-vector (within that hierarchy) is unique, but is also guaranteed to contain the union of all its base class bit-vectors. Type inclusion testing is fast, requiring only a few bit-wise operations. Krall et al [66] and Vitek et al. [137] expand on Caseau's work, creating two different encoding schemes without the requirement of a lattice-based hierarchy mapping.

4.4 What about Dynamic Casting?

Dynamic casting (introduced in Section 3.4.4) can be used to securely perform a down-cast. If line 7 of our type confusion example (Figure 41) was replaced with a `dynamic_cast`, then no memory violation or control flow high-jack could

occur. Unfortunately, the `dynamic_cast` mechanism and its use of RTTI has memory and executional overheads, which are often regarded as prohibitively high [49, 72, 60, 101]. In our own experiments we have shown dynamic down-casting to be a minimum of one order of magnitude slower than that of static casting. Others, such as Lee at al., reported dynamic casting to be two orders of magnitude slower than static casting on average [72], but as we will see in Section 5.4.1, this estimate is questionable due to the type dependencies of dynamic casting. Either way, dynamic casting incurs an obvious expense, which is why it is often avoided.

Surprisingly, despite all the work done in type inclusion testing, none of the published work [18, 24, 66, 111, 137, 139, 143] discusses how their encoding schemes can be applied to dynamic casting, one of the fundamental type-checking facilities included in C++. Without this discussion, it is unclear how these schemes could be incorporated as part of the `dynamic_cast` operation, specifically in the case of multiple and/or virtual inheritance. Integrating any of these encoding schemes for single inheritance is easy; objects have only one address-point, meaning no pointer adjustment is needed, and therefore, dynamic casting is just a type inclusion check. However, a dynamic cast may perform a pointer adjustment for objects with multiple address-points, like those from multiple and/or virtual inheritance hierarchies. To perform an object pointer adjustment, access to run-time offset information is required through a mapping between types and offset data. As none of these schemes (most notably those that target multiple inheritance [18, 66, 143]) consider dynamic casting, this mapping between types and offset information is also overlooked. Without this consideration, their encoding schemes are limited in their practical use and could never outright replace RTTI. We will now discuss work that specifically targets dynamic casting, not just type inclusion testing.

Fast Dynamic Casting The most notable work in dynamic cast optimisation is that of Gibbs and Stroustrup and their fast dynamic casting technique [45]. In their work, every class is assigned a prime multiplier, a type ID, and an offset value, adding three integers to that class's vtable. A class's prime multiplier must be unique from its base class prime multipliers, but can be the same as other classes in the hierarchy, provided that they do not share a common descendant. A class's type ID is equal to the product of its own prime multiplier and all of its base class prime multipliers (note that every prime in this calculation is unique). As a class's type ID is simply a multiple of unique prime numbers, then any base class can be quickly verified using a check for divisibility. That is, class X is a base of class Y if and only if

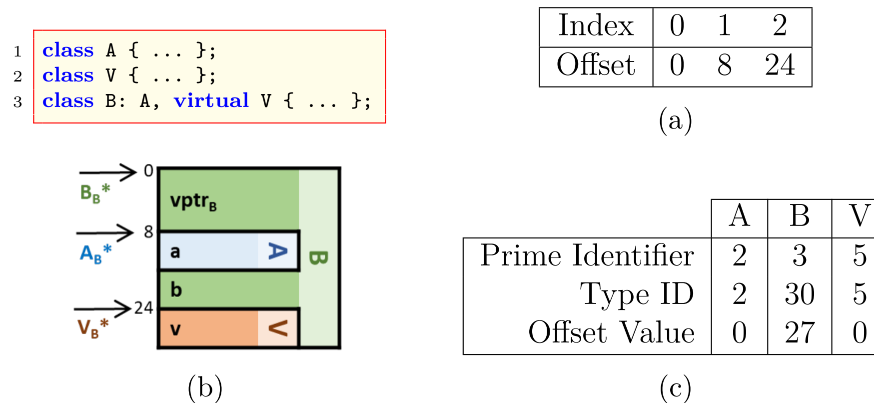
$$\mathit{type_ID}(X) \mod \mathit{type_ID}(Y) = \mathit{0}$$

Figure 43 provides an example of the fast dynamic casting encoding scheme. Figure 43b presents a new virtual inheritance hierarchy and accompanying object of the most-derived class `B`. Figure 43a presents an offset table, created as part of the encoding scheme, which stores the three offset values to each address-point in the complete `B` object. Lastly, Figure 43c presents the result of the encoding, listing the prime multiplier, type ID, and offset value assigned to each class in that hierarchy.

Offset values are used to retrieve offset information from the offset table, providing a way to determine the offset location of a sub-object at run-time. The calculation is as follows,

$$i = \mathit{offset_value}(Y) \mod \mathit{prime_multiplier}(X)$$

where X is the sub-object type (base class), Y is the complete object type, and i is the index entry of the offset table, which provides the offset adjustment to X



Index	0	1	2
Offset	0	8	24

(a)

	A	B	V
Prime Identifier	2	3	5
Type ID	2	30	5
Offset Value	0	27	0

(c)

Figure 43: Fast Dynamic Cast Example

in Y. Each class's offset value is calculated so that the above equation is satisfied for all unambiguous base classes X in Y. In our example, $\mathit{offset_value}(B) = \mathit{27}$, as this was the lowest integer satisfying

$$\begin{aligned} \mathit{offset_value}(B) &\mod \mathit{2} = \mathit{1} \\ \mathit{offset_value}(B) &\mod \mathit{3} = \mathit{0} \\ \mathit{offset_value}(B) &\mod \mathit{5} = \mathit{2}. \end{aligned}$$

This technique performs a fast, constant time type inclusion test and offset retrieval for dynamic cast operations, but it also has hierarchical limitations. These limitations come from the capacity of the type ID variable, which restricts the depth of the hierarchies this scheme can represent. The type ID variable stores the product of a set of unique prime numbers and can overflow if too many prime multipliers are used. For example, the capacity of a 32-bit (or 64-bit) word is surpassed when calculating the product of the first 10 (or 16) prime numbers. Although Gibbs and Stroustrup's design [45] allows for repeated prime identifiers across the hierarchy, repeats must not share a common descendant. This means, the greater the depth of a hierarchy, the larger the type ID variable needs to be, which overall restricts fast dynamic casting to shallow hierarchies. For this reason,

Gibbs and Stroustrup's technique is primarily aimed at smaller applications, with smaller hierarchies, like those in embedded systems [45].

Dechev et al. [32] evaluated the applicability of Gibbs and Stroustrup's fast dynamic casting technique within a Data Management Service, which is part of a Mission Data System application. This application is part of a mission critical system for remote autonomous spacecrafts and was developed by the Jet Propulsion Laboratory [91] (a NASA funded research facility). Through this evaluation, they found an improved heuristic method of Gibbs and Stroustrup's type ID assignment scheme, which better assigns prime multipliers to reduce the size of type ID values in non-virtual hierarchies. This, in some cases, can increase the overall size of hierarchies that can be encoded using this improved scheme, but the scheme is still restricted to shallow hierarchies. The fast dynamic casting technique was later implemented within a library called EVL (Extended Virtual function Library) [71]. Their library supports multi-methods, an extension to the classic virtual function, where the run-time implementation of the function (through dynamic dispatch) is dependent on the run-time type of two or more objects.

Perfect Hashing Ducournau [35] presented a new encoding scheme and type inclusion test using perfect hashing. In this scheme, additional type information is added to the vtable along with a pointer to a hash-table, storing offset information. Their paper trials a few different hash functions, using mod and bitwise AND operations for speedy type inclusion testing. However, there is no performance comparison with Gibbs and Stroustrup's fast dynamic casting [45], but an inspection of the code snippets available in each paper would suggest fast dynamic casting outperforms Ducournau's perfect hashing. In addition, perfect hashing only works for virtual inheritance, requiring further adaptation for full compatibility with all types of hierarchies.

Fail-Fast Dynamic Sub-type Checking Padhye and Sen [100] presented an encoding scheme called Fail-Fast, that optimises for failed dynamic casts, i.e. casts that result in a null return. As discussed in Section 3.4.4, dynamic casts will recursively traverse an RTTI data structure, searching for an RTTI pointer that matches the cast's target type. When source and target types are unrelated at run-time, these calls are often the most expensive, as every RTTI object must be checked before a null result is returned. Padhye and Sen observed in their own experiments that 74-93% of dynamic casts result in a null return. With this observation, they optimised only for failed dynamic casting to avoid complete RTTI traversal. Although targeting failed dynamic casts appeared fruitful in their own experiments (achieving 1.44x-2.74x speedup), we found that only 1.02% of run-time dynamic casts resulted in a null return in our own investigations of the Deal.II library [5] (discussed in Chapter 5). Thus the effectiveness of Fail-Fast is subjective to the code it optimises and cannot be recommended as a universal optimisation technique.

HexType HexType [60], as previously mentioned, is a sanitiser tool that verifies cast safety using its own hierarchical encoding scheme. It targets all forms of casting and instruments a run-time type check at each cast location. The HexType sanitiser introduces an optimised dynamic casting mechanism that utilises its own hierarchical encoding scheme. However, the introduction of the optimised dynamic casting mechanism was solely to reduce the overheads introduced by the sanitiser itself. So although HexType optimises dynamic casting, it is, first and foremost, a sanitiser tool, so it does not evaluate the performance of dynamic casting outside of the overheads introduced by the sanitiser as a whole. Thus it is hard to determine the performance benefits of their optimised dynamic casting technique.

4.5 Concluding Discussion

It is clear that there is an abundance of work in type inclusion testing and encoding schemes, each attempting to find the best trade-offs between time, space, and hierarchical coverage. Dynamic casting, which is known to be an expensive operation, has a notable use for such schemes, as a type inclusion check must be performed before any cast adjustment. Despite the obvious applicability of type inclusion testing techniques in dynamic cast operations and the possible performance improvements they could achieve, we found only three type inclusion works that explicitly target dynamic cast optimisation. These works were Gibbs and Stroustrup's fast dynamic casting, Ducournau's perfect hashing, and Padhye and Sen's Fail-Fast checking. Fast dynamic casting achieved high performance, with a constant-time checking mechanism, but limits the size of the hierarchy. Perfect hashing could encode any size hierarchy but has slower performance and cannot handle non-virtual inheritance. Fail-Fast checking can improve the performance of programs with high occurrences of failed dynamic cast operations, but make no improvements to successful cast operations and therefore cannot be a universal solution. We thus conclude that there are still opportunities to optimise dynamic casting, both in speed and hierarchical coverage.

Chapter 5

Memoised Casting

We have looked at past research on type confusion, type inclusion testing, and dynamic cast optimisation, concluding that there are more research opportunities to be had in optimising the dynamic cast operation. This chapter presents our optimisation technique.

5.1 Introduction

In the previous chapter, we discussed the prevalence of type confusion vulnerabilities in mainstream software and the security threat they pose. We also discussed various prevention methods and observed that safer casting techniques could prevent most type confusion vulnerabilities. Dynamic casting is a facility offered by the C++ language, which can safely and correctly perform a down-cast operation, the most dangerous form of casting. However, despite its availability, dynamic casting is often omitted due to performance overheads, which are regarded as prohibitively high [49, 72, 60, 101]. Whenever dynamic casting is deployed, it comes at a cost; that cost comes from type-checking, as this requires a recursive traversal of an object's RTTI data-structure (Section 3.4.4). The traversal depth is determined by the distance (within the hierarchy) between the source and target types

supplied to the dynamic cast. The greater the distance, the greater the cost.

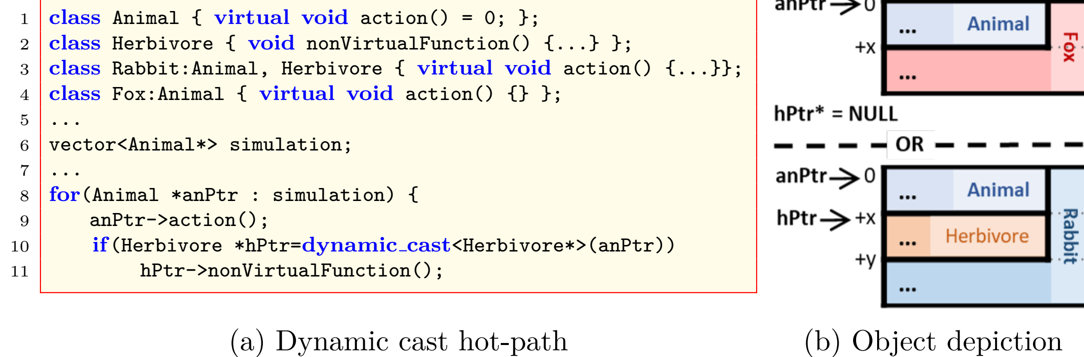
Research has primarily neglected optimisation techniques of the dynamic cast mechanism, favouring complex encoding schemes focused on fast type inclusion testing. However, RTTI provides the most robust and complete type information system and conforms to both language standards and ABI specifications. For this reason, we chose to design an effective and easy to deploy optimisation technique called MemCast. MemCast is a memoising wrapper function for the dynamic cast mechanism, which removes much of the expense of RTTI checks and improves its overall performance. With this improved performance, there is no reason why a programmer should shy away from deploying dynamic casts through the MemCast wrapper.

Hot-Paths From our experimental work (Section 5.2), we made two critical observations:

1. The cost of a single, one-off `dynamic_cast` call is almost negligible in the context of an entire program, but when it occurs on a hot-path (i.e. frequently called), its expense multiplies, magnifying the overhead.
2. Every dynamic cast call has a fixed target type but a variable source type.

To provide an example of these two observations, consider a simulation program that attempts to track the population of foxes and rabbits within a given region [8]. Figure 44a lists some of the code for this example. On line 10, there is a `dynamic_cast` with a fixed `Herbivore` target type, which exists within the `for` loop defined on line 8. The iterations of this loop are dictated by the `simulation` vector's size, which determines the number of `dynamic_cast` calls during execution, creating a possible hot-path. This particular cast can arise in one of two ways (as seen in Figure 44b):

1. When `anPtr` addresses a `Fox` object, the cast fails, and `hPtr*` is set to `NULL`. With a result of `NULL`, the `if` statement in line 10 also fails, and line 11 is



(a) Dynamic cast hot-path (b) Object depiction

Figure 44: Dynamic Cast Hot Path with Depiction

never executed.

2. When `anPtr` addresses a `Rabbit` object, the cast is successful, and `hPtr*` is assigned a new address equal to `*anPtr + x`. The adjustment of `x`, which will have been retrieved from `Rabbit`'s RTTI object, is the offset to the `Herbivore` sub-object.

One might expect an abundance of rabbits but relatively few foxes in the `simulation`, as it attempts to represent a real-world scenario. With multiple `Rabbit` objects, the code in Figure 44a will repeatedly cast the `Rabbit` objects to the `Herbivore` sub-object. This means that the dynamic cast mechanism repeatedly traverses the `Rabbit` RTTI data-structure and returns the same offset displacement result. This repeated checking and lookup is inefficient.

Stability We found that many hot-path dynamic cast sites will perform the same cast successively (i.e. cast the same source type as its previous execution). We describe this successive recurrence of source types, for a given cast site, as that cast's **stability**. Section 5.2, expands on this concept and introduces a calculation that, when used in part of a dynamic analysis investigation, can assign a percentage value to each cast site representing its stability. Cast sites with high stability (100%) receive the same source type with each execution, whereas unstable sites (0%) receive a different source type with each visit.

MemCast This chapter investigates cast stability within a real-world library application (Deal.II [5]), exposing the cost of repeated type-checking on dynamic cast hot-paths. Motivated by this study, we built and designed MemCast, a source-based memoising wrapper operation for the `dynamic_cast` operator that removes the repetitive RTTI traversal for successive casts. MemCast works by reserving two words in global memory for each cast location. This pair of words is used to store a unique ID for the last source type passed to the dynamic cast, a flag indicating a successful cast or null pointer return, and (for the successful case) an offset adjustment. By caching this data, MemCast can invoke the `dynamic_cast` mechanism once and avoid repetitive type-checking and RTTI traversal for successive casts of the same source type.

The MemCast technique does incur a cost each time the cast site receives a type different from the last; this cost is equal to the cost of a dynamic cast, plus MemCast's constant time type-checking technique and data caching mechanism. However, the amortised run-time becomes vanishingly small over time for high-stability casts, as the dynamic cast is not repeated for stable cast locations. When repeated dynamic casts are removed, the cost of a MemCast is typically equivalent to a virtual function call, which is approximately 6 times faster than the cheapest form of dynamic casting. A MemCast can be marginally more expensive for unstable casts, but we show through our experiments that only a small minority of cases are unstable. Additionally, we can mathematically estimate the minimum stability a cast site requires to outperform dynamic casting, indicating that most cast locations benefit from our MemCast technique.

To demonstrate the benefits of MemCasting, we tested the technique on three different C++ libraries. The libraries used in these tests were the Deal.II library [5] (wich was the focus of our stability investigation), the OMNet++ library [98], and the Antlr4 C++ run-time library [102]. In each case, we found an average performance speedup between 1.63-1.68%, 1.11-1.97%, and 2.75-2.82% for each

library, respectively.

Contributions This chapter makes the following contributions:

- We introduce the concept of cast stability and how it can quantify the successive recurrence of source type in down-cast locations as a percentage through profiling.
- We investigate the stability of dynamic down-cast locations and perform a systematic analysis to show where MemCasts are most beneficial for various stability values.
- We introduce MemCast, an easy-to-deploy, memoising wrapper for the `dynamic_cast` operator that, when successful, is equivalent in cost to a virtual function call.
- We evaluate MemCasting when applied to three different C++ libraries and demonstrate an overall run-time performance improvement between 1.11% and 2.82%.

Chapter Structure In Section 5.2 we introduce and investigate cast stability within the Deal.II library. Our findings prompted the design and construction of MemCast, which is discussed in Section 5.3, and a performance evaluation is discussed in Section 5.4. Beyond this, we briefly discuss related work in Section 5.5, future work in Section 5.6, and finalise with a concluding discussion in Section 5.7.

5.2 Cast Stability and Deal.II Analysis

As mentioned previously, a key observation of the dynamic dispatch mechanism is that the cast sites themselves (i.e. the individual casts located in the source code) have fixed target types but variable source types that seldom vary, if at all. In this section, we want to investigate the variation of source types for a typical down-cast site. To do this, we introduce the concept of cast stability.

5.2.1 Cast Stability

Cast stability is used to describe the successive recurrence of source types for a particular cast site location, which we quantify as a percentage:

Definition 5.2.1 (Cast Stability Formula)**.** *Let $t_1, ..., t_n$ be a sequence of source types that a single cast site location receives after n visits. From this sequence of types, the stability (S) of a cast site is*

$$S = 1 - \frac{C}{P}. \quad where \quad P = n - 1 \quad and \quad C = \sum_{i=1}^{P} \begin{cases} t_i \neq t_{i+1} & 1 \\ t_i = t_{i+1} & 0. \end{cases} \tag{1}$$

Observe that $P > 0$. The intuition is that C is the total number of times a source type (t_k), passed to the cast site, changes when compared to the previous source type (t_{k-1}) in the sequence; and P is the total number possible source type changes that could occur ($n - 1$) excluding the first visit of the cast, as the initial source type (t_1) is not considered a change in the cast's source type sequence. Finally, observe that $S \in \mathbb{R}$ where $0 \leq S \leq 1$.

Example 5.2.1. *Table 6 provides an example of calculating stability values for five different cast sites, each executed $n = 5$ times, receiving a source type of either A or B. The first execution of all five casts is ignored when counting source type changes; as it is the initial value, it is not considered a change. The total number of possible source changes is, therefore, $P = n - 1 = 4$. For all successive source types in the sequence, we count the number of times the source type changes (C) from its previous type, which varies for each observed cast. From this, we can calculate each cast site's stability. Stability is presented as a percentage; the higher that percentage, the more stable a cast site is.*

Source type sequence of each cast site	Total Changes (C)	Calculation $S = 1 - \frac{C}{P}$	Stability (S)
A A A A A	0	$1 - \frac{0}{4}$	100%
A A B B B	1	$1 - \frac{1}{4}$	75%
A A B B A	2	$1 - \frac{2}{4}$	50%
A B A A B	3	$1 - \frac{3}{4}$	25%
A B A B A	4	$1 - \frac{4}{4}$	0%

Table 6: Stability Calculation Example

5.2.2 Deal.II Experiments

We wanted to investigate cast stability in a real-world program. For this, we chose the Deal.II library [5], a finite element library that was used as part of the SPEC CPU 2006 benchmarks [123]. We chose this library partly because it illustrates how dynamic casts are deployed in simulation and partly because it is open source.

We note that dynamic down-casts and cross casts are treated identically in our analysis and within the MemCast mechanism, so there is no need to distinguish between the two in this setting. For this reason, assume from this point forward that any discussion of dynamic down-casting (the most common form of dynamic casting) also encompasses dynamic cross-casting unless expressly stated.

Method We first needed to identify each dynamic down-cast within the source to estimate cast stability within the Deal.II library (233,463 lines of C++ code, over 594 separate files). It is not sufficient to use simple tools such as grep because it is necessary to use type information to distinguish between dynamic up-casts and dynamic down-casts. Dynamic up-casts are rewritten to static casts by the compiler (Section 3.4.5), so their stability outcomes are irrelevant to our work as they are already optimised by default. The compiler does not optimise dynamic down-casts, so it is these casts we want to investigate. To identify each dynamic down-cast, we built a Clang tool [130], which allowed us to leverage the Clang compiler's front-end and static type inference engine to analyse the Deal.II library's

abstract syntax tree (AST). Our tool extensively used the RecursiveASTVisitor class, allowing us to visit each cast expression within the AST. The cast types were then inspected to determine whether the cast was a dynamic down-cast, and if so, the cast location was recorded in a separate text file. This tool identified a total of 545 dynamic down-cast sites within the Deal.II library.

We wrote a Python script that would use the results of our Clang tool to transform every dynamic cast site (identified in the separate text file) to generate a new source version. The Python script replaced each dynamic cast call with a macro that took the source and target types of the cast as arguments. The rationale behind using a macro is that different instantiations of this macro, which performed different forms of profiling, could be trialled without rerunning the Clang tool and Python script. This macro was designed as a wrapper for the `dynamic_cast` operator, which profiled every cast location's input and output types. For our analysis, we used the example programs that came with the library, each called step-x, where x is an integer. The run-time of each step-x program varies from several minutes to a fraction of a second, reflecting the complexity of the simulations (see [5] for further details). From the 59 programs provided, 49 compiled successfully, executed without run-time error and did not require any fixed input. After transformation with our Python script, we found that 33 of these programs performed at least one of the identified dynamic down-casts during execution. After running all 33 programs, we found that only 82 of the 545 cast sites we identified were exercised. Our analysis results are therefore based on these 82 sites. Note that each cast site exists within a Deal.II library function, where the library itself is shared across all step-x programs. Therefore, each cast site can be featured within the execution of multiple step-x programs[1], namely those programs that contain a control-flow path leading to the same library function.

[1]See Table 22, Appendix B for cast numbers, source locations, and featured steps.

Cast Number	Step-x	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability ($S = 1 - \frac{C}{P}$)
49	10	13	0	100.00%
	11	143	0	100.00%
	24	105,119	0	100.00%
	38	3	0	100.00%
	47	51	0	100.00%
143	12b	29	12	58.62%
	16b	79	32	59.49%

Table 7: Stability of Cast Site 49 and 143. Full results in Table 22 of Appendix B

5.2.3 Deal.II Results

After executing all 33 step-x programs, a total of 459 results were recorded. Each recording focused on a particular cast, the step-x program it featured in, the values of P and C (as defined in the stability equation in Definition 5.2.1), and the stability calculated from those values. To present all 459 results here would be excessive, so the full results are featured in Appendix B (Table 23). Table 7 presents a snippet of the full results, featuring two casts, cast number 49 and 143[2]. Cast 49 featured in a total of five step-x example programs (steps 10, 11, 24, 38, 47), whereas cast 143 featured only in two (steps 12b and 16b). For each step-x execution, the table lists the values for P, C and S. The results of these two cast sites and all others recorded in the full results table (Appendix B, Table 23), were consolidated into a single table (Table 8a). Table 8a shows how each cast performed across all programs collectively, listing the cast number, the number of step-x programs it featured in, the total values for P and C, and the overall stability value (S), which was calculated from the total number of visits recorded across all step-x programs S (rounded down).

Table 8b counts the number of cast sites that had an overall stability within a given percentage range from Table 8a. This table shows that a majority, 60/82, of

[2]Note that casts were labeled numerically in the order they were found by the Clang Tool. Some casts were not featured in the 33 step-x programs, which is why gaps appear in the cast numbering within result tables.

Cast Num	Featured in N Steps	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Overall Stability (S)
1	29	62,183,926	1	99.99%
2	17	590,168	0	100.00%
3	11	493,809	0	100.00%
4	11	493,809	0	100.00%
5	2	772,416	0	100.00%
6	3	23,913	0	100.00%
7	29	56,770	1	99.99%
8	30	122,307	1	99.99%
9	27	48,714	1	99.99%
10	3	310	0	100.00%
11	11	410	0	100.00%
12	6	435	0	100.00%
13	5	286	0	100.00%
14	2	30,930	0	100.00%
15	30	342	1	99.70%
16	14	311	0	100.00%
17	1	32	0	100.00%
18	30	25	1	96.00%
19	30	25	1	96.00%
20	9	79	9	88.60%
21	9	79	9	88.60%
22	9	79	9	88.60%
23	9	79	9	88.60%
24	4	1	0	100.00%
25	4	1	0	100.00%
26	4	1	1	0.00%
27	2	4	0	100.00%
28	3	1	0	100.00%
29	3	3	0	100.00%
30	5	1	0	100.00%
31	5	1	0	100.00%
32	3	1	0	100.00%
33	7	1,799	0	100.00%
34	3	9,361	0	100.00%
49	5	105,329	0	100.00%
54	1	4,017	0	100.00%
70	2	46,624	0	100.00%
115	2	51	0	100.00%
121	3	59	0	100.00%
123	3	6,875	0	100.00%
126	2	104	0	100.00%
128	2	51	0	100.00%
130	2	51	0	100.00%
141	2	108	46	57.40%
142	2	108	44	59.25%
143	2	108	44	59.25%
161	1	299	121	59.53%
162	1	299	120	59.86%
163	1	299	120	59.86%
224	2	23,568	0	100.00%
227	2	12	0	100.00%
228	1	7	0	100.00%
234	2	0	N/A	N/A
235	2	0	N/A	N/A
241	2	18	0	100.00%
324	1	17	0	100.00%
330	1	3,717	0	100.00%
346	1	1,467	0	100.00%
347	1	1,467	0	100.00%
349	1	39	0	100.00%
351	1	13	0	100.00%
359	1	30	0	100.00%
361	1	27	0	100.00%
403	2	12	0	100.00%
410	1	13	0	100.00%
412	1	5,295	0	100.00%
455	4	89	0	100.00%
456	4	35	0	100.00%
457	1	15	0	100.00%
461	3	203,289	0	100.00%
465	1	49,031	0	100.00%
468	2	1,580	0	100.00%
469	2	152	0	100.00%
470	2	48	0	100.00%
479	5	81,028	0	100.00%
511	1	11	0	100.00%
513	1	5	0	100.00%
514	1	0	N/A	N/A
515	1	0	N/A	N/A
516	1	11	0	100.00%
522	3	25	0	100.00%
525	3	25	0	100.00%
Total		65365856	539	-
		Average Stability		94.89%

(a) Resulting stability (rounded down) of each cast site across all step-x programs

Stability (%)	s = 100	100> s ≥99	99> s ≥95	95> s ≥80	80> s ≥55	55> s >0	s = 0	N/A
Total Cast Sites	60	5	2	4	6	0	1	4

(b) Stability Ranges

Table 8: Cast site results across all test programs

the cast sites in Table 8a are 100% stable and a further 11 have over 80% stability. Only one cast, cast 26, was unstable (0% stability). This particular cast featured in four step-x programs (Appendix B, Table 23). In three of these programs it was visited exactly once, hence S is undefined. In the remaining program it was visited exactly twice, so would not be considered to be on a hot-path. Only four cast sites (228, 234, 514, and 515) had no stability value; this is because these sites were visited once and only once in their respective step-x programs. If we were to remove these casts from our analysis, focusing only on sites invoked more

than once, then **83%** of all Deal.II's dynamic down-casts are at least **99% stable** and **91%** of all cast sites are at least **80% stable**.

We can also calculate an average overall stability for all Deal.II's dynamic down-cast sites (see Table 8a). As most cast sites have a high stability, the average stability of all 82 sites is 94.89%. That is, 94.89% of all Deal.II's dynamic down-casts are performing a type check, with RTTI traversal, identical to the check performed in the previous cast visit. Given that collectively (see total in Table 8a), 65,365,856 casts are performed beyond the initial cast call, and only 539 source type changes are recorded in total, this means (65,365,856 - 539 =) 65,365,317 dynamic casts were performed unnecessarily throughout the execution of all 33 programs. This is an extraordinary level of redundancy.

5.3 Design and Implementation

MemCast is an effective and efficient memoising wrapper function for the dynamic cast mechanism. It is an optimised version of `dynamic_cast`, which reduces its overhead, making it a viable security mitigation against type confusion vulnerabilities. It is a source-based tool, allowing programmers to employ MemCast as and where they wish. This could be as a blanket conversion of all cast sites or a more tactical deployment to the locations where it would be most beneficial. MemCast is simple to deploy, requiring only two lines of code per cast site. The first line is the instantiation of a supporting MemCast object (called a **MemCache**). The MemCache object's sole purpose is to provide a cache for that particular cast located in the source code. The second line of code is the MemCast itself, where a `dynamic_cast` operator would be deployed using a MemCast wrapper instead. The MemCast wrapper receives the same source object and target type as a dynamic cast but must also receive the MemCache object as an additional parameter. The MemCache object is used for both type-checking and pointer

offset retrieval. During run-time execution, if the MemCast functions receive a different source type than the type received in the previous visit, then the cast will default to a `dynamic_cast`. After a default `dynamic_cast` is executed, new type data will be cached in the MemCache object under the assumption that the next cast visit will be stable.

5.3.1 MemCache Objects

To create a memoisation technique for the dynamic cast operator, we need a data structure to cache the required data. The data structure used (and the data itself) must be carefully chosen to limit overheads. A naive approach that we attempted initially was to use splay trees [116]. A splay tree is a self-balancing binary tree that will rearrange itself so that the most recently accessed element will be the quickest to access again. Our initial idea was to link a splay tree to each cast site so that MemCast could dynamically add each new source type to the tree and easily access the most recently used. The thought was that splay trees would benefit high-stability casts and speed up some low-stability casts, as the self-balancing mechanism could aid fast access to other seen source types. This, however, was not the case. We found that the cost of balancing the splay tree after every use was on par with the cost of dynamic casting. From this, we deduce that any search or management of a data structure must be kept to a minimum. Thus came our final storage design, a single data set per cast site that stores its last seen source type. This simple idea provides direct access to the cached data, simplifies cache management to a straightforward overwrite, and removes the need for an expensive search technique.

One MemCache Per Cast MemCast takes advantage of the fixed target types used in dynamic casting by assigning each cast location its own MemCache object, thus creating many light-weight cache objects. Each MemCache object is

used solely to store source-type information for its associated cast site. During a casting operation, the MemCast function will receive its MemCache object as a parameter, and as these objects will be declared as global, their address location is known at compile-time and can be hard coded into the MemCast function call. This combination essentially creates a hard-coded mapping between source type information (within the MemCache object) and dynamic cast sites using the MemCast wrapper.

MemCache Data-Structure We questioned how many entries a MemCache object should store. Should it store the data of all the previous dynamic cast source types? Or is it enough to store just a few? We trialled several options, starting with splay trees [116] to store all seen source types (as previously discussed) and finished with a single MemCache entry storing only the last seen source type. It was the single MemCache entry that produced the best results. With a single MemCache entry, we store only the data from the last dynamically cast object and discard any previous data. Although there is an obvious cost to only caching one cast result, as every new source type will default to a dynamic cast, in truth, we found that the upkeep and search of any multi-entry data structure was more expensive than an occasional default dynamic cast; this was because of the high stability of our cast sites.

Two Words in Size To further optimise MemCast function calls, MemCache objects were limited to two machine words to better fit the machine cache-lines and reduce cache misses. The **first word stores the vptr** of the last source object that defaulted to a dynamic cast, and the **second word stores the offset displacement**, if an address-point displacement occurred.

Why Virtual Pointers? Recall from Section 2.4.4 that the physical addresses of RTTI objects are used as type identification keys for run-time objects. That is

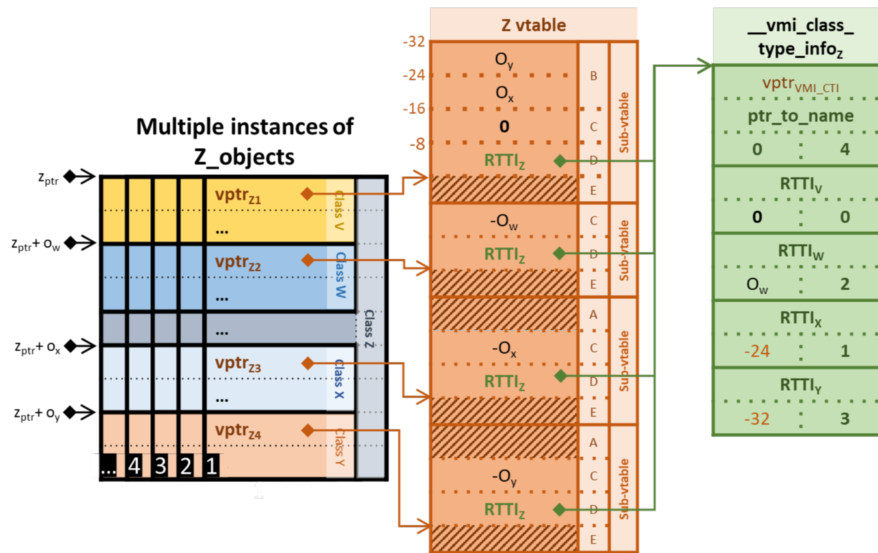


Figure 45: Virtual pointers vs RTTI pointers

to say, if two run-time objects address the same RTTI object from their vtables, then those objects have the same type. One might wonder why RTTI addresses are used rather than vtable addresses (i.e. vptrs)? Well, the reason for this can be seen in Figure 45. In this example, we depict multiple instances of a class `Z`. These `Z` objects display both multiple and virtual inheritances, as they contain four sub-object instances from classes `V`, `W`, `X`, and `Y`. Each sub-object instance contains its own vptr, which addresses the relevant sub-vtable within the complete `Z` vtable. From these four vtable entry points, the same global RTTI address can be accessed at an offset of `-8`. So no matter the address-point used to access a `Z` object, the corresponding vptr has access to the same RTTI address for `Z`, stored at the location `vptr-8`. This provides a constant-time mechanism for run-time type-checking. Using a `Z` object's vptr as its type identification key would necessitate checking the key was drawn from a set of four valid vptrs, which would entail a search. Alternatively, one might consider bounds checking, i.e. if $\texttt{vptr}_{Z1} \leq \texttt{vptr}_{?} \leq \texttt{vptr}_{Z4}$, but this assumes that all sub-vtables appear consecutively in memory, which is not guaranteed.

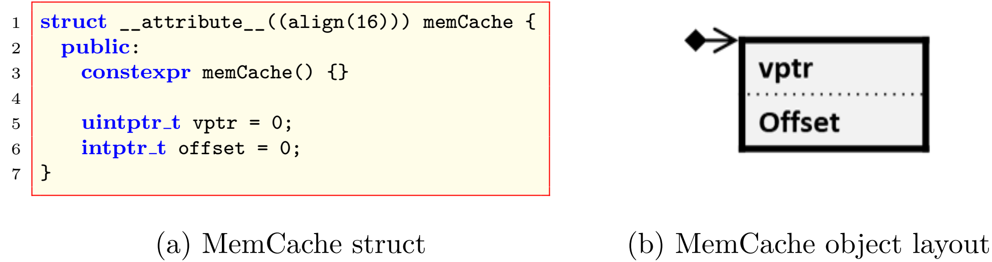
Although vptrs are not appropriate as a type identifier in the `dynamic_cast`

mechanism, they happen to be perfectly suited for MemCast. A vptr is both unique to a class type and unique to a sub-object within that type. For example, we can see that each sub-object in `Z` (Figure 45) has its own unique vptr. Therefore we can deploy a vptr as an **address-point identifier**. This is useful, as a cast operation is merely an offset adjustment from one address-point to another. For example, if we perform a cast to a `Z*` from any address-point in a complete `Z` object, then the following displacements would occur from each sub-object:

From sub-object	vptr	Displacement
V	$\mathtt{vptr}_{\mathtt{Z1}}$	0
W	$\mathtt{vptr}_{\mathtt{Z2}}$	$\mathtt{-o}_{\mathtt{W}}$
X	$\mathtt{vptr}_{\mathtt{Z3}}$	$\mathtt{-o}_{\mathtt{X}}$
Y	$\mathtt{vptr}_{\mathtt{Z4}}$	$\mathtt{-o}_{\mathtt{Y}}$
Z (complete object)	$\mathtt{vptr}_{\mathtt{Z1}}$	0

Notice that we can create a mapping between vptrs and displacements: cast sites have fixed target types, so the displacement from a vptr location to a fixed target is always the same. This is the fundamental premise behind MemCast and its MemCache objects.

MemCache object Structure for Optimal Execution Most modern processors are 64-bits and use 64-byte cache lines (at least for all current Intel processors [59]). With this in mind, we designed the MemCache objects for optimum performance on such machines. Figure 46a lists the source code for our MemCache objects, which are defined by a struct called `memCache`. The struct is forcibly aligned at 16-bytes (line 1) and contains two data members, `vptr` (line 5) and `offset` (line 6), of types `uintptr_t` and `intptr_t`, respectively. These two types are fixed-width integer types (one unsigned and one signed) capable of holding




```
1 struct __attribute__((align(16))) memCache {
2   public:
3     constexpr memCache() {}
4
5     uintptr_t vptr = 0;
6     intptr_t offset = 0;
7 }
```

(a) MemCache struct (b) MemCache object layout

Figure 46: MemCache structure and object layout

Target Type (T)	Source Object (s)	Successful Cast Result	Failed Cast Result
Pointer	Pointer	Pointer	Pointer to Null
Reference	Reference	Reference	Invoke (std::bad_cast)

Table 9: Type specifications of the dynamic cast operator

an 8-byte pointer. As these are the only data members, `memCache` objects are 16-bytes in size (as seen in Figure 46b). With a size and forced alignment of 16-bytes, `memCache` objects are aligned with cache lines, which are also 16-byte aligned [53]. The force of this is that `memCache` objects never straddle two cache lines, which reduces the number of cache misses (as we found during development).

Furthermore, we set the `memCache` constructor to a `constexpr`. The `constexpr` keyword will force the construction and initialisation of `memCache` objects to occur prior to execution (if possible). Shifting construction to compilation improves performance further.

5.3.2 Dynamic Cast Wrapper

The dynamic dispatch mechanism will accept pointer and reference types as a target, but each type of target is resolved differently. The C++ standard stipulates, what the source type should be, based on the target type of a dynamic cast, and the result from both a successful and unsuccessful cast. Table 9 summaries these expectations for the expression `dynamic_cast<T>(s)`, where `T` is the target type and `s` is the object.

5.3.2.1 Enforcing Pointer and Reference Cast Expectations

Suppose a variable is assigned to a reference or a pointer to an object. We will refer to the pointer or reference types as **pr-types** to make a clear distinction between them and the type of the object itself. As dynamic casts process pr-types differently, our wrapper must do the same, which means it requires multiple definitions. To create multiple definitions based on a target's pr-type, we use multiple class templates, each containing a single member function template (listed in Figure 47).

Figure 47 presents three template specialisations, one for each possible target pr-type: a pointer (`target*`), an lvalue reference (`target&`), or an rvalue reference (`target&&`). The two-layer template approach, using both a class and member function template, is necessary to allow for pr-type manipulation within the function body. For example, if we call the following MemCast function:

```
memCast_Resolver<V&>::memCast(c,a);
```

for a target `V&`, a source type `A a`, and a `memCache` object `c`, the second template in Figure 47 will be instantiated. If we look at the class template signature:

```
template <class target> class memCast_Resolver<target&>
```

the parameter in the class name `memCast_Resolver<V&>` matches the pr-type of the class signature name. However, the class template parameter, defined as `template <class target>`, has no pr-type assigned to it. This allows us to use the template parameter (`target`) in the function body with our own choice of pr-type. This is especially useful for reference targets, as they must be converted from a referenced pr-type to a pointer pr-tyre to perform pointer arithmetic (required for successful casts).

The class template that wraps the `memCast` function is also necessary, because if the function was defined without this class template:

```
template<class target, class source> static inline
 target memCast( memCache& cache, source& objSrc)
```

```
1  template <class target> class memCast_Resolver<target*>{
2      template<class source > static inline target*
3      memCast(memCache &cache, source* objSrc) { ... }
4  };
5
6  template <class target> class memCast_Resolver<target&>{
7      template<class source > static inline target&
8      memCast(memCache &cache, source& objSrc) { ... }
9  };
10
11 template <class target> class memCast_Resolver<target&&>{
12     template<class source > static inline target&&
13     memCast(memCache &cache, source& objSrc) { ... }
14 };
```

Figure 47: Template Specialisations for `MemCast_Resolver`

and called using:

```
memCast<V&>(c,a);
```

then the template parameter `target` is fixed with a reference pr-type, making it impossible to convert to a pointer, and in turn, perform pointer arithmetic.

Whenever a MemCast function is called in the source code, the compiler will only instantiate one of the template specialisations in Figure 47; but only if the source and target pr-types passed to that function match one of the pr-types outlined in template signatures. If pr-types do not match, a compile-time error will occur. For example, the following MemCast expression:

```
A* a;
memCast_Resolver<V&>::memCast(c,a);
```

throws a compile-time error, just like it would for a dynamic cast (`dynamic_cast<V&>(a)`), as it is trying to cast a reference type to a pointer, which is not allowed by the standard.

Targeting References vs Pointers When targeting reference types, the MemCast function is straightforward; if the source object's vptr matches the cached vptr (in the assigned MemCache object), we can perform a fast pointer adjustment using the data stored in the MemCache object. If the vptrs are unequal, then the

cast defaults to a `dynamic_cast`. If the `dynamic_cast` fails, then a `std::bad_cast` exception is invoked.

MemCast needs only apply one form of memoisation for reference targets, but there are two opportunities for memoisation for pointer targets. The first, like for references, should handle successful casts by performing fast pointer adjustments; but the second should catch unsuccessful casts, where a null pointer is returned. From our Deal.II analysis, we found that 94 cast sites (out of the 459 sites tested) returned a null pointer during execution. Of these 94 cast sites, 85 showed 100% stability, and 88 were at least 80% stable. We cache both behaviours, not only motivated by our own work, but in dependant work that suggests a high null return rate in other software [100]. Both path-ways require a vptr match; therefore, MemCast's checking technique (for pointer targets) needs to distinguish between vptrs that result in pointer adjustments and vptrs that result in null pointers. To do this, we introduce vptr flags.

Vptr Flags We can place flags within vptrs without introducing extra boolean variables. A vptr addresses a vtable at a given address-point. The address-point of a vtable is always the address immediately after the RTTI (pointer) data member, which is also the address of the first virtual function pointer, if one exists (Section 2.4.2). Pointers themselves are 8-byte aligned, hence, the address-point used to access the vtable must be an 8-byte aligned address. This, in turn, means the address stored in all vptrs are 8-byte aligned. When represented at the binary-level, an 8-byte aligned address is guaranteed to end with 000. Therefore, these three bits can be repurposed as flags (Figure 48) when caching vptrs in the MemCache objects. By using these three bits, we can retain the minimal 16-byte size (2 words) of the MemCache objects (Section 5.3.1).

The vptr flags are used within the `memCache` objects as part of their cached vptr. When a MemCast function defaults to a dynamic cast, the result of the

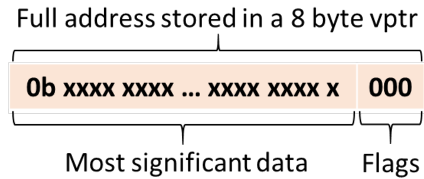


Figure 48: Vptr bit flags

Vptr flags	**Meaning**
`001`	Pointer adjustment vptr
`011`	Null pointer vptr
`000`	No vptr assigned

Table 10: Vptr Flag Meanings

`dynamic_cast` operator is checked to determine whether it is a null return or not. In each case, the source object's vptr is altered using a bit-wise OR operation, altering the appropriate bits to distinguish which result was received. The modified vptr is then cached in the `memCache` object, providing an unambiguous log of the return result from the previous cast.

The meanings behind each set of vptr flags are described in Table 10. Bit 0 (the least significant bit) indicates whether the `memCache`::vptr data member is zero. The vptr data member is initialised to zero and will remain zero until used within a MemCast function call. Bit 1 indicates the result of the previous cast, where 0 is a pointer adjustment, and 1 is a null return. Bit 2 goes unused.

5.3.2.2 MemCast Function Implementations

The MemCast Vptr Checking Algorithm Once a MemCast function is called, the source object's vptr must be checked against the MemCache vptr to determine which optimised cast can be performed, if any. Algorithm 1 presents the MemCast vptr checking algorithm, which receives both the source object vptr ($\texttt{so}_{\texttt{vptr}}$) and the MemCache vptr ($\texttt{mc}_{\texttt{vptr}}$) as parameters. If $\texttt{mc}_{\texttt{vptr}}$ stores a non-zero value, then the `memCache` object has been used in a previous MemCast call, and the value stored in $\texttt{mc}_{\texttt{vptr}}$ will be a vtable address modified to end in either a `001` or `011`. Lines 3 and 5 of the algorithm perform bit-wise OR operations followed by a vptr pointer comparisons. Only when the vtable addresses (stored in the most significant parts of the vptr) AND the vptr flags match will either `if`

Algorithm 1: MemCast Vptr Check

```
1 mc_vptr ← memCache vptr;
2 so_vptr ← source object vptr;
3 if mc_vptr == (so_vptr|1) then                /* so_vptr ends in 001 */
4 |   pointer adjustment
5 else if mc_vptr == (so_vptr|3) then           /* so_vptr ends in 011 */
6 |   return NULL pointer ;                     /* Pointer targets only */
7 else
8 |_  default to dynamic_cast
```

statement be exercised. If there is no match, then the function will default to a dynamic cast.

This checking technique also provides a level of security by enforcing a default dynamic cast call with any new `memCache` object. When a `memCache` object is initialised, its vptr is set to zero and will stay as zero until it is updated by a default dynamic cast. Initialising a vptr to zero ensures that neither if-branch is selected on an unvisited MemCache object. To see this, suppose that our MemCast function does not safeguard against unvisited MemCache objects. Also, suppose that an object had been deliberately mutated to contain a zero value where a vptr would usually be stored. Then, if such an object were passed to an unvisited MemCast site, the zero in the MemCache object would match the zero in the perturbed object. This would result in a successful MemCast result, providing an opportunity for a type confusion attack, as it would circumvent the default dynamic cast.

MemCast Function For Pointers Figure 49 provides the implementation of the `memCast` function (within the `memCast_Resolver` template) for a `target*` type. The function itself (line 12) receives two parameters; the first is the `memCache` object (called `cache`) associated with the cast location, and the second is the source object (called `objSrc`). The function body (lines 13-18) performs the vptr check described in Algorithm 1. To simplify access to the source object's vptr,

```
enum VPTR_RESULT_FLAG {
    VPTR = 1,       // 001
    NULL_PTR = 3    // 011
};
...
class obj {
    uintptr_t vptr;
}
...
template <class target> class memCast_Resolver<target*> {
    template<class source > static inline target*
    memCast(memCache &cache, source* objSrc) {
        if(cache.vptr == ((((obj*)objSrc)->vptr) | VPTR)) {
            return (target*)((char*)objSrc + cache.offset);
        } else if(cache.vptr == ((((obj*)objSrc)->vptr) | NULL_PTR)) {
            return NULL;
        }
        return memCast_dynamicCast<target*,source>(cache, objSrc);
    }
};
```

Figure 49: MemCast Function for Pointer Targets

we cast this object to a dummy class called `obj`, which has a single data member of type `uinptr_t`. (The C++11 `uinptr_t` data type is convenient in this setting because it specifies an unsigned integer guaranteed to match the size of a pointer, allowing conversion back and forth without truncation.) The cast operation is a C-style cast, which incurs no type-checking or additional assembly instructions once compiled (Section 3.4.1), therefore incurring no additional run-time cost. In addition the `vptr` field resides at the zero offset, hence the data member access does not require any pointer adjustment.

The purpose of the casts (in lines 13 and 15) is to bypass static type-safety checks and assign the object's vptr to the correct type needed for the bit-wise operation and comparison. If a vptr match occurs from either comparison, the function will return a new pointer derived from some simple pointer arithmetic (line 14) or a null pointer (line 16). If the comparisons are unsuccessful, then the call defaults to a dynamic cast. This default dynamic cast is another wrapper function, called `MemCast_dynamicCast` (line 18), which houses the standardised `dynamic_cast` call. This function is responsible for calling the `dynamic_cast` operator and updating the associated `memCache` object.

```
21 template< class target,class source >
22 target memCast_dynamicCast(memCache &cache, source* ptr) {
23     target tar = dynamic_cast<target>(ptr);
24     if(tar) {
25         cache.vptr = (((obj*)ptr)->vptr) | VPTR;
26         cache.offset = (uintptr_t)((char*)tar - (char*)ptr);
27     } else {
28         cache.vptr = (((obj*)ptr)->vptr) | NULL_PTR;
29     }
30     return tar;
31 }
```

Figure 50: MemCast's default Dynamic Cast Wrapper

```
32 template <class target> class memCast_Resolver<target&> {
33   template<class source> static inline target&
34   memCast( memCache &cache, source& objSrc) {
35     if(cache.vptr == ((((obj*)&objSrc)->vptr) | VPTR)) {
36       return *((target*)((char*)&objSrc + cache.offset));
37     }
38     target* tar = memCast_dynamicCast<target*, source>(cache, &objSrc);
39     if(!tar)
40       throw std::bad_cast();
41     return *tar;
42   }
43 };
```

Figure 51: MemCast function for lvalue references targets

Defaulting to a Dynamic Cast When MemCast defaults to a dynamic cast, it must record the results and cache them for later use. Figure 50 provides the source code for a `MemCast_dynamicCast` wrapper function that caches data in the `memCache` object provided. The first line of the function body (line 23) performs a standard dynamic cast and stores the result in the variable `tar`. Following this, an `if` statement is executed, determining what is cached in the `memCache` object. Lines 25 and 26 are executed if the dynamic cast was successful, and line 28 if unsuccessful (i.e. `tar` is a null pointer). In both outcomes, the object's vptr has its vptr flags set appropriately (enum `VPTR = 1` or `NULL_PTR = 3`) before being cached in the `memCache` object. Line 26, which is only executed for successful casts, calculates the offset displacement from the source address-point to the newly cast target address-point and caches the result.

MemCast Function For References Figure 51 provides the implementation of the `memCast` function (within the `memCast_Resolver` template) for a `target&` type. As casts to reference types produce a `std::bad_cast` exception when they fail, there is no value in caching null returns for reference types as an exception needs to be thrown immediately anyway. Therefore the body of the `memCast_Resolver` does not handle a null return. Instead, if the vptr comparison (line 35) fails, a default dynamic cast is executed and returns the result as a pointer type (line 38). If the result is null (line 39), the `std::bad_cast` exception is executed (line 40); otherwise, the result is dereferenced and returned (line 41). The `MemCast_Resolver` function implementation for rvalues references (`Target&&`) follows this same pattern.

5.4 Experimental Results

In this section, we evaluate the performance of MemCast compared to both static and dynamic casting, providing insight into the actual cost of a dynamic cast, demonstrating the superiority of MemCasting. Later, we demonstrate the impact MemCast has on a real-world program by applying MemCast to the Deal.II library. All experiments were performed on a 1.9GHz 64-bit Intel i7 UNIX-base machine with 16GiB of DDR4 SDRAM, using code compiled by the Clang 11.0.0 compiler. The Clang compiler conforms to the Itanium ABI [23] and C++ standard [57] and uses the GNU's libstdc++ standard library implementation [44], which provides the dynamic cast implementation.

5.4.1 The True Cost of Casting

5.4.1.1 Straight-Line Fitting Method

The time it takes to execute a cast must be measured in nanoseconds, and it can be challenging to accurately capture such timings due to naturally occurring noise.

To address this, we use Moreno and Fischmeister's straight-line fitting technique [89], which was explicitly designed to capture the time of small operations in the context of noise.

Moreno and Fischmeister's technique gathers a set of points (x_i, t_i), where t_i is the recorded execution time of an operation performed x_i times consecutively. From these points, the method estimates a line of best fit $t = ax + b$, where the gradient of the line, a, is the estimated execution time for a single cast, and the y-intercept, b, is the systematic (reproducible) error.

The accuracy of this technique comes from its ability to eradicate systematic errors and minimise non-systematic errors. Systematic errors, like the cost of calling time-management functions, are constant overheads across all tests. Non-systematic errors, like the execution noise of the machine (the impact of other processes, peaks and troughs in processor performance, etc.), are almost random effects that change from run to run. By calculating the line of best fit, the systematic error is removed from the time estimate and shifted into the t-intercept of the linear equation. Non-systematic errors are minimised by purging any data point anomalies, i.e. removing any outliers that significantly deviate from the line of best fit. Once removed the line of best fit is recalculated, ignoring the anomalies.

In our application of straight-line fitting, the operation performed is a single cast at a fixed call site. We choose to record the execution time of that cast (static, dynamic or MemCast) 200 times, where x_i varies between 1 and 5, resulting in a set of 1000 points: $(1, t_1), \ldots, (5, t_5); (1, t_6), \ldots, (5, t_{10}); \ldots; (1, t_{996}), \ldots, (5, t_{1000})$. From these points we calculate the line of best fit.

5.4.1.2 Testing Hierarchies

The dynamic cast operator is realised as a virtual member function in the RTTI hierarchy. Each type of RTTI used to represent an object type (recall the types `si_class_type_info` and `vmi_class_type_info` from Section 2.4.4) has its own

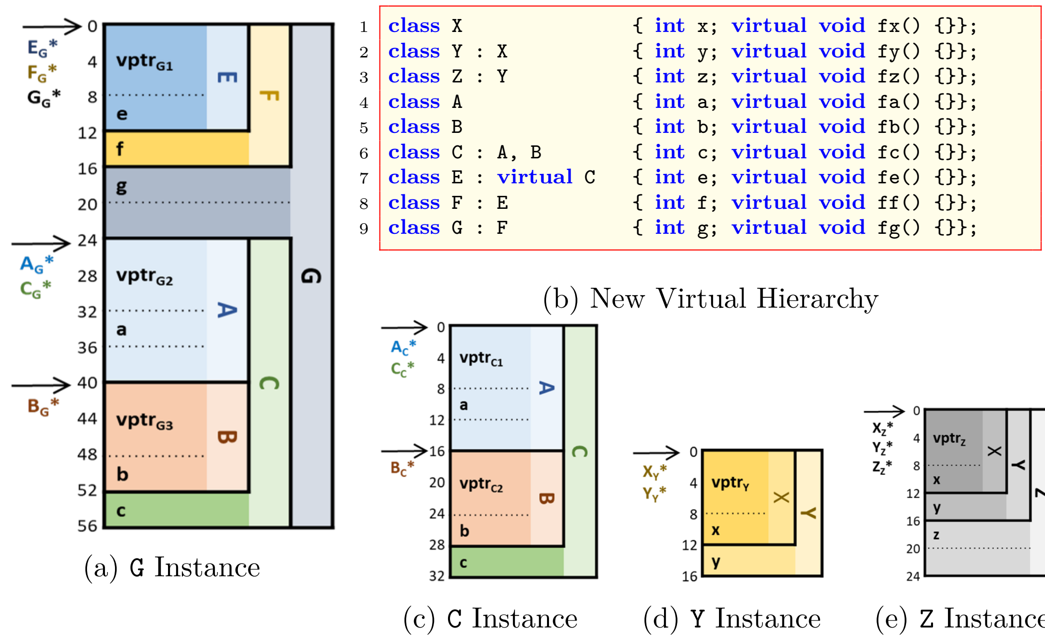


Figure 52: New Hierarchy and Object Examples with Address-Points

implementation of dynamic cast with a different performance profile. As we want to quantify the cost of each type of casting technique, we introduce two hierarchies to exercise a variety of casting scenarios and reproduce the two different RTTI data types.

Figure 52 lists the source code of the two hierarchies we will use for cast testing, alongside some complete object layouts. Classes `X`, `Y`, and `Z` are part of a single inheritance hierarchy, so produce `si_class_type_info` RTTI objects, and classes `A-G` constitute a multiple virtual inheritance hierarchy, producing both `vmi_class_type_info` and `si_class_type_info` RTTI objects. We aim to find the minimum cost of dynamic down-casting, and to this end, each hierarchy is deliberately small. We do not attempt to find the maximum cost of a dynamic cast, as the cost of such casts is intrinsically linked to the distance between the type of the target and the dynamic type of the source object within the RTTI structure. As class hierarchies technically have no bounds, the same can be said

RTTI	Cast Decription		Types Used	Stability	Casting Technique Times (ns) Static	Dynamic	MemCast
si_class_type_info	**Up-Cast**		$Z_Z* \rightarrow X_Z*$	100%	0.00	0.00	1.03
	Down-Cast	(PTC)	$X_Z* \rightarrow Z_Z*$	100%	0.00	7.59	1.03
		(STC)	$Y_Z* \rightarrow Z_Z*$	100%	0.00	7.59	1.23
		(PTS)	$X_Z* \rightarrow Y_Z*$	100%	0.00	14.90	1.00
	Unrelated (Null return)		$X_Y* \rightarrow Z_Z*$	100%	-	14.80	1.49
vmi_class_type_info	**Up-Cast**	(SL)	$C_C* \rightarrow A_C*$	100%	0.00	0.00	1.03
		(DL)	$C_C* \rightarrow B_C*$	100%	0.37	0.32	1.01
	Down-Cast	(SL)	$A_C* \rightarrow C_C*$	100%	0.00	12.20	1.00
		(DL)	$B_C* \rightarrow C_C*$	100%	0.54	9.96	1.17
	Unrelated (Null return)		$B_C* \rightarrow G_G*$	100%	-	8.55	1.48
	Virtual		$B_G* \rightarrow F_G*$	100%	-	23.90	1.00
	Alternating (Two casts performed)		$A_C* \rightarrow C_C*$ $B_C* \rightarrow C_C*$	0%	-	20.40 (avg. 10.20)	30.20 (avg. 15.10)

Average Successful MemCast (100% Stability)	1.060ns
Average Successful Null return MemCast (100% Stability)	1.485ns
Average MemCast Cache Time	4.675ns
Average Cost of a Virtual Function Call	1.200ns

Table 11: Average cost of casting per execution, for several casting scenarios

about their RTTI data-structures and, in turn, the cost of dynamic casting.

5.4.1.3 Results

Table 11 shows the results of each test performed. The first four columns describe the casting scenario, where each row distinguishes its RTTI structure, target and source types, and the stability of the cast. The last three columns give timings in nanoseconds for three casting techniques: static, dynamic, and MemCast.

To succinctly describe the source and target types of each cast, we introduce the abbreviation $B_C* \rightarrow C_C*$ to describe the following cast:

```
dynamic_cast<C*>(b); //where B* b = new C();
```

This casts `b`, which points to the `B` sub-object within a complete `C` object, to a `C*` pointer, resulting in a pointer to the complete `C` object. To amplify, observe that the `C` instance given in Figure 52c has two address-points. The top arrow marks the address-point of the complete object, which can be accessed using a pointer of type `A*` or `C*`. The bottom arrow gives the address-point of the `B` sub-object, which is addressed using a pointer of type `B*`. Notice how the address-point of the

`B` sub-object within the complete `C` object is expressed in the notation $\texttt{B}_{\texttt{C}}\texttt{*}$. Put another way, $\texttt{B}_{\texttt{C}}\texttt{*}$ can be read as B-in-C, that is, the address of the `B` sub-object in the complete `C` object. The effect of the cast $\texttt{B}_{\texttt{C}}\texttt{*} \rightarrow \texttt{C}_{\texttt{C}}\texttt{*}$, therefore, is to adjust the lower address-point ($\texttt{B}_{\texttt{C}}\texttt{*}$) to the upper address-point ($\texttt{C}_{\texttt{C}}\texttt{*}$). More specifically it is casting to the complete `C` object (C-in-C), not the `A` sub-object (A-in-C).

All casts in Table 11 are executed on one or more of the objects depicted in Figure 52, where the abbreviation of each cast tallies with the address-points given in the figure.

Single Inheritance Results We performed five different casts for the single inheritance hierarchy (classes `X-Z`), which uses the `si_class_type_info` RTTI data-structure (top half of Table 11). As this is a single inheritance hierarchy, all objects have a single address-point at their zero offset (Figure 52d and 52e). With only one address-point, casts will not perform pointer adjustment, which is why static casting has no overhead. The same can be said for the dynamic up-cast, which the compiler optimises to a static cast, which again, incurs no pointer adjustment. For down-casts, we performed three tests: (PTC) Primary to Complete, casting from the primary class sub-object to the complete object; (STC) Secondary to Complete, casting from a secondary sub-object (simply meaning not the primary object) to the complete object; and (PTS) Primary to Secondary, casting from the primary class to a secondary sub-object (not the complete object). The final cast is to an unrelated object, forcing a null return. The fastest form of dynamic casting (7.59ns) is any cast that targets the complete type: PTC or STC, which correlates with our analysis of the `__dynamic_cast` function's source code that is optimised for such cases (using the static hint `src2dst`, see Section 3.4.4). Casts that target a secondary derived class (PTS) or an unrelated class are slower because RTTI traversal (beyond the head of the RTTI data structure, at least) is required.

Multiple Virtual Inheritance Results For the multiple virtual inheritance hierarchy (classes `A-G`), which uses the `vmi_class_type_info` RTTI data-structure, we performed seven different cast tests (see the bottom half of Table 11). The instances of classes `C` and `G` have multiple address-points (see Figure 52a and 52c), so some casts will perform a pointer adjustment and a type check. The up and down-casts in these tests will, respectively, change the address-point to either a different location (DL) or retain the same location (SL). For casts performed statically (for which the pointer adjustment is fixed and known at compile-time) we can now observe the actual cost of a pointer adjustment (0.37ns and 0.54ns for the DL casts).

As well as performing an unrelated dynamic cast in the setting of a single inheritance hierarchy (`si_class_type_info`), it was also timed for multiple inheritance (`vmi_class_type_info`). Rather surprisingly, the latter is faster than former: 8.55ns compared to 14.8ns. This not only reflects the different implementations of the virtual `_do_dynCast()` function (see Section 3.4.4) but also suggests further optimisation techniques are used for unrelated source types within the `vmi_class_type_info::_do_dynCast()` definition.

We also performed a virtual inheritance cast, where we chose the source and target types to induce a particularly long traversal of the RTTI data structure, which is reflected in its performance time (23.90ns).

Notice that the relationship scenarios of these casts, and all other casts discussed so far, have little to no effect on MemCast timings. MemCast timings appear consistent for both related and unrelated casts. However, when we introduce an alternating cast scenario, which alternates the source type it received with every visit, MemCast's performance overhead is high (averaging 15.10ns per cast). Note that the alternating cast test captures the cost of performing two casts back-to-back, so it is halved to get the average cost of a single cast. This provides us with a 0% stability scenario, exercising the worst case for MemCast.

Dynamic Cast From the results in Table 11, we can estimate that dynamic casting is at least 18 times slower than static casting. This value was calculated using the only comparable casting results, the DL static and dynamic down-casts. It is not possible to quantify this multiplier for other types of casting scenarios because, in most cases, static casting costs nothing. We have, however, determined the minimum cost of dynamic down-casting to be **7.59ns**, when casting to complete objects, using a `si_class_type_info` RTTI data-structure (used in single inheritance).

MemCast The speed of MemCast depends on its outcomes: a successful MemCast that returns an object pointer without defaulting to a dynamic cast, averages **1.060ns**; a successful MemCast that returns a null pointer without defaulting, averages **1.485ns**; and a failed MemCast that defaults to a dynamic cast, which takes the time of the dynamic cast (which is variable) plus **4.675ns** (the time for vptr checking with caching, which is constant).

Though 4.675ns is not small, this overhead is acceptable because a default cast is incurred infrequently. For context, the average run-time of a virtual function call was found to be **1.200ns**. Hence the performance of a successful MemCast compares to that of a virtual function, which is regarded as an acceptable expense.

5.4.2 Evaluation of MemCast's Capabilities

The experimental results in Table 11 show that significant savings can be made by swapping stable dynamic casts with the MemCast wrapper function. However, it also showed that unstable casts are costly, as they add a constant overhead to an already expensive dynamic cast. So, the question is now, what degree of stability is required for MemCast to outperform dynamic casting? Or more precisely, what is the minimum stability that guarantees improved performance? To answer these questions we will look specifically at successful MemCasts that result in an object

pointer (as this was the most prolific scenario on our Deal.II experiments) and compare it with the fastest form of dynamic casting.

5.4.2.1 MemCast Definitions and Equations

To reason about the relative cost of MemCast and dynamic cast, we introduce formulae that aid their modelling. The following formula decomposes the cost of a unsuccessful MemCast into its constituent parts:

Definition 5.4.1 (Unsuccessful MemCast Formula)**.** *Let T_{dyn} be the time taken for a dynamic cast and T_{cache} be the time taken for the MemCast function to perform a vptr check and data cache. Then:*

$$M_{default} = T_{dyn} + T_{cache}$$

is the estimated cost of a single default MemCast (a default to the `dynamic_cast` *operator plus caching the result).*

Building on this we can estimate the cost of using MemCast:

Definition 5.4.2 (MemCast Time Estimation Formula)**.** *Let the time of a successful MemCast be $M_{success}$, let S denote the stability and x denote the number of visits to a cast site. Then:*

$$M(x, S) = M_{default} + S(M_{success}(x - 1)) + (1 - S)(M_{default}(x - 1))$$

is the estimated time for a run of $x \geq 1$ MemCast calls, where $0 \leq S \leq 1$.

The above formula estimates the run-time of a MemCast based on its stability. The leading $M_{default}$ term reflects the default cast incurred in the initial visit to a MemCast site. For completeness, we give the time for a run of dynamic casts:

Definition 5.4.3 (Dynamic Down-Cast Time Formula)**.** *Let* T_{dyn} *be the average cost of a dynamic cast at a particular cast site. Then:*

$$D(x) = T_{dyn}x$$

is the estimated time for a run of x *dynamic casts.*

5.4.2.2 Minimum Stability

There is no value in applying MemCast to dynamic up-casts, as these are reduced to static casts. Therefore our comparison focuses on dynamic down-casting only. In particular we consider single inheritance down-casting (PTC and STC), since these are the most competitive forms of dynamic down-casting. To make the comparison, we instantiate Definition 5.4.3 using the timings from Table 11 to give:

$$D(x) = 7.590x$$

since $T_{dyn} = 7.590$ is the time of a PTC and STC single inheritance dynamic cast. Likewise, instantiating Definition 5.4.1 with $T_{cache} = 4.675$ gives:

$$M_{default} = 12.265$$

Finally, we instantiate Definition 5.4.2 with $M_{success} = 1.060$ to give:

$$M(x, S) = 12.265 + S(1.06(x - 1)) + (1 - S)(12.265(x - 1)). \quad (2)$$

Figure 53 plots $M(x, S)$ and $D(x)$ against x for various S values. Observe that for $S = 0$, $M(x, S) > D(x)$ for all $x \geq 1$ (i.e. MemCast is always slower than dynamic casting when the stability of that cast is 0%). Conversely, for $S = 1$,

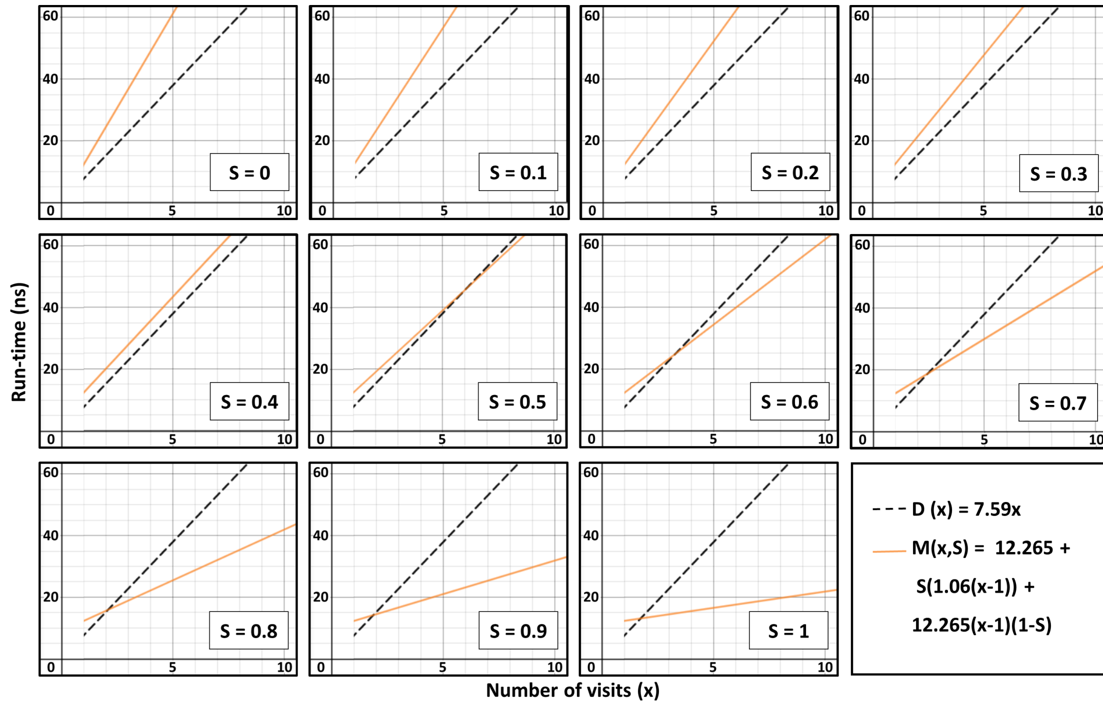


Figure 53: How stability changes the estimated cost of MemCasting

$M(x, S) < D(x)$ for all $x \geq 2$ (i.e. MemCast is always faster than dynamic casting when the cast site has 100% stability and is visited at least twice). Hence there exists some critical S value, $0 < S^* < 1$, such that for any $S > S^*$ there exists a sufficiently large $x > 1$ for which $M(x, S) < D(x)$. S^* is critical in the sense that there is no smaller value of S with this property, so we call this value **minimum stability**.

To observe the impact of increasing S, we can calculate the difference between $M(x, S)$ and $D(x)$ for two different values of S. To illustrate, consider $S = 0.6$ and $S = 0.8$, and their corresponding graphs in Figure 53. Observe that for $x = 8$, $D(x) > M(x, S)$ for both $S = 0.6$ and $S = 0.8$. But, $D(8) - M(8, 0.6) = 60.72 - 51.06 = 9.66$ and $D(8) - M(8, 0.8) = 60.72 - 35.37 = 25.35$, hence $D(8) - M(8, 0.6) < D(8) - M(8, 0.8)$. Therefore the difference between the run-times of dynamic casting and MemCasting grows as S increases: the higher the stability (S) the greater the speedup.

Calculating Minimum Stability Observe that when $S = S^*$, the gradient of $M(x, S^*)$ and $D(x)$ are equal (i.e. their lines become parallel for some S close to 0.4, see Figure 53). This provides a tactic for calculating S^*. To this end, we rearrange $M(x, S^*)$ as:

$$M(x, S^*) = (M_{default} + S^*(M_{success} - M_{default}))x - S^*(M_{success} - M_{default})$$

Equating the gradients of $M(x, S^*)$ and $D(x)$ and rearranging for S^* gives

$$T_{dyn} = M_{default} + S^*(M_{success} - M_{default}) \text{ and } S^* = \frac{M_{default} - T_{dyn}}{M_{default} - M_{success}} \tag{3}$$

Therefore, we can infer for any

$$S > S^* = \frac{M_{default} - T_{dyn}}{M_{default} - M_{success}} \tag{4}$$

$M(x, S) < D(x)$ for some (x) number of visits to the cast site. That is, a cast site with a stability $S > S^*$, is guaranteed to execute faster with MemCasting, given enough visits.

Using equation (3), and our observed values ($T_{dyn} = 7.590$, $M_{success} = 1.060$, and $M_{default} = 12.265$), we can calculate S^* as:

$$S^* = \frac{12.265 - 7.590}{12.265 - 1.060} = 0.417$$

Thus the minimum stability (S^*) is 41.7%.

For context, 98.7% of all cast sites in Deal.II are at least 55% stable (which exceeds 41.7%), suggesting that the run-time of almost all casts could be improved with MemCasting. Furthermore, 91% of Deal.II's cast sites are at least 80% stable, and recall that the performance of MemCast increases with S.

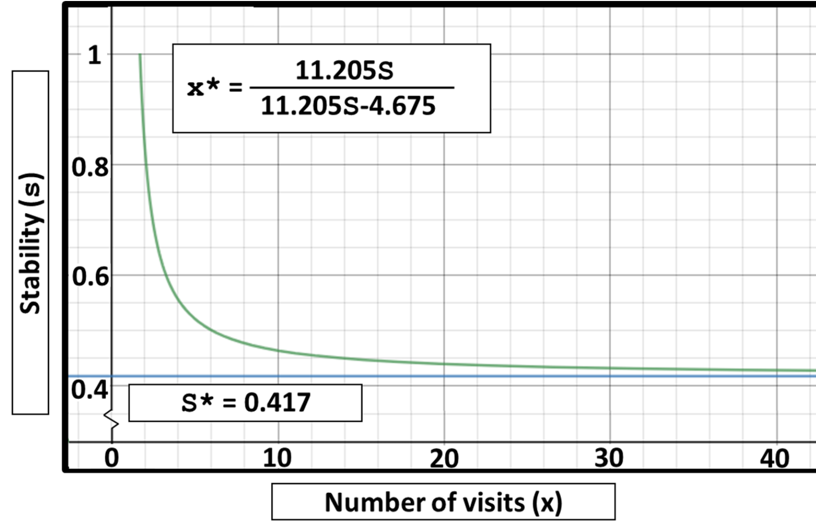


Figure 54: Minimum MemCast visits for for any $S^* < S \leq 1$

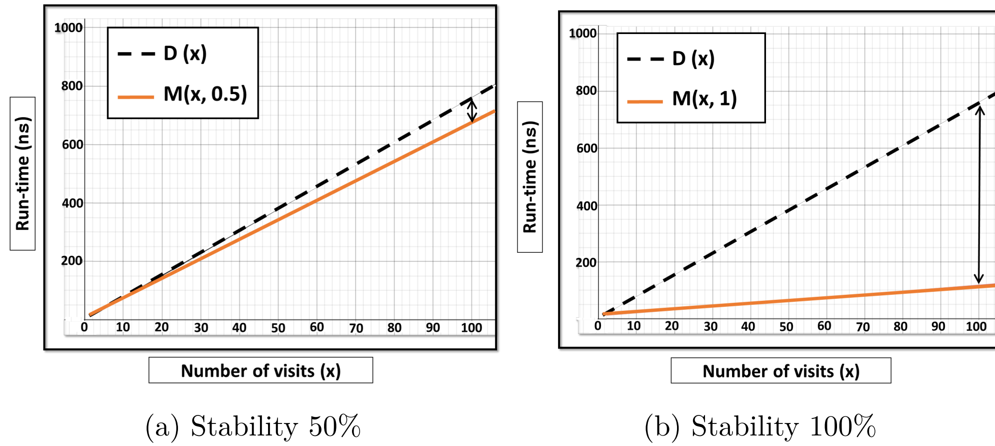
5.4.2.3 Minimum Visits

We stated earlier that for any $S^* < S \leq 1$ there exists a sufficiently large $x > 1$ for which $M(x, S) < D(x)$. Therefore for every $S > S^*$, there is a critical x value, $x^* > 1$, such that $M(x, S) < D(x)$ for all $x > x^*$. The value x^* is critical because it reveals the minimum number of cast visits required for MemCast to outperform dynamic casting. For any given $S > S^*$, x^* can be found by solving $M(x^*, S) = D(x^*)$, which equates to calculating the intercept of the $M(x, S)$ and $D(x)$. Rearranging for x^* gives:

$$x^* = \frac{S(M_{success} - M_{default})}{S(M_{success} - M_{default}) + M_{default} - T_{dyn}} \tag{5}$$

Observe that x^* is well-defined since $S(M_{success} - M_{default}) + M_{default} - T_{dyn} \neq 0$, because $S \neq S^*$. To identify x^* for our data set, we instantiate (5) with our observed values $M_{success} = 1.060$, $M_{default} = 12.265$, and $T_{dyn} = 7.590$, which gives:

$$x^* = \frac{11.205S}{11.205S - 4.675}$$



(a) Stability 50%

(b) Stability 100%

Figure 55: Run-time savings with MemCast at different levels of Stability

Figure 54 plots[3] x^* against S, along with the line $S^* = 0.417$, indicating minimum stability. For example, $x^* = 151.32$, 6.04, and 1.72, for 42%, 50%, and 100% stability, hence 152, 7, and 2 visits are needed for these stability values respectively. Observe in particular, that only two visits are required at 100% stability. Conversely, observe that x^* increases as S approaches S^* from above. Hence cast sites with low stability (close to the minimum) require more cast visits to outperform dynamic casting.

Previously we demonstrated that run-time savings accrue as S increases. Here we show that further savings accrue as x increases. Figure 55a depicts the estimated time of MemCasting for 50% stability. At 50% stability, $x^* = 6.04$. Thus the run-time saving at $x = 7$, is $D(7) - M(7, 0.5) = 53.13 - 52.24 = 0.89ns$. Notice however, when x increases to, say $x = 100$, so too does the run-time savings: $D(100) - M(100,\ 0.5) = 759 - 671.86 = 87.14ns$. Figure 55b depicts the estimated time of MemCasting for 100% stability. At $x = 100$, $D(100) - M(100, 1) = 759 - 117.21 = 641.79ns$, thus demonstrating an increase in the accrued savings not just when x is large but also when S approaches 1. The magnitude of this increase is visually depicted in Figure 55 by the double-headed arrows in both graphs.

[3]For consistency with earlier graphs, we plot x^* along the horizontal axis.

5.4.2.4 More Complex Dynamic Casting

We have assessed MemCast when it replaces the fastest form of dynamic down-casting (a single inheritance cast to the complete object). We showed that MemCast outperforms the fastest form of dynamic casting in as little as seven casts for a site with 50% stability and two casts for 100% stability. Nevertheless, one more question remains, how well does MemCast perform for more complex forms of dynamic casting? That is, cast sites that target a sub-object rather than the complete object or, which result in a null pointer; these sites incur a search through the RTTI data-structure. Although hierarchy dependent, the run-time of a complex dynamic down-cast is greater than the cost of the fastest dynamic cast we have observed. With this in mind, we will now investigate the minimum stability and executions required for MemCast when the run-time of dynamic casting is higher than our observed minimum.

Substituting $M_{default} = T_{dyn} + T_{cache}$ into (3) and (5) gives:

$$S^* = \frac{T_{cache}}{T_{dyn} + T_{cache} - M_{success}} \quad , \quad x^* = \frac{S(M_{success} - T_{dyn} - T_{cache})}{S(M_{success} - T_{dyn} - T_{cache}) + T_{cache}}$$

Then assigning $M_{success} = 1.06$ and $T_{cache} = 4.675$ results in:

$$S^* = \frac{4.675}{T_{dyn} + 3.615} \quad , \qquad x = \frac{S(T_{dyn} + 3.615)}{S(T_{dyn} + 3.615) - 4.675}$$

Table 12a evaluates these two equations, starting from the lowest observed dynamic cast time and increasing it in every subsequent table entry. The second column evaluates the minimum stability based on these hypothetical dynamic casting costs. This exercise demonstrates that as the cost of dynamic casting (T_{dyn}) increases, the minimum stability decreases. As the minimum stability decreases, it extends the range of cast sites that would benefit from MemCasting.

The last five columns in Table 12a demonstrate how the number of visits required for MemCast to outperform dynamic casting also reduces as T_{dyn} increases.

		Minimum Executions (x) for				
$\mathbf{T_{dyn}}$	S^*	$S = S^* + 0.1\%$	$S = S^* + 1\%$	$S = S^* + 5\%$	S = 50%	S = 100%
7.59	41.72%	419	43	10	7	2
10	34.34%	345	36	8	4	2
15	25.11%	253	27	7	3	2
20	19.80%	199	21	5	2	2
25	16.34%	165	18	5	2	2
30	13.91%	141	15	4	2	2
35	12.11%	123	14	4	2	2
40	10.72%	109	12	4	2	2
50	8.72%	89	10	3	2	2
75	5.95%	61	7	3	2	2
100	4.51%	47	6	2	2	2
500	0.93%	11	2	2	2	2
1,000	0.47%	6	2	2	2	2
10,000	0.05%	2	2	2	2	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(a) $M_{success} = 1.06$ (Object pointer return case)

		Minimum Executions (x) for				
$\mathbf{T_{dyn}}$	S^*	$S = S^* + 0.1\%$	$S = S^* + 1\%$	$S = S^* + 5\%$	S = 50%	S = 100%
8.55	39.82%	400	41	9	5	2
10	35.44%	356	37	9	4	2
15	25.70%	259	27	7	3	2
20	20.16%	203	22	6	2	2
25	16.58%	167	18	5	2	2
30	14.09%	142	16	4	2	2
35	12.24%	124	14	4	2	2
40	10.82%	110	12	4	2	2
50	8.79%	89	10	3	2	2
75	5.98%	61	7	3	2	2
100	4.53%	47	6	2	2	2
500	0.93%	11	2	2	2	2
1,000	0.47%	6	2	2	2	2
10,000	0.05%	2	2	2	2	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(b) $M_{success} = 1.485$ (Null pointer return case)

Table 12: Minimum Stability and Executions for the MemCast function for $M_{success} = 1.06$ (Object return case)

This is shown with five different stability values. The first three columns take the calculated minimum stability (S^*) and add a small increment (0.1%, 1%, 5%) to ensure $S > S^*$. For the purposes of later comparison we also tabulate values for 50% and 100% stability. In all cases, the minimum executions required tend toward two as the cost of dynamic casting increases. Similarly, Table 12b shows the same tendencies for the cost of a null return successful MemCast, which costs $M_{success} = 1.485ns$, slightly more expensive than a pointer return.

With these results, we can see that a cast site executed at least seven times,

with 50% or more stability, will benefit from using MemCast. The accrued run-time benefits of MemCasting increases with the number of visits, with stability, and with the run-time of complex casts. Where cast sites have stability values greater than 50%, fewer cast visits are required to benefit from MemCasting; most notably, 100% stable sites need only two visits to achieve a better run-time. With this in mind we can now apply MemCast to the Deal.II library as all but one of its dynamic casts satisfies the MemCast requirements.

5.4.3 Deal.II Revised

To investigate MemCasting in a real-world setting, we revisited the Deal.II library, where this research began.

Static Analysis of Deal.II We already know that from the 33 step-x programs, 82 cast sites (out of 545) were featured, but what about the kind of hierarchies that exist in Deal.II? Further analysis of the class hierarchies published in the Deal.II 9.2.0 documentation [28] (generated using Doxygen [33]), we found that Deal.II 9.2.0 contains approximately 2298 classes. We say 'approximately' because Doxygen is not 100% accurate with its documentation, but it does give us a rough idea of the hierarchies we are working within Deal.II. So, of the identified 2298 classes, 1362 were solely primary classes with no inheritance relationships, 895 singularly inherited from another class, and 41 inherited from multiple classes. Using the Unix tool grep [30], we found six instances of virtual inheritance but no evidence suggesting that any of the 82 featured dynamic casts used these hierarchies. The deepest hierarchy we found had a depth of 5 classes, and more than 1000 classes in the program were generated from templates.

Blanket Coverage From the class hierarchy analysis, we have a vague idea of the types of hierarchical complexity our featured 82 cast sites are working with,

mainly single inheritance and some multiple inheritance. However, we found no evidence of dynamic casting to virtually inherited bases. The exact complexity of individual casts is unknown for our test cases, but this does not matter, as we have already established each cast's stability. From our analysis of Section 5.2, we know that 77 (of the 82 cast sites) have a stability greater than 57%, which exceeds the minimum stability of the cheapest form of dynamic casting (41.7%). Moreover, the cast sites with a stability close to 57% were all visited more than seven times, making all 77 cast sites suitable for MemCast, irrespective of their dynamic casting costs. Four of the five remaining cast sites were visited only once, and the last was visited twice with 0% stability. These casts are unsuitable for MemCasting, as MemCast will make them slower. However, their overall impact is negligible, given that they are visited so infrequently. For this reason, we decided to go for blanket coverage of the Deal.II library by automatically replacing every dynamic down-cast with a MemCast.

To achieve blanket coverage of MemCast, we adapted the Python script used as part of the analysis phase in Section 5.2 to introduce supporting MemCache objects (all 545) while replacing every dynamic down-cast with a macro. The macro was automatically introduced as before; however, in this setting, it was used to invoke a MemCast.

MemCast Macro Similar to the profiling technique introduced in Section 5.2, we replaced every `dynamic_cast` call with the macro:

```
MEM_CAST(x,y,z)
```

which was defined in a separate header file as:

```
#define MEM_CAST(x,y,z) memCast_Resolver<y>::memCast(x,z)
```

where `x` is the `memCache` object, `y` is the target type, and `z` is the source object. Although the very same macro was used in Section 5.4.1, it is only here that we discuss the merit of using this approach. The purpose of the macro, particularly

during the implementation phase, was to simplify the process of fine-tuning the MemCast function for performance. The macro enabled us to encapsulate MemCast into a distinct module, separate from the testing program and the Deal.II library. This allowed us to make significant changes to the MemCast function (including its signature) without changing our testing code or the Deal.II library.

To fine-tune the run-time performance of MemCast we trialled and tested a series of small, incremental modifications to the MemCast code, which included: applying recomputation to reduce memory pressure, removing all speculative computation, reorganising the control flow so that the most frequently executed blocks were reached in the least number of instructions, investigating how MemCache layouts impact performance, and applying pragmas to force cache alignment. The methodology for reducing the run-times of MemCast was to make a single change to the function, decompile that function, and observe the number of instructions emitted. This provided a way of judging which optimisation and combinations were worth pursuing. The resulting net speedup was threefold. (Any attempt to streamline the syntax `memCast_Resolver<y>::memCast(x,z)` to say `memCast<y>(x,z)` introduced another level of templates, which ultimately degraded performance.)

Performance Testing In previous tests, recall that we used the straight-line fitting technique [89] to capture the execution time of individual casts. This method is purpose-built for small operations and, therefore, unsuitable for measuring whole program execution times. Consequently, we decided to use a different tool, specifically a Unix utility called multitime [31]. Multitime automatically executes a program n times and outputs the observed mean, min, median and maximum execution times. We used this utility, with $n = 100$, to compare the performance of each benchmark against a version that deployed MemCasting in place of dynamic casting.

Step-x			Mean			Median		
x	**# casts**	**Overall Stability**	**Benchmark (s)**	**MemCast (s)**	**Speedup**	**Benchmark (s)**	**MemCast (s)**	**Speedup**
52	519	100%	0.281	0.283	-0.71%	0.281	0.282	-0.36%
3	1032	100%	0.039	0.038	2.56%	0.037	0.038	-2.70%
20	3124	100%	0.105	0.105	0.00%	0.105	0.105	0.00%
10	3152	100%	0.092	0.094	-2.17%	0.090	0.094	-4.44%
4	4368	99.84%	0.114	0.113	0.88%	0.113	0.114	-0.88%
39	4664	92.23%	0.493	0.491	0.41%	0.493	0.490	0.61%
61	13381	100%	0.140	0.139	0.71%	0.140	0.139	0.71%
12b	14268	99.74%	0.172	0.169	1.74%	0.171	0.169	1.17%
12	26701	100%	0.159	0.153	3.77%	0.159	0.153	3.77%
38	26911	100%	0.095	0.091	4.21%	0.094	0.090	4.26%
16b	29590	99.67%	0.093	0.093	0.00%	0.093	0.092	1.08%
6	43056	100%	0.512	0.495	3.32%	0.511	0.493	3.52%
25	51032	99.99%	0.058	0.056	3.45%	0.057	0.056	1.75%
30	53806	100%	0.585	0.571	2.39%	0.582	0.567	2.58%
27	76669	100%	1.484	1.456	1.89%	1.466	1.440	1.77%
16	92848	100%	0.223	0.222	0.45%	0.223	0.222	0.45%
11	195807	100%	1.013	0.997	1.58%	1.010	0.994	1.58%
8	212026	100%	1.567	1.554	0.83%	1.545	1.540	0.32%
13	250362	100%	1.917	1.903	0.73%	1.899	1.884	0.79%
14	257290	100%	5.101	5.047	1.06%	5.089	5.030	1.16%
7	538396	100%	6.226	6.043	2.94%	6.186	6.015	2.76%
21	694337	100%	6.630	6.571	0.89%	6.613	6.553	0.91%
51	865996	100%	20.634	20.534	0.48%	20.619	20.521	0.48%
26	938781	100%	2.400	2.384	0.67%	2.328	2.299	1.25%
23	10521455	100%	13.792	13.687	0.76%	13.785	13.686	0.72%
				Average All	**1.31%**		Average All	**0.93%**
				Average > 10,000 Cast	**1.68%**		Average > 10,000 Cast	**1.63%**

Table 13: Time comparison of Deal.II with and without MemCast

Results Table 13 shows the execution time captured after running 25 of the Deal.II step-x programs[4] with and without MemCasting. The first column details each step-x program, its number, the total number of casts it contains, and its overall stability (calculated by weighting the stability of each cast by the number of times it is visited). The following two columns show the mean and median timings for each benchmark and MemCast version, and the speedup expressed as a percentage.

We found that MemCast improved performance in all but a handful of tested programs. For those that were negatively impacted, we observed that less than 5000 casts were executed during its run-time. The minimum cost of a dynamic cast was shown to be around 7.59 nanoseconds and a MemCast to be 1.06 nanoseconds. Thus with such few visits any degradation in performance is bound to be inconsequential.

What is most significant is that every program that performed more than 10,000 casts had an improved execution time (except for 16b, which exhibited no

[4] Only 25 of the original 33 programs are presented due to time restraints, as eight of the programs were simply too long to execute 200 times in a reasonable time scale.

slowdown). Looking more closely, step-38 had an execution time similar to those with less than 5000 casts; however, it had a speedup of 4.21% in its execution time when using MemCast. The very fact that every program above 10,000 casts gave an improvement (except 16b) provides substantial evidence that MemCast confers a significant improvement on dynamic cast. To further evidence MemCast's capabilities, we will test its effect on two other large programs in Sections 5.4.4 and 5.4.5.

5.4.4 OMNet++

OMNet++ [98] is a C++ simulation library primarily used for building network simulators [99]. From this library, countless open-source simulation models and model frameworks have been written, such as internet protocols, media streaming, mobile ad-hoc networks, queuing, resource modelling, and cloud computing, to name but a few [99]. Like Deal.II, OMNet++ was also featured as part of the SPEC CPU2006 bench-marking suite [123], providing a large collection of C++ source code for performance benchmark testing.

Static Analysis of OMNet++ Using a variety of tools, such as grep [30], Doxygen [33], and our own bespoke Clang tool, we found that the OMNet++ library contained 292 classes and 521 dynamic down-casts. Of these 292 classes, 66 of them are solely primary classes with no inheritance relationships, 210 singularly inherit from another class, and 16 inherit from multiple classes. The deepest hierarchy found had a depth of 11 classes and no virtual inheritance was found.

Test Programs The OMNet++ library is supplied with several sample simulation programs to facilitate learning. Collectively analysing the dynamic downcasts within each sample program, we found a broad range of cast stability outcomes (full results in Appendix C). From these findings, we chose seven of the

Simulations			Mean			Medium			SimSec/Sec		
Process	#Casts	Overall Stability	BM	MC	Up	BM	MC	Up	BM	MC	Inc
$fifo_1$	17981035	32.64%	12.524	12.355	1.35%	12.511	12.346	1.32%	29590	35396	19.62%
routing	35294	46.45%	0.065	0.066	-1.54%	0.065	0.062	4.62%	9291	9476	1.99%
dyna	1301302	50.83%	0.994	0.987	0.70%	0.988	0.983	0.51%	51831	51985	0.30%
$fifo_2$	1616289	59.30%	1.138	1.127	0.97%	1.135	1.126	0.79%	342565	346228	1.07%
aloha	24712316	74.59%	14.014	13.558	3.25%	14.02	13.603	2.97%	11996	12119	1.03%
cqn	299986	98.63%	0.177	0.174	1.69%	0.174	0.171	1.72%	98752	101157	2.44%
histograms	5000685	100.00%	3.253	3.209	1.35%	3.230	3.171	1.83%	309	313	1.40%
			Average all		**1.11%**	Average all		1.97%	Average all		**3.98%**

BM - Benchmark, MC - MemCast, UP - Speedup, Inc - Increase

Table 14: Time comparison of OMNet++ with and without MemCast.

sample programs for MemCast benchmark testing, with stabilities ranging from 32% to 100% (see Table 14). We believe this range of stability values would provide an adequate contrast to the tests performed on the Deal.II library, which predominantly featured programs of 100% stability.

Performance Testing To test the performance of MemCast when applied to the OMNet++ library, we again use the unix multitime tool [31], like we did with the Deal.II tests. This tool was set up to perform 50 runs of each of the 7 chosen programs, once with MemCasting and once without. From each set of 50 runs the multitime calculates the overall mean and median execution times, which enables us to compare a standard run to a MemCast run. In addition to these time values, OMNet++ produces its own performance measures and outputs them to the terminal after each run. We have chosen to present the results of one of these measurements, called SimSec/Sec, which measures the number of simulated seconds performed per real-time second. We include this value to support our own findings and as an interesting insight into the performance improvements made using MemCast specifically in OMNet++ simulations.

Results Table 14 presents the performance results of running a variety of OMNet++ simulation programs with and without MemCasting. Seven simulation programs were chosen; their process names, number of dynamic casts performed, and the overall stability values are listed in the first column of Table 14. The

Mean and Medium columns present the outcomes of the multitime tool for each simulation; where BM (Benchmark) reflects the captured times without the use of MemCasting and MC (MemCast) reflects the captured times with MemCasting. The 'Up' column presents the speedup incurred through the use of MemCasting as a percentage. Finally, the last column presents the (unique to OMNet++) SimSec/Sec measurement for the benchmark and MemCast simulations and the percentage increase of these simulated seconds when using MemCasting.

The results in Table 14 show that in all but one case, the performance of these OMNet++ simulations improved with MemCasting. On average, the mean speedup was 1.11% and the medium speedup was 1.97%. Where an improvement was not made, the difference in captured times was minor (0.001 seconds) and when compared with median results, suggests an anomaly in the data set that is skewing the mean calculation.

Perhaps the most interesting result is that of fifo_1, which presented with a low stability score of 32.64% (lower than the minimum stability described in Section 5.4.2.2), and yet demonstrated a performance speedup greater than 1.3%. With further analysis of this particular simulation, we found it had 32 down-casts altogether, one of which was called a total of 14,384,516 times with only 15.802% stability. However, further testing showed that using MemCast at this cast location still significantly outperformed dynamic casting. Why is this? We identified that the fifo_1 simulation program creates several new classes, all of which inherit from a library class called `cSimpleModule`. The `cSimpleModule` class happens to be one of the most-derived classes within the largest (9 classes deep) hierarchy in the library. The new classes defined in the fifo_1 extends this hierarchy to a depth of 11. The cast in question performs a cast from one of the new classes defined in fifo_1 to the `cSimpleModule` type defined in the library itself. Not only is this cast performing a traversal over a large RTTI structure, but it is also pulling RTTI

data from two different areas of memory, the programs data region and the link libraries data region. In our case, these data locations exist at almost opposite ends of memory, meaning that this cast does not have the advantage of locality like other casts seen before. We believe the lack of locality between RTTI structures and the large RTTI hierarchy itself contributes to the exceptionally slow dynamic cast performance. Hence MemCast is still favourable despite the low stability.

5.4.5 Antlr4

Antlr4 [103] is an open-source parser generator tool. The tool takes a user-defined grammar and automatically generates the source code for a parser that can interpret the language defined within that grammar. As well as generating parsers, the Antlr4 project also comes with a set of libraries that enable run-time support for its parser-related tools for multiple different languages (including C++). Antlr is a popular framework some big-name brands use, including Twitter for query parsing, Oracle within their SQL Developer IDE, and Netbeans IDE for parsing C++ code [102].

Building test programs Antlr4 comes with a C++ run-time library which we will utilise for our MemCast testing. Before starting testing, we had to create our own C++ programs that could parse specific languages. We created two programs, one that could parse C++ code and the other HTML code. To generate the parser toolkit, we had to use the Antlr generator tool and feed it a language-specific grammar. Fortunately, Antlr4 provides a wide range of predefined grammars, including one for C++ and one for HTML. In each case, the Antlr4 tool generated a parser toolkit in C++ source code for each language. Within each tool kit are several source files, two of which (‘Lexer.h’ and ‘Parser.h’) can be linked to our main program (alongside the Antlr4 run-time library) and provide all the utilities needed to parse their respective languages. We wrote both programs to have

identical main functions, so will parse the source input in identical ways; the only difference between the two is the language-specific tool kits they use.

Static Analysis of Antlr4 Our test programs are built from the Antlr4 run-time library and a specialised parser tool kit, where each tool kit extends the hierarchies found in the run-time library. As each program has its own specialised toolkit specific to the language they parse (either C++ or HTML), they each have a different number of classes and dynamic casts within their source code.

The Antlr4 run-time library alone contains 152 classes and 155 dynamic down-casts. Of the 152 classes, 68 are solely primary classes with no inheritance relationships, 83 singularly inherit from another class, and only 1 class inherits from multiple classes. The deepest hierarchy found had a depth of 5 classes, the largest hierarchy had 13 classes, and no virtual inheritance was found. The addition of the C++ toolkit (to the run-time library) added another 197 classes (totalling 349) and 579 dynamic down-casts (totalling 734). All 197 classes inherited from a class within the run-time library, extending one of the hierarchies to contain 200 classes in total. Despite this, the deepest hierarchy remained five deep. The addition of the HTML toolkit (to the run-time library) added only 17 new classes (totalling 169) and 33 dynamic down-casts (totalling 188). Again all of these classes inherited from a library class; they did not affect the deepest hierarchy, but the largest hierarchy now consists of 20 classes.

Test Program We analysed our two parsing programs with a wide range of source file inputs and found high stability (97%+) in all cases (full results in Appendix D). So we opted to base our tests on input size and see how MemCast performed when processing different amounts of data. We chose three source files for each parser to process, each increasing in size by more than double. Table 15 lists each file name, their size, the number of dynamic down-casts performed during execution, and the overall stability found in that process. Using the Unix

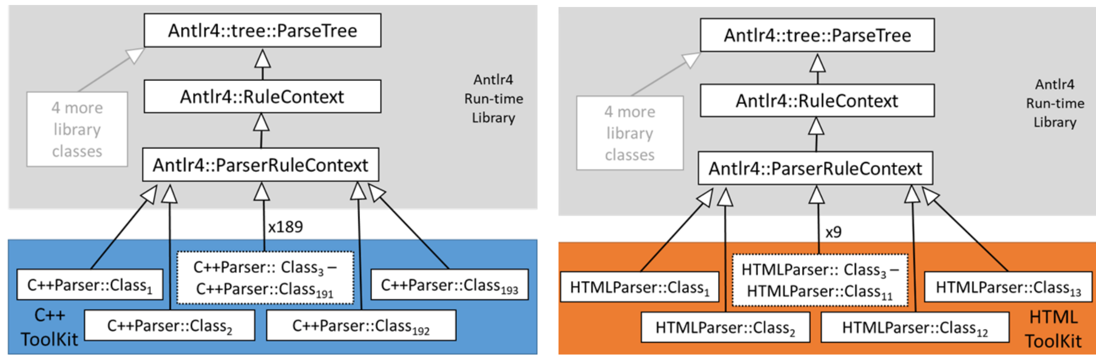
	Parser Tests				Mean (s)			Medium (s)		
	File Name	Size (kB)	# Casts	Overall Stability	Benchmark	MemCast	Speedup	Benchmark	MemCast	Speedup
C++	avrc_api.cc	48.6	8133538	97.97%	1.345	1.299	3.42%	1.343	1.295	3.57%
C++	function_lib.cc	89.7	58100011	98.79%	12.089	11.619	3.89%	12.089	11.627	3.82%
C++	data_out_base.cc	291.8	105596188	98.75%	19.859	18.980	4.43%	19.71	18.97	3.75%
HTML	antlr.html	9.5	2926395	99.75%	0.947	0.935	1.27%	0.946	0.932	1.48%
HTML	gnu.html	20.5	9881926	99.90%	3.021	2.961	1.99%	3.018	2.959	1.95%
HTML	github.html	51.8	72311578	99.93%	25.889	25.397	1.90%	25.895	25.403	1.90%
				C++ Parser:	Mean Average		**3.91%**	Medium Average		**3.71%**
				HTML Parser:	Mean Average		**1.72%**	Medium Average		**1.78%**
				Overall:	Mean Average		**2.82%**	Medium Average		**2.75%**

Table 15: Time comparison of Antlr4 with and without MemCast

multitime tool [31], we performed 100 runs of each of the three chosen input files for both the C++ and HTML parser, once with MemCasting and once without. Similarly to other tests, the multitime tool calculates the overall mean and median execution times of each set of 100 runs. The results of these runs are also presented in Table 15.

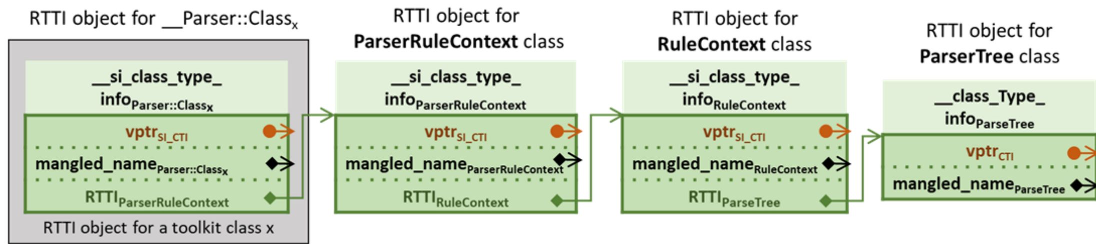
Results The results in Table 15 show an improvement in speed for all cases. The average speedup for the C++ parser was between 3.71% and 3.91%, whereas the average speedup for the HTML parser was between 1.72% and 1.78%. The overall average speedup (i.e. across both the C++ and HTML parsers) was between 2.75% and 2.82%.

Our results show that the size of the input files made little difference to MemCast's performance, as the speedup of each parser program was relatively consistent. The most interesting aspect of these results was the difference in average speedup between the two parser programs. Both programs had identical main functions but used specialised language-specific toolkits to parse their input. These toolkits added additional classes to their overall code base. The majority of these classes extend a run-time library class called `ParserRuleContext`, as seen in Figures 56a and 56b. The C++ parser adds 193 classes to the `ParseTree` hierarchy, whereas the HTML parser adds just 13. Although these hierarchies are made larger (especially in the C++ parser case), they are both single inheritance hierarchies that are only four classes deep. As a result, any RTTI structure representing any one of the most-derived classes consists of only four linked objects,



(a) C++ toolkit adds 193 classes that inherit from `ParserRuleContext`

(b) HTML toolkit adds 13 classes that inherit from `ParserRuleContext`



(c) RTTI structure for any one class from either toolkit

Figure 56: ParseTree hierarchy and RTTI structure for C++ and HTML parser programs

as seen in Figure 56c. So the RTTI structures generated from either hierarchy (depicted in Figures 56a and 56b) are no more complicated than the other, so why does the C++ program benefit more from MemCasting when compared to the HTML parser? Our theory is that the C++ Parser suffers from low memory locality.

Locality Theory Typically, the RTTI objects representing a specific hierarchy will be realised in close proximity to each other in memory. The close proximity of such objects allows the processor to take advantage of memory locality, enabling faster access to linked RTTI objects and, in turn, faster traversal through the whole structure. However, within the C++ parser program, we have 193 RTTI objects linked to one base class RTTI object (`ParserRuleContext`). With so many links to a single RTTI object, it is impossible for all RTTI links to reside nearby.

Thus, having such a bottom-heavy hierarchy within the C++ parser program ultimately decreases memory locality for that hierarchy's RTTI structures. Low memory locality for well-used RTTI structures will result in more cache misses during an RTTI traversal. This is likely true for the most-derived objects in the `ParseTree` hierarchy in our C++ parser program. Now consider our HTML parser program, which extends the `ParserRuleContext` class only 13 times (Figure 56b). This means there will only be 13 RTTI objects that link to `ParserRuleContext`'s RTTI object. With significantly fewer links, the `ParseTree` RTTI structures can benefit more from memory locality. Therefore, the performance of the HTML parser (without MemCasting) is likely superior to the C++ parser due to having better memory locality for well-used RTTI objects. This, in turn, would mean that the C++ parser had more to gain from using a dynamic cast optimiser, i.e. MemCast.

MemCache objects do not benefit from memory locality as RTTI objects do. This is because there is no data traversal in MemCast type-checking unless a default dynamic cast is called. As each of the C++ parser program runs have very high stability values (97.9%+), calls to the default dynamic cast are limited. Thus we believe that MemCast generates far fewer cache misses in the C++ parser program than the benchmark version. When this theory is extended to the HTML parser, it suggests that the HTML benchmark parser has far fewer cache misses compared to the C++ benchmark parser. Thus the HTML parser did not benefit from MemCasting to the same degree as the C++ parser did.

5.5 Related Work

Section 4.4 discussed only three works that specifically attempt to optimise the dynamic cast mechanisms; these were Gibbs and Stroustrup's fast dynamic casting [45], Ducournau's perfect hashing [35], and Padhye and Sen's [100] Fail-Fast

checking. Although we believe a successful MemCast function would outperform all three cast optimisation techniques, no open-source code could be found, making direct time comparison impossible. However, we can predict their performances against MemCast when applied to an entire program buy consideration of their designs.

- **Gibbs and Stroustrup's fast dynamic casting** would likely outperform MemCast on embedded systems for which it was designed. Their type inclusion testing technique is fast and constant time. However, their type encoding scheme is restricted to small hierarchies due to the restrictions of storing large prime multiples. MemCast, however, has no such restrictions on hierarchical size and would likely be the better choice for large programs.

- **Ducournau's Perfect Hashing** can not optimise for every type of cast, as it is restricted to virtual inheritance hierarchies. From our experiments, we found only six instances of virtual inheritance across all test programs, suggesting that virtual inheritance is rare. Given that MemCast has no restrictions on hierarchical types, it would likely outperform Ducournau's Perfect Hashing based on having higher coverage, irrespective of the speed of their type inclusion test.

- **Padhye and Sen's Fail-Fast checking** has a similar coverage issue, as they only optimise for dynamic casts with a null return. As MemCast optimises for both, it is again likely that MemCast would outperform Fail-Fast checking due to having higher coverage, even for programs with high rates of null return dynamic cast sites.

One final work that has not been discussed yet, which contained a similar memoising wrapper function (called `memoized_cast`) for the dynamic cast operator, was the Mach7 library [119]. The Mach7 library is foremost a pattern-matching

solution, providing C++ with faster code, better syntax and improved diagnostics, but also allows for the use of open class hierarchies, programs that allow class structure changes at run-time. Their `memoized_cast` function captures the results of each dynamic cast, mapping the results of each source type to a vector of valid target types. Thus, unlike MemCast, Mach7 allows dynamic casting between a source and target type to be performed once and only once, and any subsequent cast is checked and performed using their source-to-target type mapping.

Mach7's source code is openly available, allowing us to investigate the performance of its `memoized_cast` function, alongside dynamic casting and MemCasting, using the straight-line fitting technique seen in Section 5.4.1. We found Mach7's `memoized_cast` to be slightly more expensive than dynamic casting, a surprising result which suggested some form of bug or porting problem. Sadly, we could not find any published evaluation of their memoising wrapper and, therefore, could not find any evidence of its original performance and optimisation over the dynamic cast operator.

5.6 Future Work

MemCast, in its current form, is a source-based tool that is easy to deploy. Like other casting techniques (static and dynamic casting), its deployment and use are at the programmer's discretion, giving them the autonomy to decide where it is best applied. Although simple to use, it is less convenient than other casting methods within the language, as the programmer is responsible for the instantiation of MemCache objects for each cast site. Because of this, we argue that MemCasting should ideally be introduced into C++ itself. This will simplify the cast from the programmer's perspective and allow additional optimisation possibilities and safety measures.

Further Optimisations As previously discussed in Section 3.4.5, up-casting (in non-virtual inheritance hierarchies) is considered to be a safe form of casting that can be performed statically at compile-time. Many compilers apply strength-reduction to convert a dynamic up-cast to a static cast. This type of optimisation is not possible with MemCast in its current form, as MemCast has no way of deducing whether a cast is an up-cast or not. If dynamic up-casts were accidentally transformed to MemCasts, then the default conversion to static casts would not be performed, impeding performance. If MemCasting was incorporated into C++, the MemCast function could be optimised in the same way as dynamic casting.

RTTI Locality The results of testing the OMNet++ and Antlr4 libraries with MemCast suggested that the locality of RTTI structures can significantly impact the performance of dynamic casting. The evaluation of dynamic casting that we performed in Section 5.4.1 only considers small hierarchies, with the focus being on the different implementations of the virtual `__dynamic_cast` functions executed by different RTTI objects. We did not consider the impact of memory locality of linked RTTI structures. Both sets of results when testing MemCasting in the OMNet++ and Antlr4 libraries suggested that the locality of RTTI structures impacts the performance of dynamic casting. In both cases, we had unexpected results. In OMNet++ it was that MemCast appeared to outperform dynamic casting for a cast site of only 15.8% stability. In Antlr4 it was that one program had much better speedup outcomes than the other, despite having RTTI structures of identical depths. Through further analysis, the common trait between these unanticipated results was that their programs both featured low memory locality for particular RTTI objects, whether it be through a linked library or through a bottom-heavy class hierarchy. More experimental work is required to understand the impacts of the low locality of RTTI objects in dynamic casting and how MemCasting can improve performance in such cases.

MemCache Locality Further work into MemCache locality is suggested to increase the performance of MemCasting. In our experiments, MemCache objects were generated from the output of our Clang tool (Section 5.2.2), but the order and positioning of these MemCache objects were not considered. However, in future work, suppose we consider the proximity of dynamic cast sites and reflect this with the memory locality of their corresponding MemCache objects. In that case, we believe MemCasting would output even better speedup results, but further work is required to demonstrate this.

Further Safety Measures MemCast was designed to be compatible with the LLVM Clang compiler's CFI defences [131] but currently undermines its integrity. When Clang CFI is applied to the MemCast wrapper, it will check the validity of a vptr at the point of a default dynamic cast but not as part of a successful MemCast. This absence in successful MemCasts is only problematic in the case of vptr tampering by an adversary. It would entail changing both an object's vptr and the MemCache vptr to avoid the default dynamic cast. Unfortunately, the CFI defence cannot be deployed at the source level, as it is a compile-time mechanism. However, if MemCast were a language facility and the compiler knew its functionality, Clang CFI could be incorporated into a successful MemCast operation, like with dynamic casting. As part of a successful MemCast operation, vptrs would be protected from adversaries, and the CFI defence integrity would be intact.

The Clang CFI defence [131] adds just a few instructions to every vptr access to check the validity of that vptr. If added to a MemCast call, which was written to have as few instructions as possible, this will undoubtedly impact MemCast's performance. We cannot quantify the impact, but we estimate that MemCast would be 2-3 times more expensive with CFI, which is still significantly cheaper than dynamic casting. It would be interesting to see the performance overhead of

Clang CFI coupled with MemCast, as Clang CFI incurs an average overhead of 2.04% [90], whereas MemCast improves performance by 1.63-1.68%.

Multi-threaded Programs The MemCast prototype, in its current state, does not support multi-threaded programs and therefore makes no provisions to prevent type confusion under race conditions. One option to accommodate multi-threading is to use the C+11 storage specifier `thread_local`. The `thread_local` keyword, when applied to our MemCache objects, will assign them thread storage duration; meaning that each thread will have its own instance of that memCache object.

We applied thread storage duration to MemCast in our original speed tests listed in Section 5.4.3. These preliminary tests suggested that thread-safe MemCast increased the average speed of MemCasting to 2.03ns, an increase of 0.097ns (or 91.5%) compared to our original MemCast results. This increase in performance cost spears several more questions. Can its performance be improved while still ensuring thread safety? If not, is this an acceptable increase in cost per cast? How does the cost increase impact the suitability of MemCast for low-stability cast sites? Furthermore, does thread-safe MemCasting facilitate performance improvements when applied to a sizable multi-threaded software? Needless to say, the performance of thread-safe MemCasting requires further research.

5.7 Concluding Discussion

We introduced the concept of cast stability and presented evidence from a real-world library (Deal.II), that most of its dynamic down-casts have very high stability, averaging 94.89%. From this discovery, we introduced MemCast, a memoisation wrapper function that takes advantage of highly stable down-casts to improve casting speed. At the forefront of MemCast's design was its speed, which (at optimal stability) was at least seven times faster than dynamic casting. This

improvement was achieved in the context of the cheapest form of dynamic downcasting (a cast to the complete object within a single inheritance hierarchy) and achieved even better results for more complex casts. Beyond these performance experiments, we used mathematical modeling to show that even at lower levels of stability, MemCast could still outperform dynamic casting. This was achieved by finding the minimum stability required of a cast to outperform the fastest form of dynamic casting, which for our machine, turned out to be 41.7%. All but one of the cast sites we analysed within our real-world library has stability greater than this minimum, meaning all but one cast would benefit from MemCasting. Minimum stability was calculated on the assumption that the cheapest form of dynamic casting was applied, but in truth, we had no idea what types of casting were being performed; so we presented evidence that showed the more expensive dynamic casting is, the more beneficial MemCasting becomes.

With the majority of Deal.II test programs experiencing improved performance with MemCasting, we opted to test its abilities in two other C++ libraries, OMNet++ and Antlr4. In both cases, MemCasting was found to improve performance across all test cases; even in the case of a stability value of 32.64%, which is lower than the estimated minimum (41.7%). This surprising result not only brought into question the impact of low memory locality of RTTI objects when performing a dynamic cast; but also demonstrated that MemCasting has more performance benefits than we initially thought.

We discussed how MemCast could be inappropriate for some cast sites, mainly when stability is significantly low. Such sites will consistently incur the penalty of defaulting to a dynamic cast, and in turn, execution times could worsen as a result. However, in all the programs we tested, only a tiny minority of casts were found to have low stability (less than the estimated minimum), and, even in cases where it was low (like in the test case found OMNet++), MemCast can still improve performance for complex casts sites and in cases of low memory

locality. Nevertheless, MemCast is presented as a source-based tool, allowing programmer discretion over its use to avoid those rare unstable yet fast-performing cast locations (if they choose to). Even though the choice is there, we believe MemCast can be implemented as a blanket solution for all dynamic down-casts and still improve performance (as long as most cast sites benefit). If enough casts execute faster, those impeding this will quickly become insignificant, as was shown in all of our MemCast tests. All test programs were rewritten with this blanket solution to convert all dynamic down-casts (whether appropriate or not) to a MemCast. Even with the blanket solution, MemCast gained an average performance improvement between 1.11-3.91% across all test cases.

Finally, we concluded with a solid argument for introducing MemCasting as a C++ language feature. As a language feature, MemCast would be amenable to additional optimisations and security defences.

Chapter 6

Object-Oriented Code-Reuse

6.1 Introduction

Code Injection Attacks There was once a time when a hacker could exploit a simple buffer overflow vulnerability, directly injecting code instructions onto the stack and redirecting control to this code [96]. Simple attacks like this have since been thwarted with data execution protection [85], a mitigation that marks memory pages (like stack and heap memory) as non-executable, rendering code injection attacks futile. Nevertheless, attackers evolved their techniques; instead of injecting their own code, they reuse the code already present in the program, a tactic known as a code-reuse attack [77].

Code-Reuse Attacks A code-reuse attack is a run-time exploit in which control-flow is hijacked (by an adversary), and a series of code snippets (gadgets) are systematically executed (within a gadget chain) to perform some desired malicious behaviour. To perform such an attack, an adversary must find a vulnerability that allows them to inject a carefully constructed data set (known as a payload), which would result in the invocation of the desired gadget chain. Return-to-libc [92, 118] was the first attack of this kind.

Return-to-libc Return-to-libc [92, 118] worked by overwriting a stack return address (usually via a buffer overflow) to point to an attacker-chosen subroutine. Upon return, the attacker's chosen subroutine, typically a system routine within libc, would execute granting privileged access to the machine. If such a subroutine required arguments, then the attacker would carefully position each required argument on the stack as part of the original payload. This attack vector motivated the (now widely used) address space layout randomisation (ASLR) defence [105]. This randomises the address space where data and executables are stored, thus hiding the locations of subroutines. Today, ASLR is a widely-deployed defence for protecting the locations of data and subroutines, but its protection capabilities are limited as it is vulnerable to information leak attacks [39, 114].

Return-Orientated Programming Return-orientated programming (ROP) [113] is a Turing-complete exploit that can also circumvent data execution protection. A ROP gadget is a small sequence of code, usually just a few instructions, which end with a return (ret). Individually each gadget performs a small task, such as moving data to a register or executing a mathematical operation, but collectively (as a gadget chain), they perform some desired malicious behaviour. To perform a successful ROP exploit, an attacker must identify useful gadgets within the binaries and find a vulnerability that allows them to inject their payload onto the stack. An ROP payload consists of gadget addresses interleaved with data parameters, which each gadget will use. Once a gadget has been invoked, it will perform some small operation before its return instruction is used to invoke the next gadget in the chain. This continues until the attack's desired behaviour has been achieved.

ROP Variants ROP became a popular area of research, and many variants of the original attack emerged, such as Jump-Orientated Programming (JOP)

[11, 20], Pure-call orientated programming (PCOP) [110] and Just-in time code-reuse (JIT-ROP) [117].

Similarly to ROP, JOP uses small gadgets containing just a few instructions, but each gadget ends in an indirect jump rather than a return. To maintain control over a program's execution, Bletsch et al.[11] introduced a new method of chaining gadgets using a so-called dispatcher gadget. A dispatcher gadget governs the control-flow of the gadget chain; invoking the first gadget of the gadget chain, ensuring that gadget returns to the dispatcher upon completion, and repeats this action until the desired behaviour is realised. Similarly to JOP, PCOP presented a code-reuse attack based solely on gadgets ending with a call instruction and JIT-ROP presented a dynamic technique for discovering and deploying gadget chains on the fly.

As well as new variants, further advances have come with automation, with several researchers developing techniques to automate gadget search, chain building and full exploit deployment [136]. Of course, the expansion of the ROP paradigm has not gone unchallenged, and various code-reuse defences have been explored, particularly in the field of control-flow integrity (CFI).

Control-Flow Integrity The central premise behind a Control-Flow Integrity (CFI) defence [1] is to check the correctness of each control-flow path (be it an indirect jump, call, or return), against a pre-defined set of valid destination addresses. This is achieved by employing a control-flow graph (CFG). A CFG represents a superset of all possible control paths an application can traverse during execution. Any deviation from this control-flow indicates an illegal operation and will induce termination. CFI defences, in principle, are sound and will prevent any form of control-flow hijacking. However, tracking every possible control-flow, within an extensive and comprehensive CFG, comes with crippling performance overheads [15]. Thus, numerous research papers have tried to produce a CFI solution with

low overheads but high accuracy, as only those with less the 10% overheads [120] will likely be considered for real-world deployment.

Each CFI defence operates at different levels of granularity, which are often described as being either fine-grained or coarse-grained. A fine-grained CFI solution will produce a strict control-flow graph with a limited number of valid target destinations and control-flow paths. A coarse-grained CFI, on the other hand, produces CFGs with more relaxed sets of valid targets and control-flow paths; making it less accurate than fine-grained CFI, but often offers better performance overheads [15].

Advanced Code-Reuse With the advancements in CFI and other defences came further attack techniques [51, 69, 112, 138]. One such technique, and the focus of this chapter, is counterfeit object-orientated programming (COOP) [112]. COOP is a code-reuse attack that uses counterfeit objects (containing attacker-chosen vptrs) to form a payload; and uses virtual functions as gadgets.

COOP This chapter explores the COOP exploit [112] and a variant known as COOPlus [21]. COOP, at the time of publication (2015), was able to bypass a wide range of CFI defences [1, 27, 88, 133, 142], including more advanced techniques explicitly targeting C++ [43, 106, 140]. Schuster et al. [112] discussed the importance of a CFI defence to consider C++-semantics, specifically class hierarchies, as those that did not were susceptible to COOP attacks. This, of course, sparked more research into C++-semantic-aware CFI defences [14, 37, 40, 43, 58, 87, 106, 133, 135, 140, 141], many of which were capable of thwarting COOP. However, in August 2021, Chen et al. [21] developed COOPlus, a new variant of the COOP exploit that can go undetected in almost all CFI defences posed so far.

Naturally, a dynamic dispatch site will have multiple possible control-flow paths, one for each valid object type it can receive. Therefore a CFI defence will

include each path as a valid target for that dispatch site. Of course, there is only one correct path at run-time, determined by the dynamic type the dispatch receives. If CFI does not dynamically track object types, then there will always be the weakness of over-approximation. If it could infer the absolute path, then the call should be devirtualised and the function called directly. The failure of not being able to statically determine the absolute correct path, therefore, fundamentally weakens CFI. COOPlus takes advantage of this weakness and can execute, without detection, within the scope of a CFI over-approximation. This is achieved by manipulating a base class instance to address a derived class vtable, where a derived class function can be successfully invoked in a CFI defence. If that function interacts with derived class attributes, then the function will result in a memory violation, as it will access data members outside the bounds of the manipulated base class instance.

One defence COOPlus could not bypass was CFIXX [16]. CFIXX is not a CFI defence but was designed to complement CFI with object type-integrity verification. CFIXX protects the integrity of the object's type by guaranteeing the integrity of an object's vptr during dynamic dispatch. It does not, however, monitor the control-flow of a program; this falls to an accompanying CFI defence. CFIXX is considered a sound defence against COOP [3, 17, 38, 63, 104]; however, the defence breaks ABI conformance [21], which makes it unattractive for real-world deployment. We analysed the implementation of CFIXX and identified several vulnerabilities in the defence when deployed without CFI support. In light of the recent COOPlus variant and its ability to bypass CFI, we believe that these vulnerabilities pose a threat to CFIXX's integrity guarantees.

From our analysis of CFIXX and the new COOPlus variant, we argue that CFI is an unsuitable defence against COOPlus, as it does not consider type-awareness. Hence we propose a new type-integrity defence called Member Function Integrity (MFI).

MFI This chapter presents Member Function Integrity (MFI) a novel defence policy that guarantees the type-integrity of member functions. Type-integrity is guaranteed by making member functions type-aware at run-time, allowing them to verify the objects they receive are of a valid type before executing their function body. If functions only receive the correct object types, then there are no opportunities to supply counterfeit objects (or otherwise) to these functions, thus mitigating member function reuse attacks, like COOP.

Contributions This chapter presents:

- A discussion of three flaws in the current CFIXX implementation and the impact they could have on the overall integrity of this defence.
- A new defence policy, Member Function Integrity (MFI), which mitigates all known COOP and COOPLUS attacks.
- An MFI implementation proposal detailing the mechanisms and type inclusion testing methods required for deployment.
- A proof of concept that demonstrates the advantages of MFI over other defences such as CFIXX and Clang CFI.

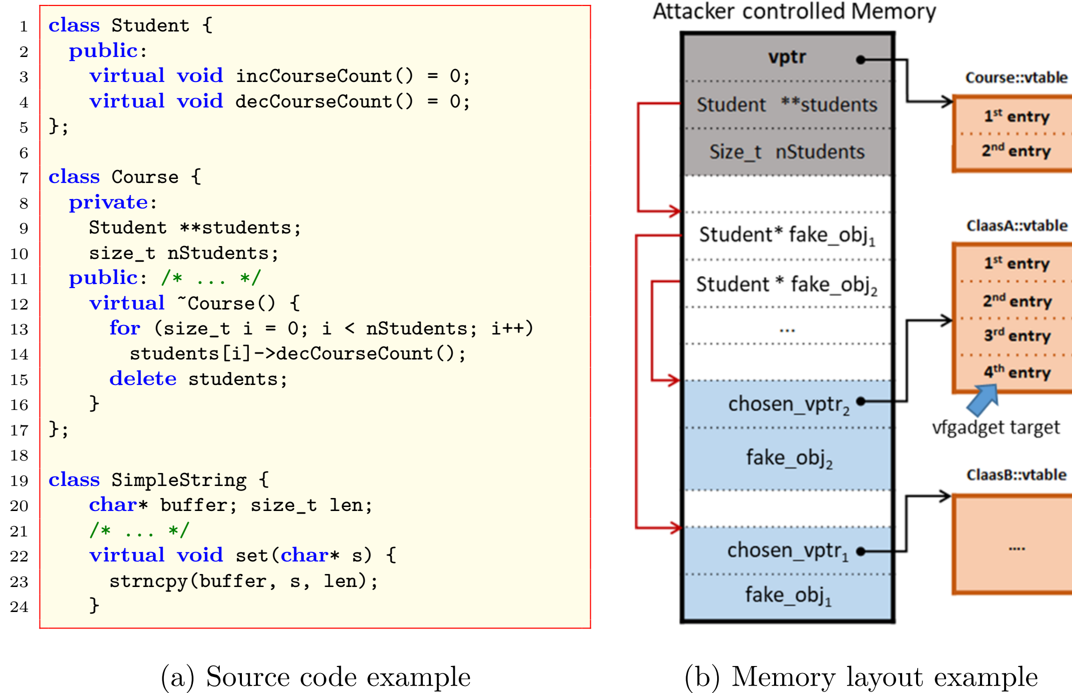
Chapter Structure This chapter illustrates the COOP exploit, demonstrating how it works, the proposed defences, and its most recent advancement (COOPLUS) in Section 6.2. Section 6.3 discusses CFIXX (one of the few defences to prevent COOPLUS) and illustrates three known flaws in the defence. Section 6.4 introduces our new defence policy, MFI, and provides an implementation proposal for its deployment. To complement the design, we present a proof of concept in Section 6.5, demonstrating how MFI can defend against attacks that CFIXX and Clang CFI cannot. We provide additional future work suggestions beyond its implementation in Section 6.6 before the chapter finalises with a concluding discussion in Section 6.7.

6.2 Counterfeit Object-Orientated Programming

Schuster et al. [112] developed a novel code-reuse attack specific to C++, called Counterfeit Object-Orientated Programming (COOP). COOP uses virtual functions as gadgets and requires a payload of counterfeit objects with attacker-chosen vptrs. These counterfeit objects are used to drive a series of attacker-chosen dynamic dispatches, which are leveraged to invoke a chain of carefully selected virtual function gadgets. When first published, COOP preyed on the lack of consideration for object-orientated semantics in coarse-grained CFI defences, which allowed it to execute without detection.

6.2.1 The COOP Exploit

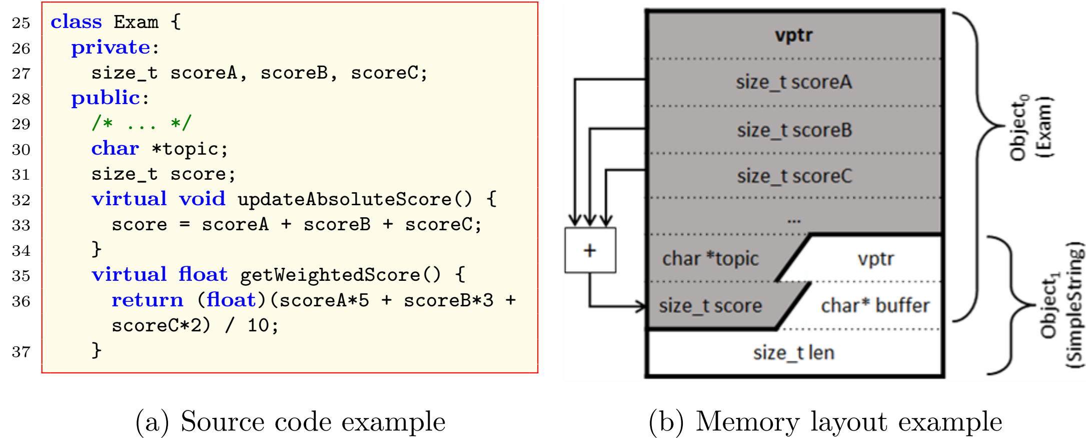
To perform a successful COOP exploit, an attacker must identify a set of useful virtual function gadgets (called vfgadgets). These vfgadgets will perform specific individual tasks but, when chained together, manifest some desired behaviour. A vfgadget chain is manufactured around a main-loop gadget. A main-loop gadget (a type of dispatcher gadget) is a virtual function that contains a loop. This loop must iterate over a container of object pointers, invoking a virtual function with each iteration. Once all components of an attack have been identified, an attacker can populate the object container (used within the main-loop gadget) with counterfeit objects. The counterfeit objects are made up of attacker-chosen vptrs and data members. The vptrs are used as part of the main-loop gadget, invoking an attacker-chosen virtual function with each iteration. These virtual functions will operate on the attacker-chosen data members within the counterfeit objects. The counterfeit objects may even overlap one another, meaning that multiple objects can share data members. Shared data members allow data to be transferred between one invoked vfgadget and the next, enabling an attacker to create useful gadget chains.



(a) Source code example (b) Memory layout example

Figure 57: COOP gadget chain example (taken from [112] © 2015 IEEE)

Main Loop and Gadget Chain Example Figure 57 provides an example of a COOP exploit (taken from the original COOP paper [112]). The source code listed in Figure 57a contains a potential main loop gadget in the virtual destructor function `Course::~Course()`. This destructor function iterates over a container of `Students`, calling a virtual function with each iteration. This virtual function (`decCourseCount()`) is the second virtual function of the `Student` vtable; hence, when dynamically dispatched, it is realised as a call using the second virtual function entry from the vptr address-point that, importantly, is one machine word away. Figure 57b depicts an attacker payload, which is used to populate the `Student` container. Notice that each counterfeit object has an attacker-chosen vptr. The counterfeit vptrs are carefully selected to address a vtable entry, which is one machine word behind the target vfgadget. When the `Course::~Course()` destructor is invoked, it will iterate over the manipulated `Students` container and dispatch a virtual function using the counterfeit objects (via their corrupt vptrs),



(a) Source code example (b) Memory layout example

Figure 58: COOP overlapping object example (taken from [112] © 2015 IEEE)

thus creating a gadget chain.

Overlapping Objects and Data Sharing There are two additional classes (`Exam` and `SimpleString`) within the code listed in Figures 57a and 58a, containing two virtual functions of interest. The first, `Exam::updateAbsoluteScore()` (line 32), performs an arithmetic operation on three `Exam` data members and stores the result in a fourth called `score`. The second, `SimpleString::set()` (line 22), uses the `SimpleString::buffer` field as a destination of a write operation. Figure 58b depicts an attacker payload, where a counterfeit `Exam` object overlaps a counterfeit `SimpleString` object. These objects overlap so that the `Exam::score` data member shares the exact memory location of the `SimpleString::buffer` data member. When the `Exam::updateAbsoluteScore` vfgadgets is executed, the attacker-chosen data stored in the data members `scoreA`, `scoreB`, and `scoreC`, will overwrite the value in the `score`/`buffer` data member. The following vfgadget in the gadget chain, `SimpleString::set()`, uses that overwritten data member as the target address of the write operation. This combination of vfgadgets enables an attacker to write to a dynamically calculated memory address, i.e. it enables arbitrary writes.

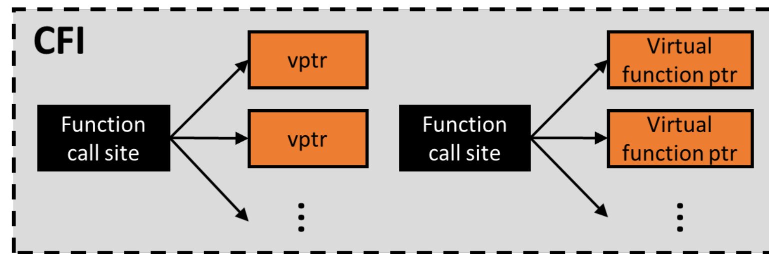
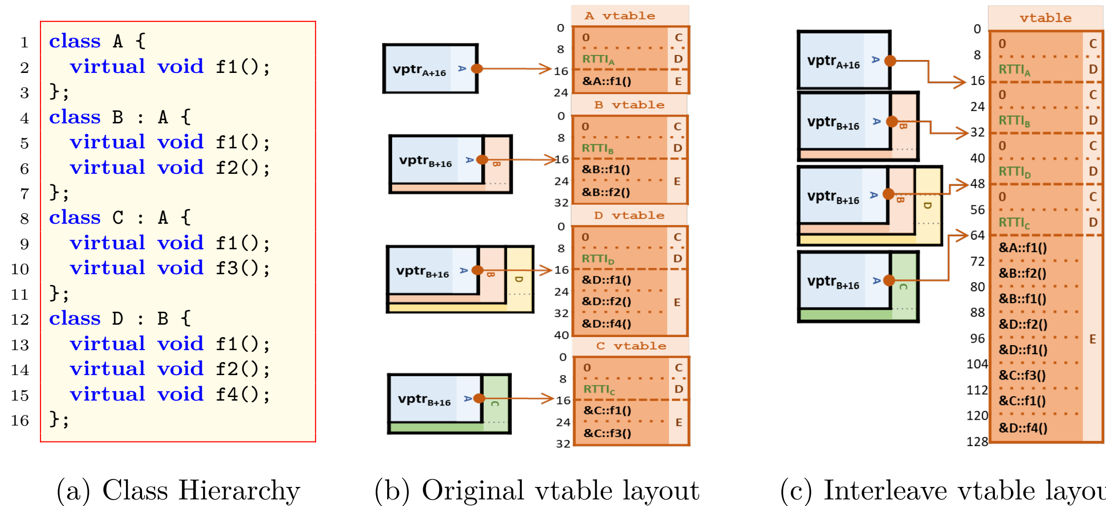


Figure 59: CFI maps function call sites to a set of valid targets

6.2.2 COOP Defenses

Since COOP's publication, there have been many proposed defences, the majority of which are C++-semantic-aware CFI solutions [14, 37, 40, 43, 58, 87, 106, 133, 135, 140, 141]. All of these defences protect dynamic dispatch call sites, by verifying their control-flow path before the target virtual function is called. Some specifically protect vtables with vptr checking [14, 37, 40, 43, 133, 140], others protect virtual function pointers [58, 87, 106, 135, 141]. In either case, these CFI defences create a mapping between a call site and a set of valid targets (Figure 59), be it vptrs or virtual function addresses. However, many of these defences have since been bypassed by COOPlus [21], a recent variant of COOP (discussed in Section 6.2.3). One defence that could prevent the COOPlus variant was CFIXX [16]. CFIXX is not a CFI defence, but a complementary defence to CFI, that enforces type-integrity at dispatch sites. CFIXX's implementation will be discussed further in Section 6.3.

Before we can explain how COOPlus actually works, and most particularly, how it can bypass CFI defences, we first introduce and examine the implementation of a CFI defence. For this we chose Clang CFI [131], as it is a real-world CFI defence, which can safeguard against COOP exploits.



(a) Class Hierarchy (b) Original vtable layout (c) Interleave vtable layout

Figure 60: Interleaved vtable layout in Clang CFI

6.2.2.1 Clang CFI

Clang CFI [131] is available in the Clang compiler and provides control-flow protections on dynamic dispatch sites through vptr checking. The Clang CFI defence uses an altered version of vtable interleaving [14]. Vtable interleaving is a method of combining all vtables from one hierarchy into one large vtable. As a single vtable, all its address-points can be arranged in such a way that verifying a vptr can be performed as a straightforward (constant-time) range and alignment check.

Figure 60 depicts two vtable layouts. One is the standard layout produced by the Clang compiler (Figure 60b), the other by Clang CFI (Figure 60c). The method of vtable interleaving, used by Clang CFI, ensures two properties of the Itanium ABI [23] are upheld:

1. The displacement to the complete object and RTTI field (labelled C and D, respectively) exist at a constant negative offset from a vptr address-point.
2. Virtual function pointers exist at the same offset entry from a vptr for all derived-class types.

Table 16 demonstrates how both properties are satisfied, listing the constant offset of all vtable entries before and after interleaving. These offsets hold true for all

Vtable Entry	Original Offset	Interleaved Offset
Displacement to complete	-16	-16
RTTI	-8	-8
f1()	0	+48
f2()	+8	+40
f3()	+8	+40
f4()	+16	+72

(a) Offset of each vtable entry

Class	Run-time Type	Start	End
A	A, B, C, D	16	64
B	B, D	32	48
C	C	64	64
D	D	48	48

(b) Valid address-point ranges for each static type

Table 16: Metadata for vtable interleaving example

vptr address-points, given that the vptr has access rights to a particular function.

Clang CFI performs vptr checks in two stages. First is a range check based on the static type of the object. Table 16b lists the valid ranges for each class type from our example in Figure 60c. The second stage is an address-point check, ensuring a valid address-point is used within the specified range. The address-point check is supported using a bit-vector, where each bit represents an aligned offset location (within a valid range), and a set bit corresponds to a legal address-point.

Clang CFI provides a fine-grained CFI defence against vtable hijacking attacks. It enforces accurate vptr targets at dynamic dispatch sites by verifying that the vptr used is one from a set of valid vptrs expected at that location. For example, Clang CFI will map the virtual function call `obj->f1()` (with static type `A* obj`) to all four address-points in Figure 60c (i.e. map to all four class vptrs). This is because, `f1()` is inherited by every derived class, so should be accessible to every derived class object. Upon invocation, Clang CFI checks that the vptr stored in `obj` points to a location between `+16` and `+64` of the interleave vtable and ensures it addresses one of the four possible address-point locations.

6.2.3 COOPlus

Chen et al. (2021) [21] proposed an advanced form of COOP (coined COOPlus) that can bypass advanced C++-semantic-aware CFI defences [58, 62, 63, 93, 94,

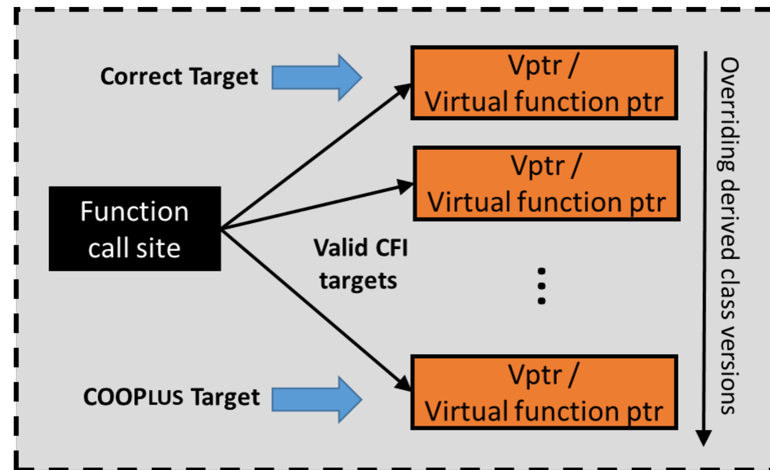
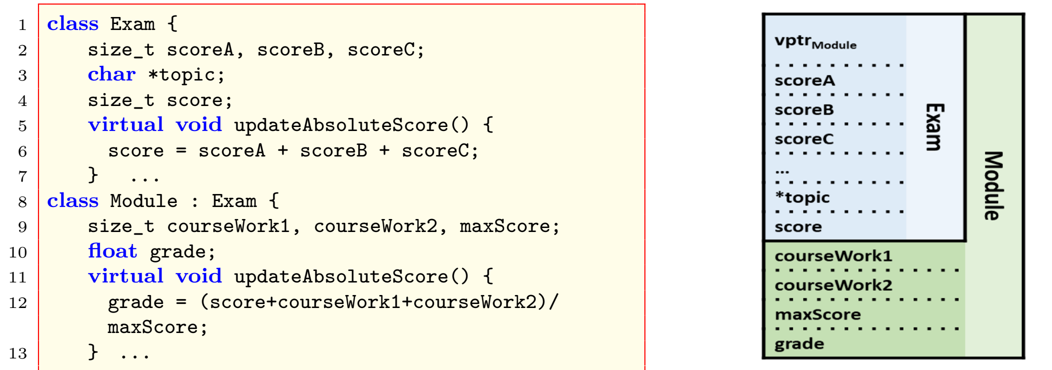


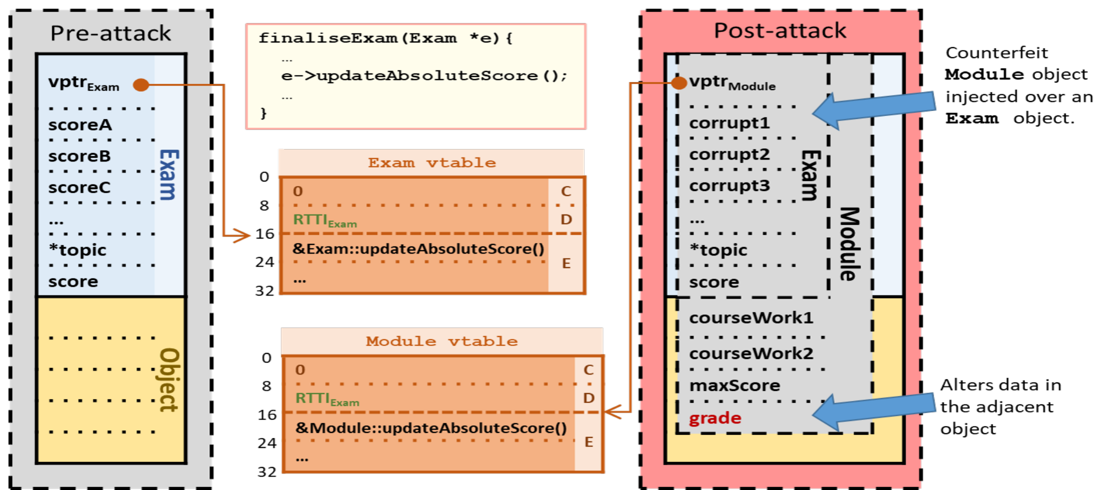
Figure 61: COOPLUS targets out-of-context but type-conforment CFI destinations

133]. C++-semantic-aware CFI defences generate a set of valid targets for a dynamic dispatch site using static analysis. Static analysis will generate an overestimation of valid target vptrs (or virtual function pointers) to accommodate all possible (valid) dynamic types the dispatch may use. COOPLUS exploits this overestimation by targeting derived class functions while using a base class instance (see Figure 61), an illegal operation that appears valid under CFI.

To demonstrate a COOPLUS attack, we introduce the `Module` class in Figure 62a, alongside its object's layout in Figure 62b. The `Module` class inherits from `Exam`, overriding its virtual function `updateAbsoluteScore()` and declaring several new data members. In Figure 62c, the function `finaliseExam()` receives an `Exam` object and uses it to dynamically dispatch the `updateAbsoluteScore()` function. Given an `Exam` object poised for this function call, an attacker may overwrite the data fields of that object, including altering its vptr to address the `Module` vtable. When this object is used in the dynamic dispatch, the `Module` version of `updateAbsoluteScore()` will be invoked, which is a valid target under many CFI defences. This particular function call will result in data outside the bounds of the original `Exam` object being modified, affecting the object adjacent to the `Exam` instance (see pre and post-attack in Figure 62c).



(a) New `Module` class source code (b) `Module` object layout



(c) COOPlus attack, using an overridden function in a derived class vtable to manipulate data members in an adjacent object

Figure 62: Example of a COOPlus attack

COOPlus has demonstrated that a C++-semantic-aware CFI defence, which limits valid function call destinations to polymorphic function implementations, is still not enough to protect against COOP attacks. However, one defence that can prevent COOPlus is CFIXX [16].

6.3 CFIXX Under the Microscope

6.3.1 Object Type-Integrity

Burow et al. [16] proposed a novel defence policy called object type-integrity (OTI). OTI is different to CFI, as CFI protects a program's control-flow, whereas OTI protects an object's type. If the integrity of all object types is upheld, then an attacker cannot change or create counterfeit objects, making attacks such as COOP, infeasible.

6.3.2 CFIXX Implementation

Burow et al. [16] developed a defence technique called CFIXX to enforce OTI. CFIXX is realised as a series of patches to the LLVM Clang 3.9.1 compiler. These patches enforce OTI by altering the constructor function and the dynamic dispatch mechanism in CFIXX-hardened binaries. The technique is built on the premise that once a whole object is constructed, its vptr does not change. Thus an object's type can be protected by ensuring the correctness of its vptrs. This protection is achieved by mapping each (eligible) object address-point to a single valid vptr assigned to that location during construction. This mapping is then stored within a metadata table (MDT) within a secure memory location. Thus the CFIXX patches alter dynamic class constructors so that when a vptr is assigned, it is also mirrored within the MDT. If an attacker could corrupt or create a fake vptr, that vptr would not exist in the MDT, as it would not have been assigned through a constructor. Dynamic dispatch is also patched under CFIXX so that the mechanism is performed using only vptrs from the secured MDT, as these vptrs are known to be safe and valid. Therefore, CFIXX prevents corrupt or counterfeit vptrs from being leveraged within the dynamic dispatch mechanism, thus preventing COOP and COOPlus exploits. However, one should note that by altering the dynamic dispatch mechanism, CFIXX-hardened binaries break ABI

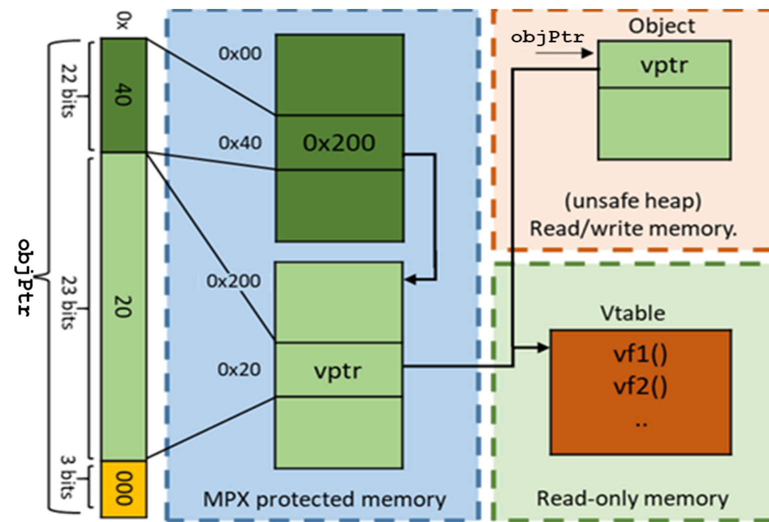


Figure 63: Two-level look-up metadata table

conformance [21].

Secure Metadata CFIXX stores all assigned vptrs within a metadata table (MDT), which resides in a memory region secured by MPX (memory protection extension) [55, 56]. Figure 63 illustrates the layout of the MDT and how entries in the table are accessed using object address-point locations. The MDT is organised as a two-level lookup table, which implements a mapping between an object's address-point and its vptr. To find an object's corresponding MDT entry the object's address (stored in `objPtr`), which is conceptually just a sequence of bits, is divided into high-order, middle-order, and low-order bits. The high-order and middle-order bits index a two-level lookup table, whereas the lowest three bits are ignored. The high-order bits index the first-level table and retrieve the address for that object's second-level table. The middle-order bits index this second-level table, pinpointing where the object's vptr MDT entry resides. The three lowest-order bits are irrelevant because vptrs are word (8-byte) aligned on 64-bit architectures. This technique of tracking an object's vptr is similar to so-called conservative pointer-finding, which is used in Boehm-Demers-Weiser conservative garbage collector [61].

Compiler Patches In a COOP attack, an attacker will inject their own, carefully crafted, counterfeit objects with fake vptrs. As such objects are injected, a constructor is never called to create them; only legitimate objects are created through constructors (except for RTTI objects, which are constructed at compile time). For this reason, CFIXX will add a new MDT entry every time an object is constructed; this is achieved through patching the sub-routine `CodeGenFunction::InitializeVTablePointer`, which is responsible for realising vptr assignment within the binaries of a class constructor. The patch does not perturb the original functionality of the routine, i.e. vptrs are still assigned within an object, but additional instructions add an MDT entry as well. Another sub-routine (`CodeGenFunction::GetVTablePtr`), which is responsible for realising the code that retrieves an object's vptr during dynamic dispatch, is also altered by CFIXX. This routine would originally access a vptr through the object itself, but CFIXX redirects this retrieval, accessing an object's vptr only through the MDT. This redirection ensures that the vptr used within any dynamic dispatch is valid and the same vptr assigned to that object during construction.

6.3.3 CFIXX Vulnerabilities

CFIXX is a complementary defence to CFI and is widely considered sound against COOP attacks [3, 17, 38, 63, 104], COOPLUS attacks [21], and even defines ground truth for the VPS (VTable Pointer Separation) defence [104]. However, CFIXX is not favoured for real-world deployment because it breaks ABI conformance [21]. Furthermore, CFIXX relies on MPX [55, 56], a now-discontinued data protection mechanism [70, 95], to protect its metadata table. Due to the novelty of CFIXX, we investigated its security capabilities particularly when deployed in isolation (without an accompanying CFI), but we assumed a secured MDT. Under these conditions, we identified three vulnerabilities within CFIXX that could result in a COOP exploit.

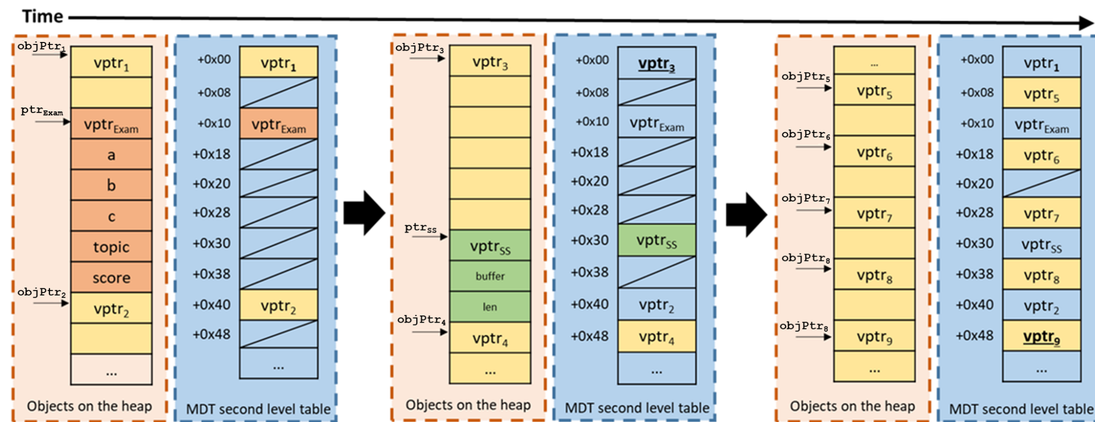


Figure 64: Overpopulated MDT over time as Heap memory is recycled. Objects include an `Exam`, `SimpleString`, and other trivial class instances

We note that two of the three vulnerabilities we discuss were briefly addressed in its original CFIXX paper [16]. However, we extend this discussion by providing examples of how CFIXX can be bypassed with these vulnerabilities. In some of these cases, an accompanying CFI defence would safeguard these vulnerabilities; however, in light of COOPlus, we believe even a CFI and CFIXX-hardened program could be bypassed if these vulnerabilities were not resolved.

6.3.3.1 Overpopulated MDT

In CFIXX, class constructors are responsible for adding new vptr entries to the MDT. This creates a mapping between a live object and its (safe and valid) vptr. However, in the current implementation of CFIXX, vptrs are not zeroed once their objects have been deallocated. Without careful deallocation management, ghost vptrs (dead vptrs left behind by deallocated objects) will continue to reside in the MDT until another constructor function overwrites them. If a particular memory segment has a high recycle rate of dynamic objects, more and more entries will be added to its corresponding MDT segment. This would result in an MDT populated with both dead and live vptrs, creating a unique use-after-free vulnerability, specific to vptr pointers.

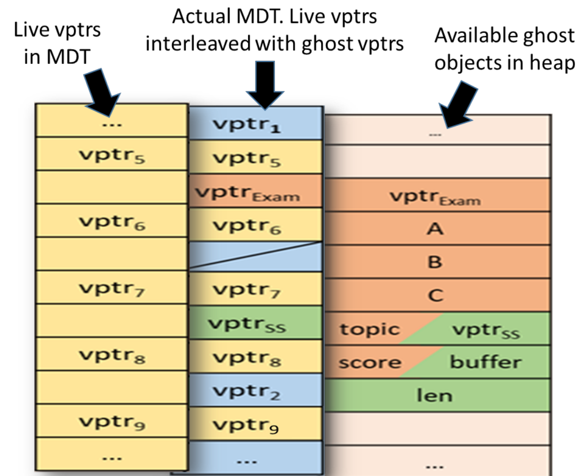
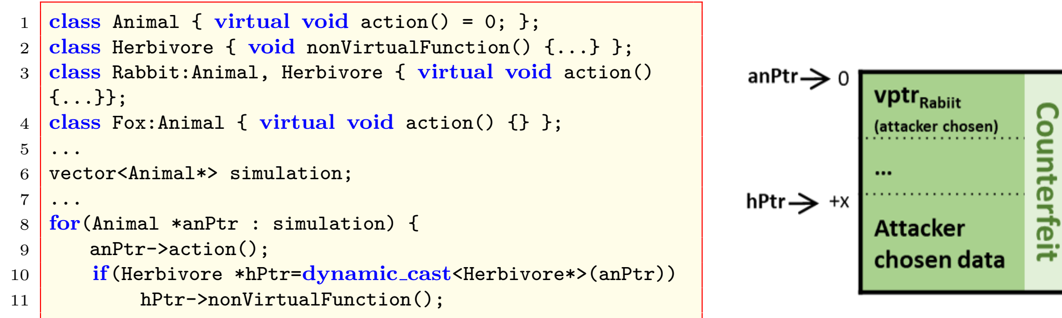


Figure 65: Possible ghost objects an attacker could exploit given the ghost vptrs available in the MDT

Figure 64 depicts a segment of heap memory and its corresponding MDT entries. Over time, this heap segment is recycled, new objects replace the old, and within the MDT, live vptrs become interleaved with ghost vptrs. Among these ghost vptrs are $\texttt{vptr}_{\texttt{Exam}}$ and $\texttt{vptr}_{\texttt{SS}}$, which address the `Exam` and `SimpleString` vtables, respectively. The position of these two MDT entries creates the illusion that an `Exam` and `SimpleString` object still reside in heap memory (as depicted in Figure 65). We call these objects ghosts, as they once lived in memory but have since been deallocated. Under CFIXX's current implementation, a dynamic dispatch using a ghost object is valid and will leverage the dispatch using a ghost vptr from the MDT. In this particular example, an attacker could launch the same exploit, with the same payload, as the example detailed in Section 6.2.1.

Ghost vptrs and ghost objects could be used to leverage a full-scale COOP attack within a CFIXX-hardened program (although harder to achieve). Figure 65 demonstrates that, given enough vptr entries in the MDT, a useful payload could be constructed that aligned with both live and ghost vptrs and enabled the execution of a useful gadget chain. This demonstrates the importance of carefully managing the integrity of the MDT; without which the defence is impaired.



(a) Dynamic cast in animal simulator program (b) Payload layout

Figure 66: Forced type confusion under CFIXX and calls to non-virtual member functions

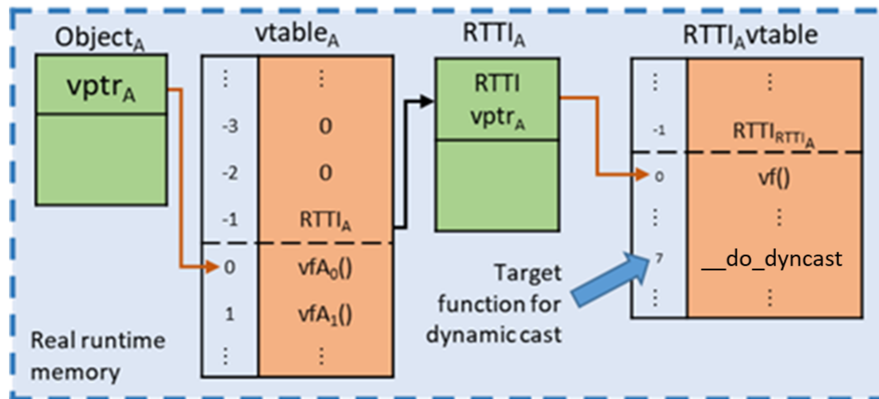
6.3.3.2 Dynamic Cast

As previously discussed in Section 3.4.4, the `dynamic_cast` operator generates a call to the global `__dynamic_cast` function. The `__dynamic_cast` function takes an object, retrieves its RTTI via its vptr, and uses the RTTI object to invoke the virtual `__do_dyncast` function. However, we found that the process of retrieving RTTI within the `__dynamic_cast` function is unprotected by CFIXX. Thus, within dynamic casting, there exists a vptr access that is not redirected through the secure MDT. As a result, a dynamic cast can be performed on a counterfeit object without CFIXX detection.

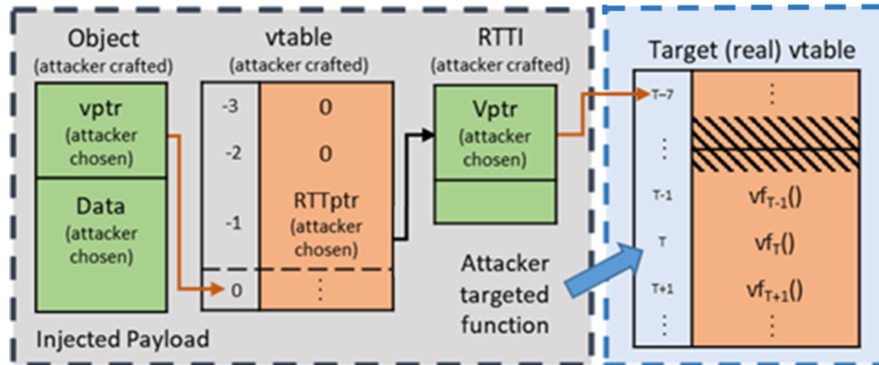
Non-Virtual Calls We revisit our simulator example of foxes and rabbits in listing Figure 66a. This example performs a dynamic cast to a `Herbivore*` type (line 10) before calling a non-virtual function from the `Herbivore` class (line 11). Suppose an attacker constructed a counterfeit object using a `Rabbit` or `Herbivore` vptr (see Figure 66b) and passed this object to the dynamic cast. Under CFIXX protection, this dynamic cast would be successful. Once the cast is complete, the non-virtual function is invoked using the counterfeit object.

Missed Dispatch CFIXX protects only the binaries compiled using a CFIXX-enhanced compiler; this makes sense, as it is not just about incorporating the MDT, but every constructor function and dynamic dispatch site must be rewritten to incorporate the defence. Therefore, to ensure complete coverage, all libraries linked to the program must also be compiled with CFIXX. This is problematic because CFIXX is an ABI-breaking defence [21]. Any dynamically linked libraries recompiled with CFIXX may be incompatible with other programs; thus, developers must compile these libraries statically or maintain two versions, one that is CFIXX hardened and the other that is ABI-conformant. The most prolific library this applies to is the Standard C++ Library. The Standard C++ Library is dynamically linked to almost all C++ programs and is done so automatically by the compiler without a developer needing to specify this with compilation flags. For this reason, we believe this library could easily be overlooked when applying this defence. In fact, if any dynamically linked library is overlooked during compilation, it poses a significant risk to the integrity of CFIXX, which we will now demonstrate with the Standard C++ Library.

The Standard C++ Library is responsible for RTTI, and part of RTTI is dynamic casting (see Section 3.4.4). For example, in the GNU Standard C++ Library [44], the dynamic cast mechanism uses RTTI objects and the `__do_dyncast` function to determine an object's run-time type. The `__do_dyncast` function is a virtual function and is therefore called using the dynamic dispatch mechanism. If the standard library is overlooked during CFIXX compilation, this dispatch will go without CFIXX protection. Figure 67a depicts how data members are accessed as part of a dynamic cast. The global `__dynamic_cast` function retrieves the RTTI object and uses it to dynamically dispatch the `__do_dyncast` function. When `__do_dyncast` is unprotected, it is possible to dispatch an attacker-chosen function within a `dynamic_cast` call. Figure 67b depicts such a case with an elaborate attacker payload. This payload consists of a counterfeit object, a counterfeit



(a) Normal `dynamic_cast` that dispatches the `__do_dyncast` function.



(b) Payload used to exploit `dynamic_cast` and target a specific vfgadget

Figure 67: Normal and exploited version of the `dynamic_cast` mechanism.

vtable, and a counterfeit RTTI object. Here the counterfeit RTTI object has an attacker-chosen vptr that addresses a real vtable with a virtual function of interest. If this particular counterfeit object were passed to a `dynamic_cast` site, the attacker's chosen function would be invoked.

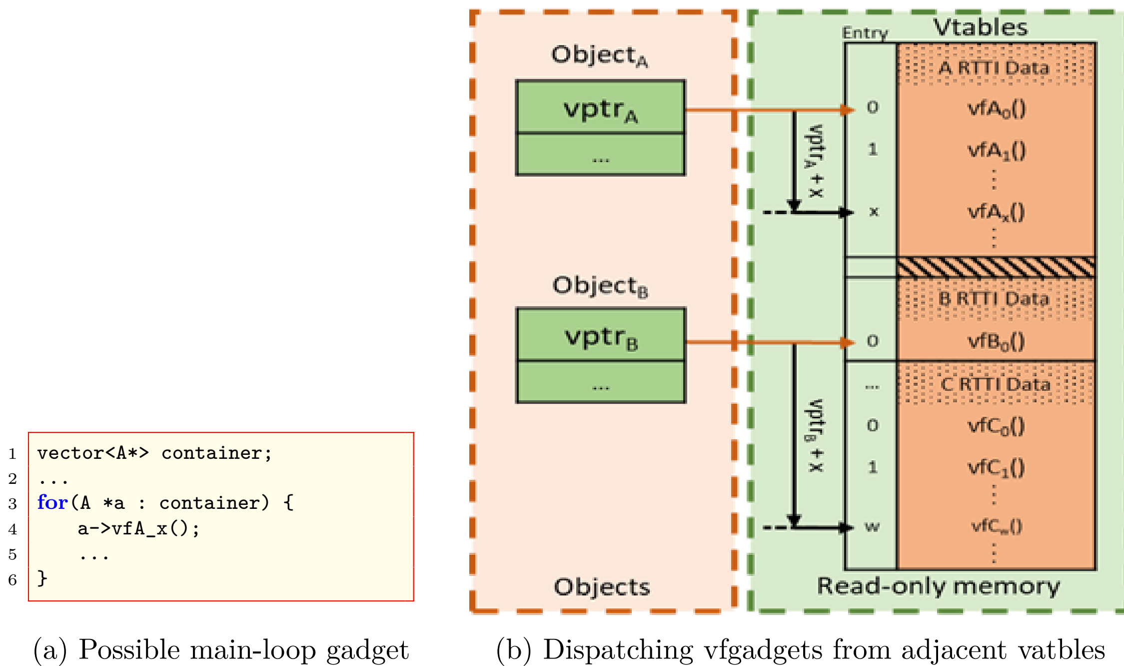
Rightly so, one would argue that the above scenario is unlikely and would be exceptionally difficult to manufacture an attack through this particular dispatch site. After all, the most basic of CFI vtable protections would prevent vtable injection. However, we argue that it demonstrates a more significant issue within CFIXX: it takes only one exploitable and unprotected dynamic dispatch site to break the integrity of the defence. OTI requires a fine-grained approach, which in this case means that all dispatch sites need hardening, and all live dynamic objects

need tracking. **If one dispatch site is exploitable and unprotected** (through a missed library or otherwise), **all virtual functions become available to an attacker**, a dyer consequence for a small human error. In cases where CFIXX and CFI are applied together, security would fall to the accompanying CFI defence, which would limit the number of vfgadgets available to an attacker but would not avoid COOPLUS. Note that if a library was overlooked when applying a CFIXX defence, it may have also been overlooked when applying the accompanying CFI defence. We discuss such a scenario in Section 6.5.4.

6.3.3.3 Adjacent Vtables

The dynamic dispatch mechanism is critical in a COOP attack because it can indiscriminately call a virtual function, given (whether valid or not) an object address, a vptr, and a vtable offset. The CFIXX defence ensures the validity of two aspects of this non-discriminatory execution, the object address and its vptr, by forcing dynamic dispatch through the MDT. However, CFIXX contains no mechanisms for checking valid vtable offsets, and by the author's own admission [16], there is no validation for checking that the correct object is used in a dispatch (this falls to the accompanying CFI). In an isolated CFIXX defence, it is possible to dispatch a function from an adjacent vtable; but only if the call site has a sizeable offset adjustment and receives the wrong object type. This vulnerability occurs because CFIXX does not apply type-awareness to dispatch sites.

Figure 68 presents an example of invoking a virtual function from an adjacent vtable. Figure 68a lists a `for` loop, which iterates over a container of `A` objects, dispatching the virtual function $\texttt{vfA}_{\texttt{x}}\texttt{()}$ with each iteration. This dispatch site is realised as a call using the virtual function found at an `x` offset from the supplied vptr (i.e. $\texttt{vptr}_{\texttt{A}}\texttt{+x}$). If `x` is significantly large, then an attacker could supply the dispatch site with an unrelated object. Suppose the attacker used a valid (live) `B` object, which had few virtual function entries in its vtable. When the dispatch is




```
1 vector<A*> container;
2 ...
3 for(A *a : container) {
4     a->vfA_x();
5     ...
6 }
```

(a) Possible main-loop gadget (b) Dispatching vfgadgets from adjacent vatbles

Figure 68: Calling virtual functions in adjacent vtables

executed, the displacement `+x` will be applied to the `B` vptr (i.e. `vptr`$_B$`+x`). For a large enough `x`, this adjusted vptr will not only reside outside the bounds of the `B` vtable but could address a virtual function entry within the adjacent vtable (in this case, `C`, see Figure 68b). As `B` is live, its vptr (`vptr`$_B$) has a valid entry in the CFIXX's MDT, so an adjacent vtable call like the one described would bypass CFIXX's defences.

6.3.3.4 Accompanying CFI

In its current implementation, we have shown that CFIXX has several vulnerabilities that could allow an attacker to bypass its defences. In principle, ghost vptrs are an issue of MDT management, not an issue of the defence policy itself; it is, therefore, a vulnerability that can be patched and rectified in future versions. However, the other vulnerabilities discussed expose a weakness in CFIXX's defence methodology: CFIXX not only complements CFI but is dependent on it. We conclude, therefore, that an isolated CFIXX defence can only offer partial

protection, and when accompanied by CFI, its security level is only as good as the CFI's ability to protect these loopholes.

The most significant vulnerability is an unprotected dispatch site (through linking or otherwise), which could open up all virtual functions to exploitation. An accompanying CFI can safeguard this particular vulnerability, but what guarantee is there that a CFI defence will identify the dispatch sites CFIXX could not? The answer depends on which CFI defence it uses; thus, this raises further research questions about which CFI defence would best accompany CFIXX.

6.4 Member Function Integrity

The majority of COOP defences discussed thus far explore the protection of virtual tables or virtual pointers alongside dynamic dispatch call sites, placing all checking mechanisms prior to a function dispatch. Only one other defence has broken this mould, Bauer and Rossow [9], who explored the idea of dropping vtables altogether and hence eliminating the threat of vtable hijacking[1]. We also wish to break this mould with the exploration of post-dispatch member function protection, i.e. performing type-checking as part of the prologue of a member function body after a dispatch. We have called this defence Member Function Integrity (MFI).

Figure 69 depicts the differences in each defence. CFI generates a one-to-many mapping between a specific virtual dispatch site and its valid targets, either vptrs or virtual functions. This mapping generates an over-approximation of targets, which can be bypassed with COOPLUS. CFIXX generates a one-to-one mapping between objects and vptrs, and although it upholds the integrity of an object's vptr, it does not protect dispatch sites. Without this protection, unrelated functions can be invoked from adjacent vtables; thus, CFIXX must lean on CFI defences to block such loopholes. MFI will also generate a one-to-one mapping;

[1] Their technique replaces vtables with large switch statements. From our examination of their paper, we believe their technique would not prevent COOPLUS attacks.

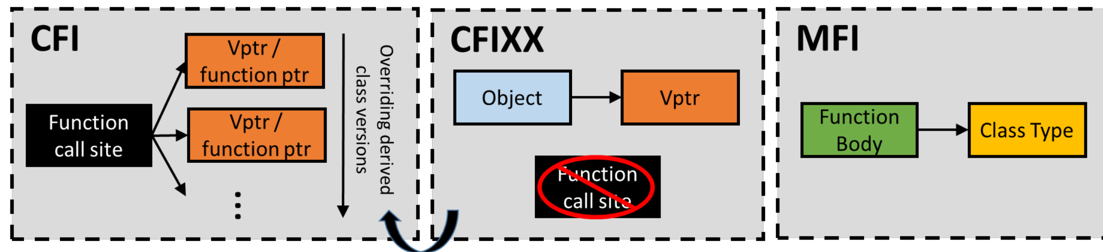


Figure 69: Mappings produces in each defences technique

however, this mapping will be between a member function and its class type. With this mapping, member functions can become type-aware, enabling them to verify that the objects they receive are of a valid type. Like CFIXX, MFI does not protect function call sites, but because type verification occurs post-dispatch (within the functions themselves), adjacent vtable calls are impossible. Thus, call sites do not need the same level of protection, and MFI can independently protect against COOP and COOPlus attacks.

6.4.1 Defence Policy

We present a new defence policy, member function integrity (MFI), which guarantees that all member functions are invoked using an object of its own type, thereby preventing code-reuse attacks containing member functions. Member functions are defined within a class and therefore have an associated type. At the source level, these functions are invoked using an object of its associated class type (or a sub-type see Section 2.3.5). At the binary level, this is realised by passing the address of the invoking object as a function's first parameter (Section 3.2). The address-point used in a member function call must address a (sub-)object matching the function's type (Section 2.3.5). In other words, every member function should receive an address-point to a (sub-)object of the same type as the function itself. However, member functions have no means of verifying they receive the correct data type at run-time, which allows function reuse attacks like COOP.

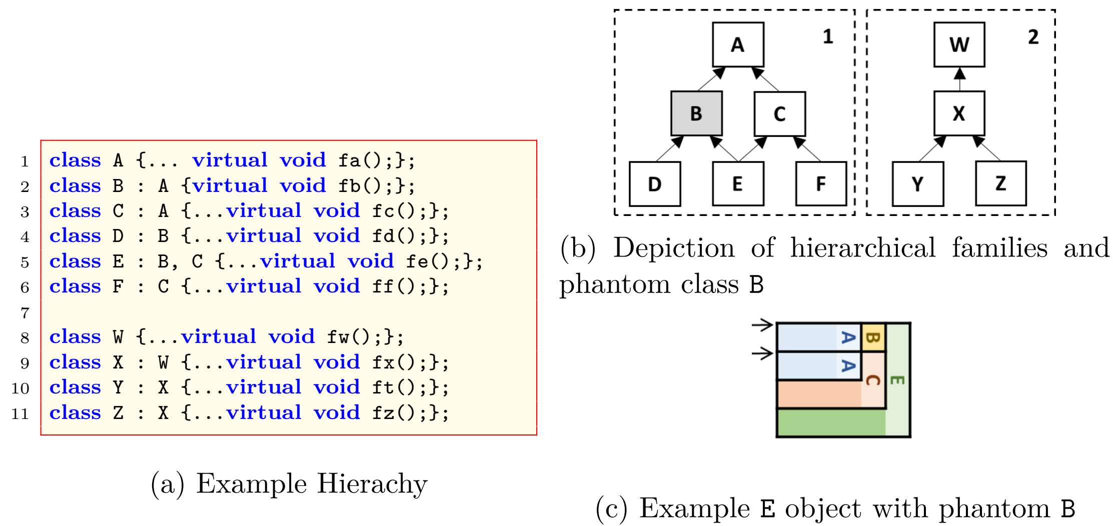
However, if member functions were made type-aware, object type compatibility could be verified, and member function reuse exploits mitigated.

6.4.2 Implementation Proposal

We propose introducing a dynamic object type tracking technique and a type inclusion test strategy (first discussed in Section 4.3). In our type inclusion strategy, classes will be assigned a unique class code and their instances an encoded key that encapsulates hierarchical relationships. Functions defined within a protected class will naturally have access to the class code, which will be stored as a const static variable. As a const static variable, its value will be realised as a literal within the instruction code (Section 2.2.5). Therefore when a function performs a type inclusion test, the class code will be embedded within the function prologue, making the function type-aware while preventing manipulation by an adversary. The embedded class code will be compared with the encoded key linked to the run-time object. If the object's key is compatible with the function's class code, the function must be a member of the object's type and execution is permitted to continue; otherwise, an exception should be thrown. To achieve this, we propose repurposing the type inclusion testing scheme used in Bitype [101] (a run-time type confusion detector) and applying it to member functions.

6.4.3 Converting Bitype's Encoding Scheme

Pang et al. (2018) [101] developed a novel compiler-based tool that identifies and reports type confusion vulnerabilities. Their prototype tool (Bitype) is built on top of the LLVM Clang compiler, using LLVM's compiler toolchain technologies to instrument a type inclusion testing technique that protects cast sites. The two main components of Bitype are its object tracing technique, which uses the same MDT set-up as CFIXX, and a novel, safe encoding scheme called **safe sets**.



(a) Example Hierachy

(b) Depiction of hierarchical families and phantom class `B`

(c) Example `E` object with phantom `B`

Figure 70: Bitype example hierarchy

6.4.3.1 The Encoding Scheme

Bitype's novel safe encoding scheme consists of a safe set code and a fast type inclusion checking technique. A safe set code is assigned to every class and is used to represent the inherited relationships of each class within its own hierarchy. Each safe set code consists of a series of binary flags, where each flag represents a single relationship. These encoded relationships are used to check the validity of a cast at run-time. The checking technique is a series of fast bit-wise operations that compare two safe set codes. This comparison verifies whether or not an object's source type and cast target type have a valid hierarchical relationship and if the cast is safe to proceed. If a cast is deemed safe, the program will continue; otherwise, a bug report is filed, and the type confusion location is identified.

Bitype's Safe Sets Encoding We will demonstrate Bitype's encoding scheme using the example listed in Figure 70. Figure 70a introduces two distinct class hierarchies, referred to as hierarchical **families**. Once Bitype has identified all families (as depicted in Figure 70b), each class within a given family is assigned a **class index**. Class indexes are just integer values assigned sequentially (starting

Class	Class Index	Class Code	Safe Set Code
A	1	00001	00001
B	1	00001	00001
C	2	00010	00011
D	3	00100	00101
E	4	01000	01011
F	5	10000	10011
W	1	00001	00001
X	2	00010	00011
Y	3	00100	00111
Z	4	01000	01011

(a) Bitype encoding scheme

Class	Family Index	Class Index	Class Code	Safe Set Code
A	1	1	000001	000001
B	1	2	000010	000011
C	1	3	000100	000101
D	1	4	001000	001011
E	1	5	010000	010011
F	1	6	100000	100101
W	2	1	000001	000001
X	2	2	000010	000011
Y	2	3	000100	000111
Z	2	4	001000	001011

(b) MFI encoding scheme

Table 17: Adapted Bitype encoding scheme for MFI

with 1) in a top-down left-to-right manner through the hierarchy. In this example, class `B` is considered a **phantom class**. A phantom class is a derived class that does not introduce any new data member attributes, just member functions. Without the introduction of new attributes, a phantom class' object layout is identical to its direct-base class layout (as seen with sub-objects `A` and `B` in Figure 70c). As there is no difference in layout, casting between the two is always considered safe, so both classes receive the same class index. Table 17a lists the class index assignments made for our hierarchical examples.

Class indexes are transformed into bit-vectors, called **class codes**, where the nth bit (starting with the least significant bit) is set to `1` for a class indexed as n. Using these class codes, a class's **safe set code** can be generated. A safe set code represents all hierarchical relationships for an individual class within a single family, where each bit represents a particular class relationship. Formally, a class's safe set code is the logical (inclusive) disjunction of its class code and the class codes of all its base classes. For example, if we denote a class's safe set code as SSC_{class}, then SSC_F is the inclusive disjunction of class codes `F`, `C`, and `A`:

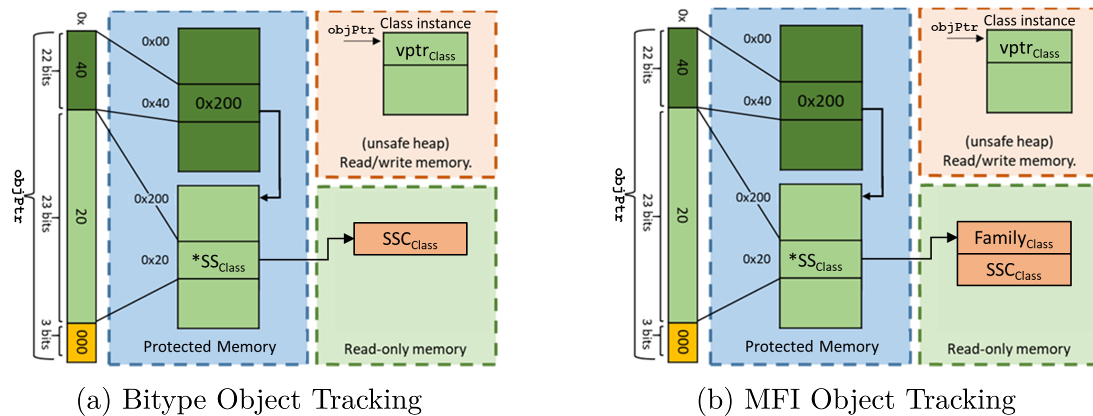
$$SSC_F = 10000 \lor 00010 \lor 00001 = 10011$$

All class safe set codes are listed in Table 17a.

MFI Encoding Scheme To apply Bitype's safe set encoding scheme to an MFI defence, some alterations are required:

1. **Phantom classes will not share class codes.** A cast between a phantom class and its direct-base class is deemed safe under Bitype, allowing both classes to share the same index and safe set codes. However, in MFI, phantom classes may introduce new member functions, which should only be accessible to phantom class instances and their derived-types. These functions should never be accessible from a base class instance, and therefore the distinction between base and a derived phantom classes must be explicit.
2. **Family Indexes are assigned to every class.** Bitype does not encode family data into its safe set codes, as it can rely on static type-checking to prevent any form of cross-family casting. MFI, on the other hand, has no such luxury. A virtual function hijack attack could receive any object from any hierarchy, so an MFI encoding scheme must distinguish between hierarchical families. For this reason, class codes must be accompanied by a (static const) family index value. This family index value also accompanies MFI safe set codes.
3. **Safe set codes represent a set of types positioned at an object's address-point** In MFI, safe set codes will not represent all hierarchical class relationships but only those that share the address-point of a class's complete object instance. A derived class's (MFI) safe set code is evaluated as the logical (inclusive) disjunction of its class code and all base class codes that share the complete object address-point produced by that derived class. For example, consider the complete `E` object (Figure 70c) that shares its complete address-point with two sub-objects from classes `A` and `B`. `E`'s safe set code is therefore the inclusive disjunction of `E`, `B` and `A`'s class codes, i.e. $SSC_E = 10000 \vee 00010 \vee 00001 = 10011$, as seen in Table 17b.

Table 17b lists the results of MFI's adapted encoding scheme for the hierarchies



(a) Bitype Object Tracking (b) MFI Object Tracking

Figure 71: Adapted Bitype object tracking for MFI

listed in Figure 70.

6.4.3.2 Object Tracking

Bitype uses the same object tracking technique as CFIXX [16], a two-level lookup metadata table (MDT) that links an object's address-point to its safe set data (Figure 71a). MFI could apply the same object tracking technique (assuming accurate MDT management is available), mapping live object address-points to safe set data and its family index value (Figure 71b).

Bitype Table Management Although Bitype uses CFIXX's MDT strategy, it is not implemented in the same way. CFIXX is a defence mechanism that is realised as a series of patches to the LLVM Clang compiler, where the compiler itself has been adapted to produce CFIXX-hardened binaries. Bitype is built on top of the Clang compiler and therefore does not interfere with the compiler's source code. Instead, Bitype protection is instrumented using a series of LLVM passes.

Bitype adds an object's safe set code to the MDT using an instrumented `trace_obj()` function, which is inserted immediately after every (detectable) new object construction. However, this method cannot detect all object initialisations,

and object coverage is approximately 83%, suggesting limitations to this method. For this reason, we believe a compiler altering method, like CFIXX, provides the best coverage.

MFI Table Management Like CFIXX, we propose that our MFI defence be implemented within the compiler. The compiler will then have the ability to apply all necessary instrumentation to implement the MFI defence. These instrumentations will ensure all member functions are realised with MFI-hardened code: modifying virtual member functions to include type inclusion testing and modifying constructors/destructors to include MDT management.

In Section 3.3, we discussed the process of object construction, and how (non-primary class) objects are constructed through a series of nested constructor calls. Each constructor generates an instance of its own class, first by invoking its base class constructors and then initialising its own data members. Where a class has a vtable, its constructor will assign a vptr. This often means the vptr is repeatedly overwritten by a derived class constructor, with each unwinding of the nested constructor call. The same should hold for MDT entries; each constructor should assign its own MDT entry, which the callee constructor should overwrite if they share the same address-point.

In MFI, every constructor will assign its own safe set data to the MDT, using the address-point passed to the constructor function to determined the MDT entry. If a derived class object shares that address-point with a base instance, then the MDT entry will be overridden during construction. Once an object is wholly constructed, each address-point will have an assigned safe set and family index, which will correspond with the most-derived (sub-)object type at that address-point location. For example, Figure 72 depicts a fully constructed `E` object from the class hierarchy in Figure 70. The most derived-type located at address-point $\texttt{p}_0$ is `E`, therefore $\texttt{p}_0$ will map to `E`'s safe set data in the MDT. On the other hand,

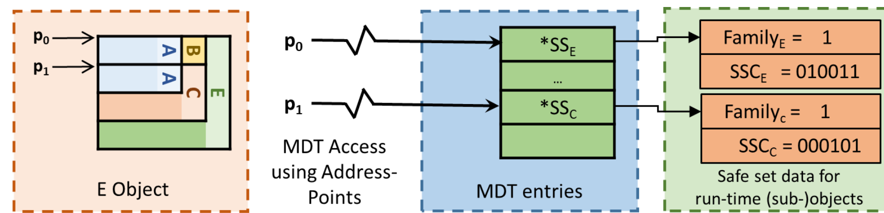


Figure 72: Object tracking data for MFI binaries

the most derived-type located at address-point p_1 is C, so p_1 will map to C's safe set data.

Destructors should be modified to perform the same actions as constructors, but instead of initialising MDT entries, it should nullify them.

6.4.3.3 Type Inclusion Test Technique

Member functions have types. A member function's type corresponds with the class it was defined within. When a member function is invoked, the compiler will perform an implicit cast on the invoking object to ensure that the object passed to the function is of the same type. Thus once a function is called, the expectation is that it will receive an object of the same type as the function itself. At the binary level, this is realised by a function receiving an object address that points to a (sub-)object instance of the function's type. This is the fundamental premise behind the MFI defence. MFI checks that the object address-point, passed to the function, addresses an object type that matches the type of that function.

Type Testing Once control-flow enters the function's body, the object's address-point, passed to the function, is used to access that object's safe set and family index from the MDT. The function itself is type-aware, as it has direct access to both the const static family index and class code defined within its class. With this information, a type inclusion test can be performed within the prologue of a protected member function.

The type inclusion test takes the following form: given a function call (`f`) using

an object address-point `p`, the object's family index ($\texttt{F}_\texttt{p}$) and safe set code ($\texttt{SSC}_\texttt{p}$) can be retrieved from the MDT. The function itself has direct access to its own family index ($\texttt{F}_\texttt{f}$) and its class code ($\texttt{CC}_\texttt{f}$). A type inclusion test is performed in two parts: First, we check that the object and the function belong to the same family using:

$$F_p == F_f$$

If true, a second test is performed that verifies whether the object address-point, passed to the function, addresses a (sub-)object instance matching the function's type. This test performs an exclusive disjunction ($\oplus$) and comparison, taking the the following form:

$$SSC_p \oplus CC_f < SSC_p$$

If this test returns true, then the object used in the function call is valid. However, if either test fails, an incompatible object type was passed to the function, and execution should end.

Table 18 provides an example of MFI type inclusion testing, using an object $\texttt{e}_{p_i}$`= new E()`, where the variable `e` stores either the address-point $\texttt{p}_0$ ($\texttt{e}_{p_0}$) or $\texttt{p}_1$ ($\texttt{e}_{p_1}$) to the `E` object depicted in Figure 72. Each row of this table breaks down the result of an MFI type inclusion test, using a function from a particular class. The functions featured are from classes `A` to `F` (the same hierarchical family as `E`), and a single function ($\texttt{X::fx(e}_\texttt{pi}\texttt{)}$) from class `X`, which is part of a different family. We can see that, when using the complete object address-point, $\texttt{p}_0$, only functions defined in classes `A`, `B`, and `E` are permitted to execute. This reflects the (sub-)objects available at that address location. Likewise, only functions defined in classes `A` and `C` can execute using the $\texttt{p}_1$ address-point, which also reflects the sub-objects available at that location. Function calls outside of `E`'s hierarchical family (for example, attempting $\texttt{X::fx(e}_\texttt{p0}\texttt{)}$) do not go beyond the family testing phase of this type inclusion test and as a result, will not be permitted to execute. Collectively, this table shows that when using an `E` object, only functions defined

	Function call	F_f	CC_f	$F_f == F_e$	$SSC_{e_{pi}} \oplus CC_f$	$SSC_{e_{pi}} \oplus CC_f < SSC_{e_{pi}}$
p_0	`A::fa(ep0)`	1	000001	T	010010	T
	`B::fb(ep0)`	1	000010	T	010001	T
	`C::fc(ep0)`	1	000100	T	010111	F
	`D::fd(ep0)`	1	001000	T	011011	F
	`E::fe(ep0)`	1	010000	T	000011	T
	`F::ff(ep0)`	1	100000	T	100011	F
	`X::fx(ep0)`	2	000010	F	-	-
p_1	`A::fa(ep1)`	1	000001	T	000100	T
	`B::fb(ep1)`	1	000010	T	000111	F
	`C::fc(ep1)`	1	000100	T	000001	T
	`D::fd(ep1)`	1	001000	T	001101	F
	`E::fe(ep1)`	1	010000	T	010101	F
	`F::ff(ep1)`	1	100000	T	100101	F
	`X::fx(ep1)`	2	000010	F	-	-

Table 18: Example MFI type inclusion testing for $F_e = 1$, $SSC_{e_{p0}} = 010011$ and $SSC_{e_{p1}} = 000101$

in classes `A`, `B`, `C`, and `E` can execute, which reflect `E`'s inheritance relationships.

This logical testing method ($SSC_p \oplus CC_f < SSC_p$) is simply checking whether a single bit, within an object's safe set code (SSC_p), is set. More specifically, a function (f) from a class indexed n will check that the nth bit of an object's safe set code is set. The nth bit of SSC_p signifies that a specific class instance (the function's class instance) resides at the address-point p. When this bit is set, then the result of $SSC_p \oplus CC_f$ is guaranteed to be less than SSC_p. For example, when calling `B::fb()` using the $\mathtt{e_{p0}}$ address-point of an `E` object, then:

$$SSC_{e_{p0}} = 010011$$

$$CC_{B::fb} = 000010$$

$$SSC_{e_{p0}} \oplus CC_{C::fb} = 010001 < 010011 = SSC_{e_{p0}}$$

Alternatively, if the bit is not set, then the result is guaranteed to be greater. For

example calling `C::fc()` using the same address-point:

$$SSC_{e_{p0}} = 010\,0\,11$$

$$CC_{C::fc} = 000\,1\,00$$

$$SSC_{e_{p0}} \oplus CC_{C::fc} = 010\,1\,11 > 010011 = SSC_{e_{p0}}$$

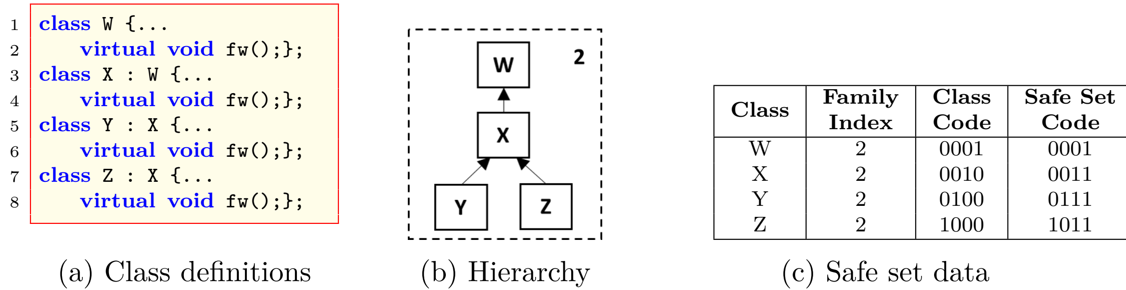
The MFI testing method is very similar to Bitype's method [101], except that Bitype does not perform a family index comparison.

6.4.3.4 Non-Virtual Function Protection

It is possible to extend the MFI defence to non-virtual functions and non-dynamic objects, if desired. However, if this defence were applied to non-dynamic objects, compound inline objects would need to be considered. A compound object is an object that resides inside another object, not through inheritance but as a data member. Therefore, a compound inline object is a compound object that aligns itself with an address-point of the encapsulating object, i.e. shares an address-point with the object it is a member of.

Compound inline objects are problematic for MFI's encoding scheme, as they create shared address-points between objects of different families. Sharing an address-point between unrelated classes means that safe set data cannot be accurately stored for both instances. A simple solution to this is to add padding to every class with a compound inline object. By adding padding, the compound object will shift out of line with the address-point location, providing it with its own unique address-point. This padding could be as simple as adding a dummy class attribute or rearranging class attributes so that a compound object is not the first data member of a class instance.

Furthermore, non-dynamic objects do not contain a vptr, so are not guaranteed to be at least 8-byte aligned like dynamic objects are. The current implementation



(a) Class definitions (b) Hierarchy (c) Safe set data

Figure 73: MFI example hierarchy and safe set data

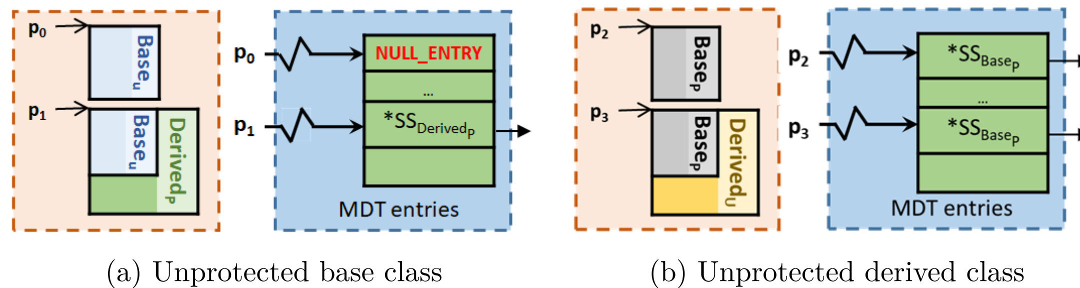
Function call	F_f	CC_f	$F_f == F_p$	$SSC_p \oplus CC_f$	$SSC_p \oplus CC_f < SSC_p$
`W::fw(p)`	2	0001	T	0000	T
`X::fw(p)`	2	0010	T	0011	F
`Y::fw(p)`	2	0100	T	0101	F
`Z::fw(p)`	2	1000	T	1001	F

Table 19: Example MFI type inclusion testing for a `W` object with a single address-point `p` mapping to a family index $F_p = 2$ and a safe set $SSC_p = 0001$.

of the MDT relies on objects being at least 8-byte aligned. Thus to protect non-dynamic objects, either they must be forced into 8-byte alignment, or the MDT needs to expand to tracking objects of all alignments.

6.4.4 Benefits of MFI

A COOPlus Defence To show that MFI would successfully defend against a COOPlus attack, we look at the example hierarchy in Figure 73. Suppose that a program constructs a `W` object with address-point p. In an MFI hardened binary, the `W` object would generate an MDT entry, storing its family index $F_p = 2$ and its safe set $SSC_p = 0001$. Now suppose an attacker attempted a COOPlus attack on this program by modifying the vptr of the `W` object, forcing it to address a derived-type's vtable in the hope to invoke the overridden `fw()` function. Table 19 lists the results of a type inclusion test for each version of the `fw()` function, showing that `W::fw()` is the only function version that would execute with a `W` object. This example demonstrates that MFI can prevent COOPlus attacks.



(a) Unprotected base class (b) Unprotected derived class

Figure 74: MFI protect with partial coverage

Protects Against Untracked Objects Any function that is MFI hardened will check the validity of the objects it receives, guaranteeing that it will only ever execute using valid address-points. If a protected function is passed an untracked object, the type-integrity check will fail, as untracked objects will not have a valid MDT entry. MDT entries not mapped to live objects should contain a special null entry pointing to a zero family index and a zero safe set.

Partial Protection The integrity of MFI will also hold in partially protected hierarchies. Critically, this means that **functions protected by MFI remain protected irrespective of the other classes in the hierarchy**. To clarify, suppose we have an MFI-protected derived class ($\texttt{Derived}_{\texttt{P}}$) that inherits from an unprotected base class ($\texttt{Base}_{\texttt{U}}$). The unprotected base class will contain unprotected functions, and its instances will not be tracked through the MDT. On the other hand, the protected derived class will contain protected functions and its instances will be tracked. Consider an instance of both the base and derived class (as seen in Figure 74a). The $\texttt{Base}_{\texttt{U}}$ object can interact with its own functions as normal (because no run-time type-checking occurs), but if it were used to invoke any derived class functions (which are MFI-protected), this would fail, as the address-point of $\texttt{Base}_{\texttt{U}}$ ($\texttt{p}_0$) maps to a null MDT entry. Now consider the protected derived class instance $\texttt{Derived}_{\texttt{P}}$. Similarly, this object can invoke base class functions, as no type-checking occurs, but it can also interact with its own protected functions, as it would pass MFI type-safety checks.

The same partial protection applies to MFI-protected base classes with unprotected derived classes. Consider the objects in Figure 74b. The base class `Base`$_P$ has MFI protections, which means its instances are tracked with MFI, and functions are hardened with MFI protections. The derived class `Derived`$_U$ is unprotected, so its class instances are not tracked through MFI. However, as the `Derived`$_U$ class inherits from `Base`$_P$ (which is MFI-protected), the `Base`$_P$ sub-object within `Derived`$_U$ is tracked through MFI. The tracking still occurs due to the nested constructor calls generated when initialising a `Derived`$_U$ object. As `Base`$_P$ is MFI protected, its constructor, whether used to generate a complete or sub-object instance, will still initialise an MDT entry. Thus when a `Derived`$_U$ object invokes an MFI-hardened `Base`$_P$ function, it will still pass MFI checks thanks to its sub-object (`Base`$_P$) being tracked.

Unprotected Classes from Linked Libraries As we have just discussed, partial protection of hierarchies is possible under MFI. The same holds true for unprotected link libraries. That is to say, **functions protected by MFI remain protected irrespective of other linked code bases**. To clarify, suppose we have an MFI-hardened program linked to an unprotected library. Further, suppose that the attacker had found a vulnerable dispatch site and attempted to use that site as part of a COOP exploit. Their payload of counterfeit objects will be injected into memory, bypassing the use of constructor functions that typically instantiate an object. Without using a constructor function, their objects will not be given an entry into the MDT. This means that MFI-protected functions cannot be used as part of their exploit, as without an MDT entry, type-safety checks will fail. Thus, the only vulnerable and viable vfgadgets in this exploit attempt will be the unprotected functions introduced by the library itself, severely limiting the pool of functions available to the attacker to construct a viable gadget chain.

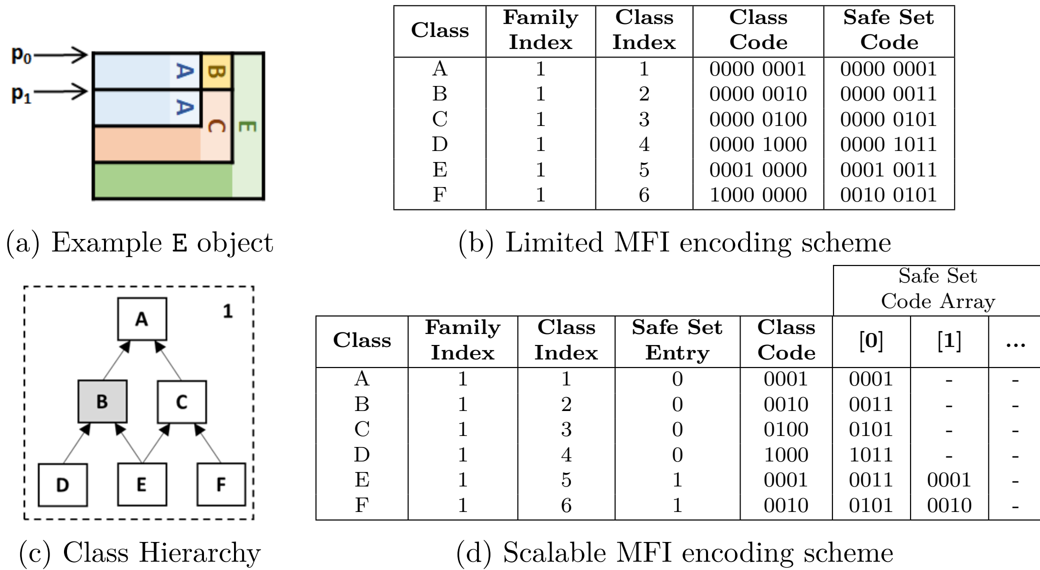
ABI Compatible Although the MFI defence should be deployed similarly to CFIXX, it should not impede ABI conformance. CFIXX breaks ABI conformance [21] by altering the vptr access of a dynamic dispatch mechanism defined in the ABI itself [23]. MFI, on the other hand, extends a function's prologue to incorporate type testing or MDT modification. Because it is incorporated into a function's body, just like a programmer's code is, there is no reason for it to break ABI conformance.

6.4.5 Scalability

One obvious drawback to the method described so far is that the size of the safe set code will limit the number of hierarchical relationships represented. This, however, can be fixed by storing safe set codes as byte arrays (rather than large data types) and altering type testing to a one-byte bit-wise operation.

We discussed in Section 6.4.3.3 that safe set tests check for a single bit flag, meaning all other bits in the test are irrelevant. If all other bits are irrelevant, then they do not need to be present in testing. Therefore, why not reduce these bit-wise operations to a single byte (the byte containing the bit we are interested in) rather than the whole safe set code (which could be 32, 64, or even 128-bits long)? The Bitype paper [101] provides an algorithm for single-byte type-checking, which they use to simplify type checks; we, however, want to use the same technique for scaling up our hierarchical encoding scheme.

Scalable MFI Encoding To allow for single-byte type-checking and, in turn, represent much larger hierarchical relationships, we must alter the MFI encoding scheme seen so far. To do this, we return to our previous example (Figure 70), which we reprinted in Figure 75 for convenience. Table 75d presents the new scalable encoding scheme, which can be compared to the previous one in Table 75b. In our scalable scheme, we switch the single data safe set code into a byte array. Safe



(a) Example E object

Class	Family Index	Class Index	Class Code	Safe Set Code
A	1	1	0000 0001	0000 0001
B	1	2	0000 0010	0000 0011
C	1	3	0000 0100	0000 0101
D	1	4	0000 1000	0000 1011
E	1	5	0001 0000	0001 0011
F	1	6	1000 0000	0010 0101

(b) Limited MFI encoding scheme

(c) Class Hierarchy

					Safe Set Code Array		
Class	Family Index	Class Index	Safe Set Entry	Class Code	[0]	[1]	...
A	1	1	0	0001	0001	-	-
B	1	2	0	0010	0011	-	-
C	1	3	0	0100	0101	-	-
D	1	4	0	1000	1011	-	-
E	1	5	1	0001	0011	0001	-
F	1	6	1	0010	0101	0010	-

(d) Scalable MFI encoding scheme

Figure 75: Scalable encoding scheme example

set codes are calculated in the same way as previously described (Section 6.4.3.1) but are then separated into single bytes and assigned to an array. We also add a new data entry to the encoding scheme as a whole, called Safe Set Entry, which lists the index of the safe set array where the class code flag exists. Due to the way hierarchies are categorised and safe set codes are generated, the safe set entry will also define the maximum length of the array. Run-time objects will still be mapped to the MDT, but the MDT entry will point to a whole Safe Set structure. This structure must contain the Family Index, the Safe Set Entry, and the Safe Set array.

Scalable MFI Type Inclusion Testing To accommodate the scalable safe code array, the type inclusion test must also be altered to take the following form: given a function call (`f`) using an object address-point `p`, the object's family index ($\mathtt{F_p}$), safe set entry ($\mathtt{SSE_p}$), and safe set code ($\mathtt{SSC_p[SSE_p]}$) can be retrieved from the MDT. The function itself has direct access to its own family index ($\mathtt{F_f}$), safe set entry ($\mathtt{SSE_f}$), and its class code ($\mathtt{CC_f}$). A type inclusion test is performed in **three** parts:

1. Check that the object and the function belong to the same family
$$F_p == F_f$$
2. Check that there exists an entry for comparison in the object's safe set code array:
$$SSE_f <= SSE_p$$
This check prevents out-of-bounds access from the safe set array, as the object's safe set entry is also the array length.
3. Check for a relationship between object and function using the relevant byte from the object's safe set code array:
$$SSC_p[SSE_f] \oplus CC_f < SSC_p[SSE_f]$$

If any test fails, then the object used in the function call is invalid, and execution should end.

Scalable Example To demonstrate that the addition of safe set code arrays can still function as part of the MFI encoding scheme, we repeat the example from Table 18 in Table 20. Again we use the object $\texttt{e}_{p_i}$`= new E()` (depicted in Figure 75a), where the variable `e` stores either the address-point $\texttt{p}_0$ ($\texttt{e}_{p_0}$) or $\texttt{p}_1$ ($\texttt{e}_{p_1}$) to an `E` object. Each table column breaks down the result of the scalable MFI type-inclusion testing scheme. We will discuss each phase of the type inclusion test and the functions it deems incompatible with the object address-point used.

1. **Family index check ($F_p == F_f$):** The comparison of family index values returns false for the `X::fx` function in both cases, correctly identifying this function as being from an unrelated hierarchy.
2. **Check for safe set code entry ($SSE_f <= SSE_p$):** The function's safe set entry ($\texttt{SSE}_\texttt{f}$) is compared against the safe set entry for the relevant object address-point ($\texttt{SSE}_{\texttt{e}_{\texttt{pi}}}$). As the object's safe set entry specifies the length of its safe set code array, this check ensures no out-of-bounds reads in the following test. It also correctly eliminates functions `E::fe` and `F::ff` from

	Function call	F_f	CC_f	F_f == F_e	SSE_f	SSE_f <= $SSE_{e_{pi}}$	$SSC_{e_{pi}}[SSE_f]$	$SSC_{e_{pi}}[SSE_f]$ $\oplus$ CC_f	$SSC_{e_{pi}}[SSE_f]$ $\oplus CC_f$ $< SSC_{e_{pi}}[SSE_f]$
p0	A::fa(e_{p0})	1	0001	T	0	T	0011	0010	T
	B::fb(e_{p0})	1	0010	T	0	T	0011	0001	T
	C::fc(e_{p0})	1	0100	T	0	T	0011	0111	F
	D::fd(e_{p0})	1	1000	T	0	T	0011	1011	F
	E::fe(e_{p0})	1	0001	T	1	T	0001	0000	T
	F::ff(e_{p0})	1	0010	T	1	T	0001	0011	F
	X::fx(e_{p0})	2	0010	F	-	-	-	-	-
p1	A::fa(e_{p1})	1	0001	T	0	T	0101	0100	T
	B::fb(e_{p1})	1	0010	T	0	T	0101	0111	F
	C::fc(e_{p1})	1	0100	T	0	T	0101	0001	T
	D::fd(e_{p1})	1	1000	T	0	T	0101	1101	F
	E::fe(e_{p1})	1	0001	T	1	F	-	-	-
	F::ff(e_{p1})	1	0010	T	1	F	-	-	-
	X::fx(e_{p1})	2	0010	F	-	-	-	-	-

Table 20: Example MFI type inclusion testing for $F_e = 1$, $SSE_{e_{p0}} = 1$, $SSC_{e_{p0}} = \{0011, 0001\}$, $SSE_{e_{p1}} = 0$, and $SSC_{e_{p1}} = \{0101\}$

executing on the $\mathtt{e_{p1}}$ address-point, as the class code flags for these functions exist outside the bound of the $\mathtt{e_{p1}}$ safe set code array.

3. **Relationship check** $SSC_p[SSE_f] \oplus CC_f < SSC_p[SSE_f]$**:** The `XOR` comparison is performed using the function's class code ($\mathtt{CC_f}$) and the relevant entry from the object pointer's safe set array ($\mathtt{SSC_{e_{pi}}[SSE_f]}$). This last test correctly eliminates all remaining unrelated functions, returning the same results as the previous example in Table 18.

This example demonstrates that a scalable MFI defence is possible and capable of removing the restrictions on relationship encoding seen in our first scheme. In our scalable scheme, hierarchies are now restricted by the size of the family index and the safe set entry. These values are represented with integers, so even using short ints (2 bytes in size) would allow for 65536 families and, within those families, 65536 safe set code entries. As a safe set code entry has four bits, it can store four different relationships; thus, each individual family can have as many as 65536*4 = 262144 classes, and if all family indexes were used, the encoding scheme could store up to 262144*65536 = 17,179,869,184 classes. This is more than enough for any large and complex program, but even if it was not (for family indexes, safe set entries, or both), the short data type could be increased to 4, 8 or even 16 bytes.

6.5 MFI Proof of Concept

To back up the theory of MFI protection and demonstrate its benefits compared to other defences, we opted to provide a proof of concept. To prove this concept, we built a contrived stock management program with a single buffer overflow vulnerability that would enable us to perform three different dynamic dispatch exploitation techniques. These techniques are as follows:

1. A vptr overwrite with derived vptr type (COOPlus)
2. A dynamic dispatch which invokes a function from an adjacent vtable
3. The use of an unprotected dynamic dispatch from a linked library

We will present the stock management program and each exploitation technique. After explaining each exploit, we will discuss their effectiveness when attacking three different versions of the stock management program, each with a different defence deployed (CFIXX, Clang CFI, or MFI). The third exploit example relies on an unprotected dynamic dispatch. To provide such a dispatch, the program dynamically links to a separate library (containing dynamic dispatch code) that is compiled separately without any defensive protections.

6.5.1 Example Program Design and Vulnerability

Our example program is a stock management system that could exist in a retail store setting. It is a simple command line-based program that stores details of stock items as well as staff and store data. The system is designed to have four different user types, each with a different level of privilege and data access. The lowest privileged user, a `Guest` user, can access the system without authentication and has the ability to query stock and leave reviews. The remaining three user types: `Staff`, `Manager`, or `Admin`, have increasing privilege levels but are only accessible through password authentication. An `Admin` user has the highest privilege level and can access and modify all system data. The goal of each exploit

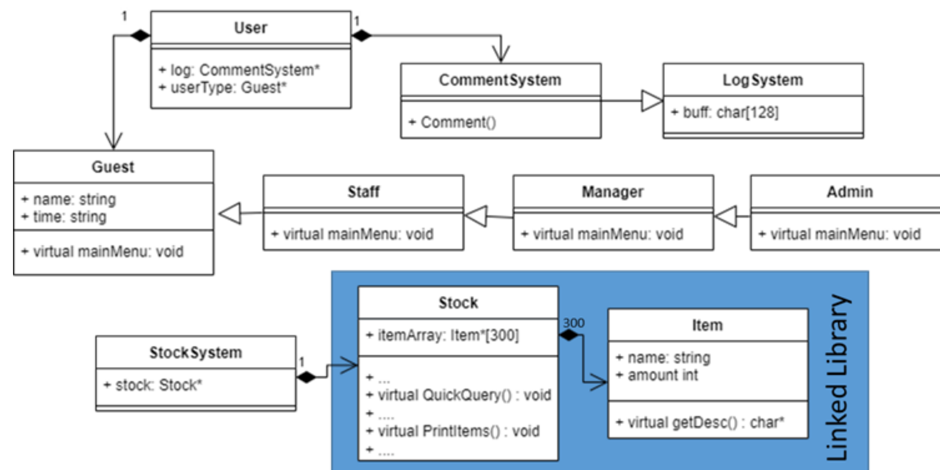


Figure 76: Stock management example program for MFI proof of Concept

will be to access the system as a `Guest` user, bypass the authentication process, and acquire `Admin` level privileges.

Software Design The stock management system relies on inheritance to determine the authority of a user. This inheritance hierarchy can be seen in Figure 76, consisting of the classes `Guest`, `Staff`, `Manager`, and `Admin`. These classes all have their own implementation of a virtual function called `mainMenu`. The `mainMenu` function displays a list of menu options to a system user, which ultimately dictates what they can and cannot access during a session. Figure 77 depicts two of these main menus, one for a guest user (the output of `Guest::mainMenu`) and the other for an admin user (the output of `Admin::mainMenu`). The guest main menu provides a user with limited options and limited access to the system, whereas the admin main menu provides complete access to the whole system. Each of the main menu versions is managed in the same way; the menu items are printed to the terminal, and a unique switch statement processes the user input.

Figure 78 lists the source code for the `main` function and the `User` class in the management system. Upon starting the program, the `main` function creates a new `User` object (line 2). A `User` object stores two pointer variables, `log` and `userType`

```
########################################
######        Guest System        #####
########################################

Main Menu.  Select Your Option  :
 Login                       : Enter 0 :
 Leave a comment/review      : Enter 1 :
 Quick Stock Query           : Enter 2 :
 help                        : Enter h :
 Quit Software               : Enter q :
```

(a) `Guest::mainMenu()`

```
##########################################################
######             Admin System                    #####
######         Authorised Personal Only            #####
###### You must log out before leaving unattented #####
######      or personal details are at risk        #####
##########################################################
#
# Main Menu.  Select Your Option :
# Stock Query                  : Enter 1 :
# Settings                     : Enter 2 :
# Stock System                 : Enter 3 :
# Access Staff Data            : Enter 4 :
# Store Admin                  : Enter 5 :
# help                         : Enter h :
# Log out                      : Enter q :
# Admin:
```

(b) `Admin::mainMenu()`

Figure 77: Differences in main menus for `Guest` and `Admin` users

```
1 int main() {
2   user = new User();
3   stockSystem = new StockSystem();
4
5   while (true) {
6     user->userType->mainMenu();
7   };
8 }
```

(a) `main` function

```
9  class User {
10     public:
11     User() {
12         log = new CommentSystem();
13         userType = new Guest();
14     }
15     CommentSystem *log;
16     Guest* userType;
17 };
```

(b) `User` class and constructor

Figure 78: Source code from sock management system

(lines 15 and 16), where `userType` is initialised to a `Guest` object (line 13). After all objects are constructed, the `main` function will enter an infinite loop (line 5). This infinite loop will call the `mainMenu` function, dictating which menu the user will see and, in turn, their privilege level. Initially, as `user->userType` is assigned a `Guest` object, this function call will bring up the main menu associated with a `Guest` user (Figure 77a). A `Guest` user can opt to log in from this main menu, and upon success, the `user->userType` field will be overwritten to a derived-type (`Staff`, `Manager`, or `Admin`), depending on the credentials used. After login, the control flow returns to the `main` function's infinite loop. As the type stored in `user->userType` has changed, so will the `mainMenu` function dispatched at this location. This is the intended control flow path for users to escalate their privilege level.

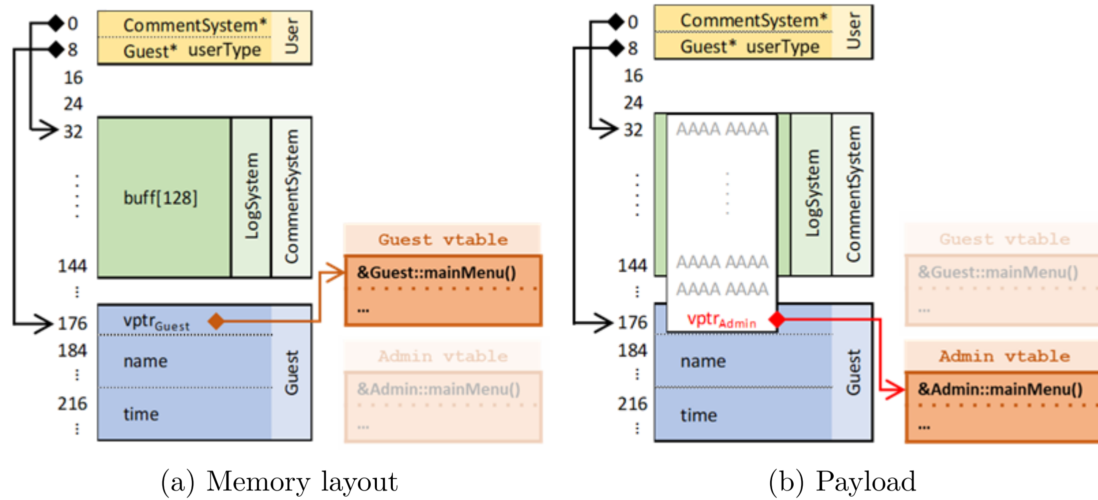
Exploitation Goal The `Admin::mainMenu()` function will be the target member function of each exploit example. This function is prized, as once invoked, an attacker will have full access to the system without any further authentication. Therefore, our goal for each example exploit will be to invoke `Admin::mainMenu()`, as a `Guest` user, without authentication.

Vulnerability The vulnerability in this program is a single buffer overflow inside the `CommentSystem` class (see Figure 76 and lines 12 and 15 in Figure 78b). The `CommentSystem` class inherits from a `Log` class that contains a char buffer of 128 characters. Inside the `CommentSystem` class is a function called `comment()`, which reads user input, and stores it in the inherited buffer but performs no length checks on the user's input string. This contrived scenario represents an easy error in coding, where one programmer makes assumptions about the security of another's code. In this case, the former programmer failed to check the length of user input within their code because they assumed that the `Log` class protected its own char buffer, thus creating a buffer overflow vulnerability.

6.5.2 Exploit 1: COOPlus vptr Overwrite

Recall from Section 6.2.3 that a CFI defence overestimates the set of target functions at a dispatch site to cover all possible control-flow paths of a polymorphic object. A COOPlus exploit takes advantage of this overestimation by targeting derived class functions while using a base class instance, an illegal operation that appears valid under CFI. Our first exploit example does precisely this.

Memory Layout The partial memory layout of the stock management system is depicted in Figure 79a. Here we can see that the `User` object constructed within the `main` function (Figure 78a line 2) exists at offset 0 and addresses both a `CommentSystem` object and a `Guest` object (its data members). The `Guest`

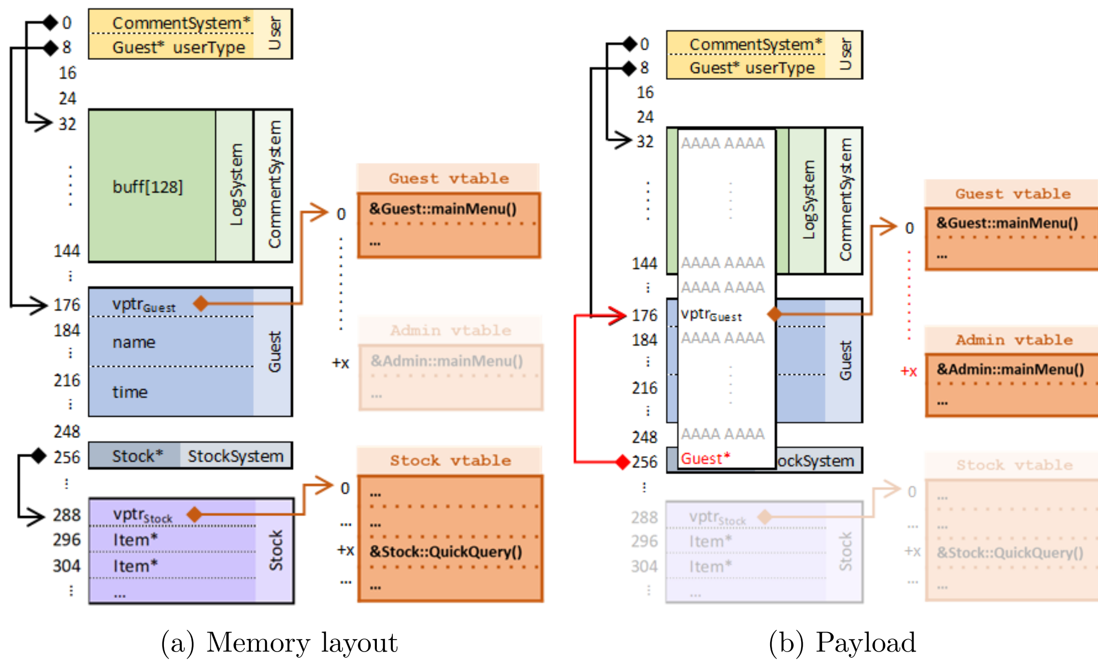


(a) Memory layout (b) Payload

Figure 79: COOPLUS exploit example

object (Figure 79a offset `+176`) has a vptr (`vptr`$_{\texttt{Guest}}$) that addresses the `Guest` vtable. It is this vptr that is used as part of the dynamic dispatch within the infinite loop in the `main` function (line 6, Figure 78a). During the initial run of this loop, the `Guest::mainMenu()` function is invoked, providing guest-level privilege (Figure 77a).

Payload Figure 79b depicts the payload we used to perform the COOPLUS exploit. The `CommentSystem` buffer (at offset `+32`) has a buffer overflow vulnerability, which we, as a `Guest` user, can exploit. To exploit this vulnerability, we write a comment to the buffer that exceeds its maximum length and overwrites the vptr of the adjacent `Guest` object to `vptr`$_{\texttt{Admin}}$ (at offset `+176`). As our `Guest` object now addresses the `Admin` vtable, the `Admin::mainMenu()` function is invoked upon return to the `main` function, granting access to the admin main menu (Figure 77b). Thus as a `Guest` user, we have achieved the highest privilege access (Admin) without authentication.



(a) Memory layout (b) Payload

Figure 80: Vtable out-of-bounds access exploit

6.5.3 Exploit 2: Adjacent Vtable Access

In Section 6.3.3.3, we described an issue within the CFIXX defence that would allow an incorrect virtual function call from an adjacent vtable. An adjacent vtable call happens when a dispatch site accesses a smaller vtable than expected, calling a function entry outside the bounds of that vtable but part of an adjacent vtable instead. This exploit example will demonstrate this scenario.

Memory Layout The `main` function (Figure 78a) initialises a `User` object and a `StockSystem` object (line 3). The `StockSystem` object appears in memory after the `User` object and its fields (Figure 80a offset `+256`) and stores a single pointer to a `Stock` object. As the `Stock` class contains many virtual functions, the `Stock` object addresses a large vtable. A guest user has access to some of these virtual functions, particularly (in this example) the `QuickQuery` function that resides in the `x` entry of the `stock` vtable (see Figure 80a).

Payload Figure 80b depicts the payload used in this exploit example. In this scenario, our payload will overflow the buffer, assign $\texttt{vptr}_{\texttt{Guest}}$ back to itself (at offset `+176`) and continue to overflow until we reach the `StockSystem` object (at offset `+256`). As this object only stores a pointer to another object, we overwrite this pointer to address our `Guest` object instead (situated at offset `+176`).

Invoking the `QuickQuery` Function When a guest user attempts to query the stock in the stock system, the `QuickQuery` function should be dynamically dispatched. The following table outlines what happens during this dispatch, both before and post-payload. The table lists the outcome of each step within the dynamic dispatch process for both scenarios.

		Outcome	
	Performing the `QuickQuery` dynamic dispatch in four steps	**Normal Dispatch (Figure 80a)**	**Dispatch post payload (Figure 80b)**
1	Take the object pointer stored within the `StockSystem` object at offset +256	`Stock*`	`Guest*`.
2	Access the vptr of the retrieved object	$\texttt{vptr}_{\texttt{Stock}}$	$\texttt{vptr}_{\texttt{Guest}}$
3	Add a `+x` offset to the vptr	$\texttt{vptr}_{\texttt{Stock}}$ `+ x`	$\texttt{vptr}_{\texttt{Guest}}$ `+ x`
4	Call the virtual function addressed by the offset vptr	`Stock::QuickQery()`	`Admin::mainMenu()`

As demonstrated, the result of querying the stock after implementing the payload results in calling the `Admin::mainMenu` function. Once again, we have escalated our privilege level, as a guest user, without authentication.

6.5.4 Exploit 3: Unprotected Library

In Section 6.3.3.2, we discussed the issues around unprotected dynamic dispatch sites, stating that it only takes one unprotected dispatch to undermine CFIXX and CFI defences. We will demonstrate this using an unprotected dispatch within a linked library in this example exploit.

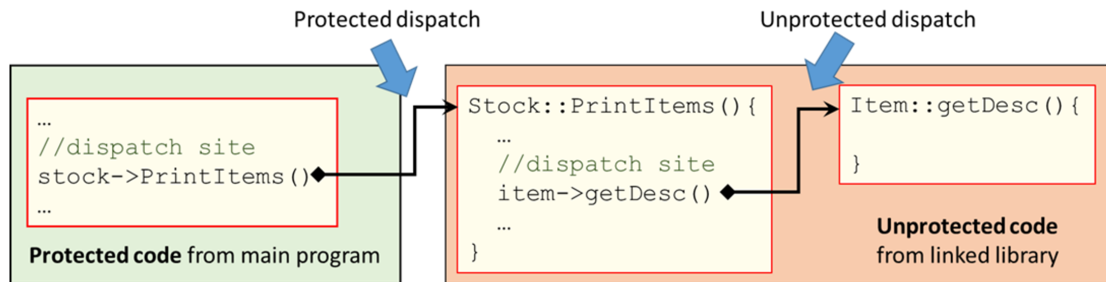
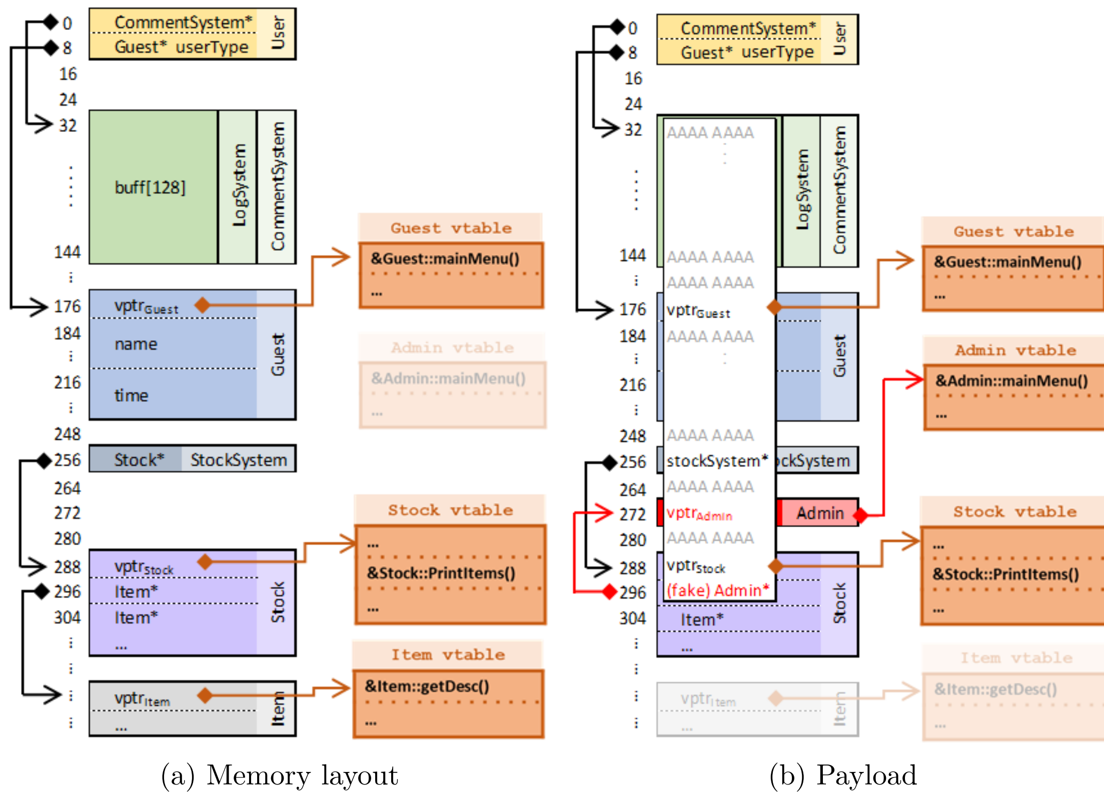


Figure 81: Unprotected dispatch in unprotected library

Dynamic Link Library and Vulnerable Dispatch This example targets a dynamic dispatch within a dynamically linked library. The dispatch is targeted because the library is not compiled with the same protections as the main program. Our example library defines two classes used within the stock management system: `Stock` and `Item`. Both classes have virtual functions, but one in particular, `Stock::PrintItems()`, calls another virtual function (`Item::getDesc()`) within its function body. It is this embedded function call to `getDesc()` that is unprotected. To clarify this, consider Figure 81, which depicts a control flow graph of the dynamic dispatch of the `PrintItems()` function. The call to `PrintItems()` was made within the protected source code; hence, CFIXX and Clang CFI will protect this dispatch site. However, the embedded call to `getDesc()` will not be protected because this call was made within the unprotected library code. Because this function is unprotected, it will be the target of our example exploit.

Memory Layout The `Stock` class stores an array of `Item` pointers, which is reflected in the memory layout depicted in Figure 82a (offset +296). When a user invokes the `Stock::PrintItems()` function, it will iterate through the array of `Item` pointers and dynamically dispatch the virtual function `getDesc()` on each instance. Note that the `getDesc()` function happens to be the first entry of the `Item` vtable.



(a) Memory layout (b) Payload

Figure 82: COOP through unprotected library exploit example

Payload Figure 82b depicts the payload used to perform the third and final exploit. In this case, we perform a more typical COOP exploit by inserting a fake `Admin` object into memory with an `Admin` vptr (as seen at offset `+272`). We also overwrite one of the `Item` pointers (within the `Stock` array) to address our fake `Admin` object.

Calling `PrintItems` Post Payload After inserting the payload (Figure 82b), we can invoke the `Stock::PrintItems()` function. Like before, this function will iterate through the array of `Item` pointers calling the first virtual function entry of the `Item` vtable. However, because we have redirected the first `Item` object to point to our fake `Admin` object, the first virtual function within the `Admin` vtable is called instead (`Admin::mainMenu`). By achieving the `Admin::mainMenu` call, we have again elevated to the highest privilege level without authentication.

6.5.5 MFI - Source-Base Implementation

The MFI proof of concept was implemented as an additional code file within the project. This code file contains the functions that manage the MDT and the SafeSet structure. Safe set data was derived by hand, and all MDT function calls were added to the source manually.

SafeSet structure Figure 83a lists the `SafeSet` struct, containing the `familyIndex` (16-bits), `SafeSetEntry` (16-bits), `classCode` (8-bits), and `safeSetCodeArray[]` (8-bit array). Figure 84a depicts the memory layout of the `SafeSet` struct; these structures will be addressed by run-time objects (via the MDT) and by the member functions (as a hard-coded address).

Safe Set Data Safe set data for the stock management system was derived by hand, using the class hierarchy depicted in Figure 84b, and can be referenced in Figure 84c. The class hierarchy depicts four class families, the largest (derived from `Guest`) is a single inheritance hierarchy made up of 4 classes. Because the maximum number of classes in a family is four, the largest safe set code will flag four relationships, meaning all safe set code arrays will have a maximum of one element[2]. As there is only one element in all safe set code arrays, all safe set entries are set to zero. Note that classes `Stock` and `Item` are not listed with safe set data as these classes are part of an external library, so we do not have access to its code base and, in turn, the ability to protect their member functions with MFI.

Once safe set data was gathered, it was manually added to the source code. Figure 83b lists the source code for the `Admin` class. In line 14, a new `SafeSet ss` data member is added. This data member is declared as both `static` and `const`, meaning the values it stores will never change and will be accessible from

[2]This is reflected in the source code of the `SafeSet` struct with `safeSetCodeArray[1]` defined with one element

```
1  typedef struct SafeSet {
2    const uint16_t familyIndex;
3    const uint16_t SafeSetEntry;
4    const uint8_t classCode;
5    const uint8_t safeSetCodeArray[1];
6    constexpr SafeSet(uint16_t fi,
7    uint16_t sse, uint8_t cc, uint8_t ssc) :
8        familyIndex(fi), SafeSetEntry(sse),
         classCode(cc), safeSetCodeArray {ssc} {}
9  } SafeSet;
10
11 void MFIInitialization();
```

(a) `SafeSet` source code

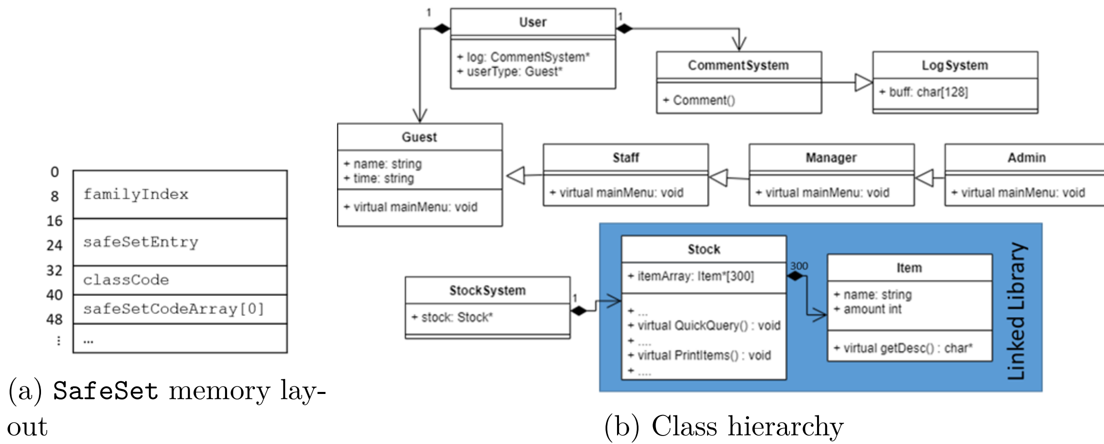
```
12 class Admin: public Manager {
13   public:
14     static const SafeSet ss;
15
16     Admin():Manager() {
17       MFI_AddToMDT(this, ss); ... }
18
19     virtual void Admin::mainMenu() {
20       MFI_verify_call(this,ss); ... }
21     ...
22 };
23 const struct SafeSet Admin::ss = {4,0,8,15};
```

(b) `Admin` source with MFI alterations

```
24 __attribute__((always_inline))
25 inline static void MFI_verify_call(void *thisPtr, const SafeSet& ss_func) {
26     // Look up MDT Entry
27     unsigned long idx1 = (unsigned long)thisPtr >> L2_NUM & L1_MASK;
28     void **level2 = MFILookupStart[idx1];
29     if(level2 == NULL) { // Check level 2 entry exists
30         printf("MFI ERROR: No level 2 pointer found. Exit.");
31         exit(-1);
32     }
33
34     // Grab safe set structure from MDT
35     unsigned idx2 = (unsigned long)thisPtr >> 3 & L2_MASK;
36     idx2 = idx2 << 1;
37     SafeSet *ss_MDT = (SafeSet *)level2[idx2];
38     if(ss_MDT == NULL) { // Check safeset structure is not null
39         printf("MFI ERROR: No level 2 SafeSet entry found. Exit.");
40         exit(-1);
41     }
42
43     if(ss_MDT == &ss_func) // Check if the function and object are the same type. (fast check)
44         return;
45
46     if(ss_MDT->familyIndex != ss_func.familyIndex) { // check family index
47         printf("MFI ERROR: TypeConfusion. Wrong Family Index");
48         exit(-1);
49     }
50
51     uint16_t func_SS_index = ss_func.SafeSetEntry;
52     if(func_SS_index > ss_MDT->SafeSetEntry) { // Check safe set entry exists
53         printf("MFI ERROR: TypeConfusion. Unrelated Function, may belong to a base class");
54         exit(-1);
55     }
56
57     // Check the function class code exists in the object safe set
58     uint8_t MDT_SSC = ss_MDT->safeSetCodeArray[func_SS_index];
59     uint8_t func_cc = ss_func.classCode;
60     if((MDT_SSC ^ func_cc) > MDT_SSC ) {
61         printf("MFI ERROR: TypeConfusion. Unrelated function, may belong to a base class");
62         exit(-1);
63     }
64 }
```

(c) `MFI_verify_call` source code, adapted from Bitype [101]

Figure 83: MFI source code implementation for stock management example



(a) `SafeSet` memory layout

(b) Class hierarchy

Class	familyIndex	safeSetEnrty	classCode	safeSetCodeArray[0]
LogSystem	1	0	0001	0001
CommentSystem	1	0	0010	0011
StockSystem	2	0	0001	0001
User	3	0	0001	0001
Guest	4	0	0001	0001
Staff	4	0	0010	0011
Manager	4	0	0100	0111
Admin	4	0	1000	1111

(c) Safe set data

Figure 84: MFI `SafeSet` implementation for stock management example

all class instances through member functions. If a member function uses the `ss` attribute, it will be realised by the compiler as a hard-coded pointer. As safe set codes provide type information, their hard-coded pointers can be viewed as hard-coded type identification, ultimately making any member function that uses them type-aware.

MDT Management For MDT management, we modified the code published as part of the Bitype paper [101] to work with `SafeSet` objects and perform MFI type testing. The majority of this code remained untouched except for two functions. The first was responsible for adding entries to the MDT; we renamed this function to `MFI_AddToMDT(void *thisPtr, const SafeSet& ss)` and adapted it to store `SafeSet` structures. The second function was responsible for type inclusion testing; this was renamed to `MFI_verify_call(void *thisPtr, const`

`SafeSet& ss)` and altered to perform the MFI type inclusion tested outlined in Section 6.4.5. Both of these functions were integrated into the `Admin` class (Figure 83b). `MFI_AddToMDT` was added as the first line of the constructor (line 17), and `MFI_verify_call` was added as the first line of every member function (line 20). Noticed that both these functions are passed the `this` keyword (i.e. the object's address-point) and the static const safe set attribute (`ss`) as parameters.

The `MFI_verify_call` function is listed in Figure 83c. Again, this function takes two parameters: the object's address-point (`thisPtr`), and the function's safe set attribute (`ss_func`). Using the `thisPtr` in lines 26-41, another pointer to a `SafeSet` structure is copied from the MDT and stored in the variable `ss_MDT` (safe set from MDT). The function will then proceed to perform the type inclusion test. First, to save time, line 43 checks whether `ss_MDT` and `ss_func` address the same safe set object. If they do, then the object and the function are from the same class, and execution can continue. If they are not the same class type, then a full type inclusion test is performed. First, the family indexes are compared in line 46, followed by the safe set entries on line 52, and finally, a relationship check between function and object in line 60. Only if all three tests pass will the function be allowed to continue to execute.

6.5.6 Defence Comparison

We tested all three exploits against CFIXX, Clang CFI, and our MFI prototype; the results are in Table 21.

CFIXX Recall that any object construction under CFIXX protection will store its vptr in a secure metadata table and redirect any dynamic dispatch site through that table. This combination protects the integrity of an object's type and does not allow valid vptrs to be altered. As a result, CFIXX can protect against the first exploit type that altered an object's vptr. However, CFIXX cannot protect

	Successful prevention?		
	CFIXX	Clang CFI	MFI
Exploit 1 COOPlus	✓	✗	✓
Exploit 2 Adjacent Vtable	✗	✓	✓
Exploit 3 Unprotected Library	✗	✗	✓

Table 21: MFI, CFIXX, and Clang CFI comparison against three different exploits

against the second exploit because it does not protect the control flow of a dynamic dispatch, i.e. a dispatch site can receive an incorrect object type and invoke a different function to the expected control flow.

Clang CFI In contrast to the CFIXX defence, Clang CFI protects the control flow of a dynamic dispatch site but not the integrity of the object types. Clang CFI will ensure that the vptr used in a dynamic dispatch site will be from a set of expected vptrs. This set consists of the vptr associated with the function's type and vptrs from all derived class types (to accommodate for polymorphism).

This means that Clang CFI can protect against our second exploit, as $\texttt{vptr}_{\texttt{Guest}}$ is not one of the expected vptrs when trying to dispatch the `QuickQuery` function. On the other hand, Clang CFI cannot protect against the first exploit, as a base class vptr ($\texttt{vptr}_{\texttt{Admin}}$) is one of the expected vptr types that could be used to invoke the `mainMenu()` function.

Exploit 3 CFIXX and Clang CFI could not protect against the third exploit as both defences can only accommodate libraries compiled with the respective defence strategies. This exploit again highlights the most significant flaw of these defences: it only takes one unprotected dispatch site to allow an attacker to invoke any member function within the program.

MFI MFI has been designed to perform type-checking after a dynamic dispatch. Ideally, this type check would be performed within the prologue of a function being called, but as our proof of concept is source based, we insert our type-checking

function as the first line of code in each member function. Because the type checks exist within the function, dynamic dispatch sites have no control-flow protection; thus, anything goes at the point of dispatch. However, the moment a member function begins execution, the function will ensure that the object it is interacting with is of a valid type and will stop execution immediately if invalid.

Our MFI prototype was able to prevent all three types of exploits discussed. Because MFI performs all type-safety checks post dynamic dispatch, it does not have the same weaknesses as the other defences. In all three exploits, the `Admin::MainMenu()` is successfully dispatched by the attacker, but this function stopped execution the moment it realised it received an invalid object type. In both the first and second exploit, MFI checks and determines that the `Guest` objects used are from the same family, but would fail the legal relationship test as they are not an instance of the derived class `Admin`, and in turn, the program was stopped. In the third exploit, we attempted to inject a fake `Admin` object. This object was not created using a constructor, so its type data will not exist in the MFI metadata table. Thus, when the `Admin::mainMenu()` function is called, the `MFI_verify_call` function will fail before any type of testing is performed, because no MDT entry could be found.

Why MFI is Superior In our MFI-hardened stock management system, an attacker can still exploit the vulnerable dynamic dispatch within the linked library to invoke non-MFI-protected functions. However, this means they are restricted to invoking only the member functions within the unprotected library (i.e. functions from classes `Stock` and `Item`). This fact makes MFI significantly more secure than CFIXX and Clang CFI. In CFIXX and Clang CFI programs, a single vulnerable dispatch site could allow any member function in the program to be invoked. However, in MFI, a single vulnerable dispatch can only exploit unprotected functions, and unprotected functions do not undermine the security of other functions

with MFI protection. This makes MFI a better security strategy for protecting member functions from code reuse attacks.

6.6 Future Work

This chapter presented the proposal and proof of concept for a defence technique against member function reuse attacks such as COOP. As it is just a proposal, its implementation and evaluation are obvious future work challenges.

Implementation We believe MFI is best suited to compiler-level implementation, which would facilitate automatic MDT management and insertion of type inclusion tests. For the MDT, further work is required to evaluate its coverage and management of stack-allocated objects. Also, further investigation is required to determine whether a CFIXX-style metadata table is secure enough for this defence, as MDT integrity will be paramount.

Evaluation This chapter outlines a fine-grained MFI defence. Fine-grained, in this setting, means that every object is traced, and every member function performs a type inclusion test. Tracking every object and performing a type test within every function will undoubtedly incur overhead, but only experimentation can determine whether its cost is acceptable. Of course, the Bitype [101] type inclusion testing scheme is just one of many schemes that could be adapted and adopted for MFI testing. Further research is needed to identify whether this is the best scheme for the job and whether further optimisation could be applied.

Coarse-grained MFI If a fine-grained MFI defence adds an unacceptable overhead, then a coarse-grained MFI defence could provide an appropriate compromise. A coarse-grained MFI would limit its deployment to specific classes and member functions. Which classes and member functions are protected will

require further research, but we have a possible solution based on how COOP vfgadgets are categorised. Schuster et al. [112] categorise different vfgadgets into types based on their operation. These types include but are not limited to main loop gadgets, arithmetic operations, register load, register read, invocation of a function via a field, etc. We theorise that every COOP exploit has a minimal set of gadget types that must be executed before the attacker can gain complete control. If this minimal set of gadget types could be identified, then only the member functions that match those gadget types need protection (and their class instances tracked). By protecting a strict subset of member functions, an attacker would not be able to complete a full gadget chain required to gain full program control.

Microsoft C++ ABI MFI, in its current conceptual form, will not be compatible with the Microsoft ABI [86]. Most notably, it is incompatible with overridden virtual functions dispatched from a virtually inherited sub-object. These virtual functions receive an address that is offset from an object's address-point. These offset addresses do not correlate with those used during object construction and, therefore, will not have their valid MDT entry. In the Itanium ABI, for which MFI was designed, functions are guaranteed to receive an address-point pointing to an object type that matches the function's type; this is not guaranteed for programs compiled under the Microsoft C++ ABI. Applying MFI to software compiled to conform to the Microsoft ABI requires further consideration and possible alterations to the design.

As we have already discussed, a course-grained MFI solution is possible, and unprotected functions do not diminish the integrity of protected ones. As overridden virtual functions in virtual hierarchies are the issue for MFI compatibility in Microsoft's ABI, one solution is not to apply MFI protection in those specific cases. This solution is the easiest to implement, but further research is required to

identify how many functions this would likely leave vulnerable. During the course of our research, virtual inheritance rarely appeared in the projects we analysed. However, this is a tiny sample and may not be emblematic of all programs. A more thorough survey of the popularity of virtual inheritance may be required before a blanket omission of these functions is considered a viable solution.

Another solution, one that would require more research and ingenuity, is to track these offset locations like we would for an object's address-point. The difficulty is that these offset locations are not used as part of the constructor function (unlike actual object address-points) and must therefore be identified through different mechanisms. Additionally, if such offsets were tracked, they could not address the same safe sets as the object address-points. Recall that MFI safe sets represent the class relationships shared by a common address-point, not the whole hierarchy. As these offsets will solely be used by overridden virtual functions in virtual hierarchies, they need their own unique safe sets, and in turn, so do the virtual functions that use them to perform the type testing. An issue that could arise from this is a clash between an offset location and a real address-point. In such cases, safe sets would need to be merged to allow both types of function calls, something that is not possible under the current design. Trying to deal with these rogue functions dramatically increases the complexity of the MFI defence. More research into the mechanics of these problematic function calls is needed to design an appropriate solution for programs compiled under the Microsoft ABI.

Preventing Type Confusion Another branch of research is MFI's capability of preventing type confusion. Although MFI is not a type confusion detector, the defence should be triggered when a confused object is used within a member function. This delayed response will not pick up the original vulnerability (i.e. the point in the program an object becomes confused) but, downstream, will prevent a confused object from being used within a member function. Member functions

operating on object instances are the core of any OO program; thus, we believe that MFI can go beyond just a COOP mitigation and protect the integrity of the object types themselves.

Binary Rewriting A natural progression of research for source-based defences is implementing it as a binary-level defence through a binary rewriting tool. A significant benefit to defensive binary rewriting is that it allows modern defence policies to be applied retrospectively to legacy code. However, most binary-level defences fall short due to coverage issues. Pawlowski et al. [104] built a binary rewriter capable of implementing CFIXX's defence policy called VPS (VTable Pointer Separation), reporting an average of 97-98% coverage for all dynamic dispatch sites. As we have discussed, coverage issues can be a significant flaw in defences such as CFIXX and Clang CFI, as a single vulnerable dispatch can open up all member functions to a code reuse attack. However, this is not the case for MFI, and as we have shown, the integrity of MFI-hardened functions is upheld, even in partially protected code. This provides a unique opportunity for further work into applying the MFI defence within a binary rewriting tool, as, unlike other defences in this space, the inevitable partial coverage of binary rewriting would not undermind the defensive policy.

6.7 Concluding Discussion

This chapter discussed COOPLUS, a new variant of the COOP exploit that can bypass almost all modern C++-semantic-aware CFI defences. One of the few defences that could prevent a COOPLUS attack was CFIXX, which we examined to expose some vulnerabilities and shortcomings. In response to these findings, we designed our own defence policy, MFI, proposed a method for its implementation and demonstrated its capabilities with a proof of concept. As part of the proof of concept, we compared our MFI defence with the CFIXX and Clang CFI defences,

providing examples of exploits that MFI can protect against, but could bypass both CFIXX and Clang CFI. Thus we have demonstrated that MFI has a superior security strategy for protecting member functions from code reuse attacks.

The MFI defence strategy is unique, as it is the only defence strategy (to our knowledge) that protects member functions post-dispatch. What makes MFI so powerful is that its integrity did not falter in the presents of vulnerable code (within unprotected linked libraries), unlike other defences. Its ability to prevent such threats is a direct consequence of moving type tests to member functions so that they can be performed post-dispatch. This design choice not only provided a powerful feature but a multitude of avenues for further research, which we hope has inspired others to pursue.

Part III

Reflection

Chapter 7

Concluding Discussion and Future Work

7.1 Conclusion

7.1.1 Low-level C++ Implementation

To guard against common C++-specific bugs and exploits, a low-level understanding of C++ is vital. Despite this, research papers often briefly explain low-level C++ features specific to their work, with no forward references for further reading. Their limited explanations leave readers with an abstracted and incomplete perception of low-level C++. Where papers did provide references, they all point to a standard reference book, which is now over 25 years old and focuses on compilers that have long been discontinued. It was clear that the topic of low-level C++ within a modern-day compiler needed revisiting, and this was the premise behind the first half of this thesis.

We presented a modern-day look at low-level C++ object orientation (OO) on modern-day compilers. Discussions were limited to object layouts and standardised OO features to narrow our focus on this broad topic. Nevertheless, we

attempted to unveil each level of abstraction, from machine to source, for a more well-rounded and in-depth understanding of C++ OO. We hope this contribution will aid in better low-level C++ understanding, better C++ code and better security in the future.

7.1.2 MemCast

Securing the integrity of all object types of a C++ program is paramount in preventing type confusion vulnerabilities and code reuse attacks, particularly COOP. At the source level, compilers perform type checks to verify the correctness of all static object types and their uses. However, protecting the dynamic types (of polymorphic objects) is not supported, so it falls to the programmer to call type verification functions explicitly. Standardised dynamic type verification in C++ relies on run-time type information (RTTI), where type checks incur a search through the tree structure of RTTI objects. Dynamic casting incurs such costs while verifying an object's relationship with the cast's target type. This cost, for some developers, is deemed prohibitively high and therefore, dynamic casting is avoided.

We presented a novel profiling measurement called cast stability that quantifies the degree to which the source type changes from one dynamic dispatch call to the next. We applied this analysis to the Deal.II library and, surprisingly, found that most cast sites were 100% stable, and all but a few were at least 55% stable. This was significant, as it reveals a gross redundancy in dynamic casting, as every single cast incurs an RTTI search despite performing an identical type check in its previous execution.

In light of our stability revelations, we designed and implemented MemCast, an optimisation technique for the dynamic cast operator, which exploits highly stable casts to reduce the costs of a validated cast to that of dynamic dispatch.

We also evaluated the true cost of dynamic casting and compared it with MemCasting, presenting evidence that MemCasting would outperform dynamic casting in as few as seven visits for a cast of 50% stability. This allowed us to apply a blanket change to the Deal.II library (as all but one casts sites were at least 55% stable), swapping all dynamic down-casts with a MemCast call. With this change, we achieved an average run-time speedup between 1.63-1.68%. To further demonstrate MemCast's capabilities, we applied the blanket change to two other large C++ libraries, OMNet++ and Antlr4. The resulting performance speedup in these cases was between 1.11-3.91%. Arguably the message of this speedup is not their absolute values, but rather, the developer can apply validated casting without undue consideration of the cost.

7.1.3 MFI

We discussed both Clang CFI [131], a C++-semantic-aware CFI defence, and CFIXX [16]), a type integrity defence that protects virtual pointers, and their ability to defend against member function reuse attacks. When these defences are deployed together, they complement each other and protect the defects in the other's defences. Clang CFI can be bypassed using a COOPPlus (a COOP variant) attack, which CFIXX can defend. CFIXX can be bypassed by dispatching virtual functions from an outer bound vtable read, which Clang CFI can prevent. However, the most significant flaw featured in both defences is that they need complete coverage, i.e. every single dispatch site must deploy the respective defences. If an attacker finds a single vulnerable dispatch site (perhaps due to an unprotected linked library), then every virtual function in the program is open to an attacker for code reuse.

We presented a novel defence policy, Member Function Integrity (MFI), that brings type-awareness to member functions and the ability to verify the object types they receive. We provided a detailed implementation proposal, evidenced

how MFI can protect against the flaws discussed in Clang CFI and CFIXX, and provided a scalable version of the defence for large hierarchies. We finished the chapter with a proof of concept. We built a simple program with a vulnerability that enabled three exploitation techniques, COOPlus, out-of-bounds vtable call, and using an unprotected library. This program was compiled separately with each defence: Clang CFI, CFIXX, and MFI (as a source-based prototype) and then compared the effectiveness of each defence against each exploit. The results demonstrated that Clang CFI and CFIXX could prevent one of the three exploits, whereas MFI could prevent all three, demonstrating that MFI has a superior security strategy for protecting member functions from code reuse attacks.

7.2 Future Work

7.2.1 MemCast

MemCast Implementation We believe that the argument for MemCast is so compelling that it is a candidate for inclusion in C++ itself. This would confer several advantages:

1. MemCache management could be shifted from the programmer to the compiler itself, saving the programmer concerns about the setup and management of MemCache objects.
2. Under compiler control, MemCast could profit from the optimisations that benefit dynamic casting, such as devirtualising up-casts.
3. It would allow other defences, such as CFI, to be integrated with MemCasting, strengthening its security and defending against code reuse attacks.

Gaining a Better Understanding of MemCast's Capabilities Our experimental testing revealed that large and bottom-heavy hierarchies generated RTTI structures of low memory locality. Where dynamic casting interacted with these

cases of low locality, casting speeds appeared impeded. Thus we inferred a correlation between the two, prompting future research opportunities on the impact of low locality RTTI objects in dynamic casting. Many questions emerge from this topic, such as: How prevalent is low RTTI locality in large code bases? At what size and shape does a hierarchy impact RTTI locality? At what measurable locality of RTTI objects is dynamic casting significantly affected, and by how much? How does low locality dynamic casting compare to high locality casts with long RTTI traversals? And what improvements can MemCast make to low locality cast sites? The list goes on. By answering these questions, we can better understand the capabilities of MemCast as a performance-enhancing tool in larger code bases.

Further Optimisations Another opportunity for study, again related to locality, is MemCache locality. What difference does it make when locality is considered when placing MemCache objects together? Is it best to place MemCache objects based on the proximity of their dynamic cast sites? Or should dynamic cast sites be ranked on their usage and their MemCache objects position so that the most used are bundled together? Either way, additional research into optimal memory locality could further improve the speedups seen with MemCasting.

7.2.2 MFI

MFI Implementation, Performance and Coverage Our MFI proposal provides the most opportunities for future work and research. The most pressing research question, post-implementation of MFI, is its overhead and whether it is acceptable for real-world deployment. After this, the question is how to extend MFI to non-virtual functions and other ABI implementations, such as Microsoft's ABI.

Course-Grained MFI Beyond MFI implementation is the idea of a minimum vfgadget set, a set of vfgadget types required in any successful COOP exploit. If such a set exists, the question is, can a course-grained MFI defence, which protects only vfgadgets types from this set, provide enough protection to mitigate all COOP and COOPPlus exploits? The answer to this question would be most interesting if fine-grained MFI is shown to have excessive performance overheads. In such a case, course-grained MFI defence could reduce overheads sufficiently to make the defence more viable.

MFI for Microsoft's C++ ABI We designed MFI around the Itanium C++ ABI [23], which has different object layouts and function conventions from Microsoft's ABI [86]. These differences in convention mean that Microsoft ABI programs will require more complex MDT management and more safe set structures. If one wishes to expand MFI to Microsoft ABI programs, more work is necessary to understand its calling conventions, particularly for virtual functions based in virtually inherited classes. Such functions are different in Microsoft's C++ ABI because they are not guaranteed to receive an object's address-point but could receive an offset from it instead. At first glance, accommodating such functions will require more safe set data to be generated and mapped to these specific offset locations. But how we encode hierarchical relationship data for these addresses and decide which functions will use them in their type-safety checks is yet to be understood. Accommodating the Microsoft ABI will ultimately impact the simplicity of MFI's current implementation proposal; management of the MDT will likely have to change, constructor functions will need to adapt, and safe set data collection will need a new algorithm to identify all class address-point relationships and the member functions they relate too.

Can MFI Protect Against Type Confusion Vulnerabilities? Finally, we would like to see an evaluation of MFI's ability to prevent memory corruption

brought on by type confusion. Under the MFI scheme, there are no preventative measures against type confusion vulnerabilities; an object's type can become confused without immediate detection. However, when a confused object is used within a member function call, MFI will identify a problem and immediately stop execution. Therefore, MFI has multiple uses for protecting type integrity. However, if utilised to prevent a type confusion vulnerability, all member functions (virtual and non-virtual) would require protection (i.e. a fine-grained MFI defence). Further investigation is required to determine the impact and overhead such a scheme would achieve and, again, whether its performance is acceptable.

Final Comment We hope this MFI proposal can guide further research into type integrity and contribute to the improvement of C++ security as a whole.

Bibliography

[1] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. L. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS* (2005), V. Atluri, C. A. Meadows, and A. Juels, Eds., ACM, pp. 340–353.

[2] Alex. The virtual table, February 2023. Learn C++ (Online Blog) `https://www.learncpp.com/cpp-tutorial/the-virtual-table/` Accessed: 22-04-2023.

[3] Almakhdhub, N. S., Clements, A. A., Bagchi, S., and Payer, M. μRAI: Securing Embedded Systems with Return Address Integrity. In *27th Annual Network and Distributed System Security Symposium, NDSS* (2020), The Internet Society.

[4] Arm Ltd. *C++ ABI for the Arm 64-bit Architecture (AArch64)*, Second quarter ed., 2020. `https://developer.arm.com/documentation/ihi0059/latest`.

[5] Arndt, D., Bangerth, W., Blais, B., Clevenger, T. C., Fehling, M., Grayver, A. V., Heister, T., Heltai, L., Kronbichler, M., Maier, M., Munch, P., Pelteret, J.-P., Rastak, R., Thomas, I., Turcksin, B., Wang, Z., and Wells, D. The `deal.II` Library, Version 9.2. *Journal of Numerical Mathematics 28*, 3 (2020), 131–146. `https://dealii.org/deal92-preprint.pdf`.

[6] AT&T. *UNIX System V AT&T C++ Translator Release Notes*, 1985. http://www.softwarepreservation.org/projects/c_plus_plus/cfront/release_1.0/doc/ReleaseNotes-Lifeboat.pdf.

[7] AT&T. *System V Application Binary Interface*, 4.1 ed., 1997. http://www.sco.com/developers/devspecs/gabi41.pdf.

[8] Barnes, D. J., and Kölling, M. *Objects First with Java - A Practical Introduction Using BlueJ (5th Edition)*. Prentice Hall, 2012.

[9] Bauer, M., and Rossow, C. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking. In *IEEE European Symposium on Security and Privacy* (2021), IEEE, pp. 650–666.

[10] Biallas, S., Olesen, M. C., Cassez, F., and Huuck, R. Ptrtracker: Pragmatic pointer analysis. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2013), pp. 69–73.

[11] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS* (2011), ACM, pp. 30–40.

[12] Booth, K. S., and Lueker, G. S. Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms. *J. Comput. Syst. Sci. 13*, 3 (1976), 335–379.

[13] Bounov, D., Kici, R. G., and Lerner, S. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *23rd Annual Network and Distributed System Security Symposium* (2016), The Internet Society.

[14] Bounov, D., Kici, R. G., and Lerner, S. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *23rd Annual Network and Distributed System Security Symposium, NDSS* (2016), The Internet Society.

[15] Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR) 50*, 1 (2017), 16:1–16:33.

[16] Burow, N., McKee, D., Carr, S. A., and Payer, M. CFIXX: Object Type Integrity for C++. In *25th Annual Network and Distributed System Security Symposium, NDSS* (2018), The Internet Society.

[17] Burow, N., Zhang, X., and Payer, M. SoK: Shining Light on Shadow Stacks. In *Symposium on Security and Privacy* (2019), IEEE, pp. 985–999.

[18] Caseau, Y. Efficient Handling of Multiple Inheritance Hierarchies. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference* (1993), T. Babitsky and J. Salmons, Eds., ACM, pp. 271–287.

[19] Cattiaux, C., and Szkudlapski, K. Visual C++ RTTI Inspection, July 2013. (Online Blog) `https://blog.quarkslab.com/visual-c-rtti-inspection.html` Accessed: 22-04-2023.

[20] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS* (2010), E. Al-Shaer, A. Al-Shaer, and V. Shmatikov, Eds., ACM, pp. 559–572.

[21] Chen, K., Zhang, C., Yin, T., Chen, X., and Zhao, L. VScape: Assessing and Escaping Virtual Call Protections. In *30th USENIX Security*

Symposium, USENIX (2021), M. Bailey and R. Greenstadt, Eds., USENIX Association, pp. 1719–1736.

[22] Chrome infrastructure. Chromium Bugs - dynamic_cast used. https://bugs.chromium.org/p/chromium/issues/detail?id=9270 Accessed: 04-05-2023.

[23] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI;. *Itanium C++ ABI (Draft)*, 2017. https://itanium-cxx-abi.github.io/cxx-abi/.

[24] Cohen, N. H. Type-Extension Type Tests Can Be Performed In Constant Time. *ACM Transactions on Programming Languages and Systems 13*, 4 (1991), 626–629.

[25] CVE. Common Vulnerabilities and Exposures, 2021. Online Resource. https://cve.mitre.org/ Accessed: 09-04-2021.

[26] Dahl, O.-J. The Birth of Object Orientation: the Simula Language. In *From Object-Orientation to Formal Methods*, vol. 2635 of *Lecture Notes in Computer Science*. Springer, 2004, pp. 15–25.

[27] Davi, L., Koeberl, P., and Sadeghi, A.-R. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *The 51st Annual Design Automation Conference, DAC* (2014), ACM, pp. 133:1–133:6.

[28] Deal.II Documentation. The deal.II Library: Class Hierarchy. https://www.dealii.org/9.2.0/doxygen/deal.II/hierarchy.html Accessed: 07-02-2023.

[29] Dean, J., DeFouw, G., Grove, D., Litvinov, V., and Chambers, C. Vortex: An Optimizing Compiler for Object-Oriented Languages. In

Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA (1996), L. Anderson and J. Coplien, Eds., ACM, pp. 83–100.

[30] Debian Manpages. grep Unix Utility. `https://manpages.debian.org/testing/grep/grep.1.en.html` Accessed: 02-03-2021.

[31] Debian Manpages. multitime Unix Utility. `https://manpages.debian.org/testing/multitime/multitime.1.en.html` Accessed: 15-12-2021.

[32] Dechev, D., Mahapatra, R. N., and Stroustrup, B. Practical and Verifiable C++ Dynamic Cast for Hard Real-Time Systems. *J. Comput. Sci. Eng. 2*, 4 (2008), 375–393.

[33] Doxygen. Doxygen. `https://www.doxygen.nl/` Accessed: 02-03-2023.

[34] Duck, G. J., and Yap, R. H. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2018), J. S. Foster and D. Grossman, Eds., ACM, pp. 181–195.

[35] Ducournau, R. Perfect Hashing as an Almost Perfect Subtype Test. *ACM Transactions on Programming Languages and Systems 30*, 6 (2008), 33:1–33:56.

[36] Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[37] Elsabagh, M., Fleck, D., and Stavrou, A. Strict Virtual Call Integrity Checking for C++ Binaries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, Eds., ACM, pp. 140–154.

[38] Erinfolami, R. A., Quach, A. T., and Prakash, A. On Design Inference from Binaries Compiled using Modern C++ Defenses. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID* (2019), USENIX Association, pp. 15–30.

[39] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO* (2016), IEEE Computer Society, pp. 40:1–40:13.

[40] Fan, X., Sui, Y., Liao, X., and Xue, J. 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* (2017), T. Bultan and K. Sen, Eds., ACM, pp. 329–340.

[41] Farkhani, R. M., Jafari, S., Arshad, S., Robertson, W., Kirda, E., and Okhravi, H. On the Effectiveness of Type-Based Control Flow Integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference* (2018), ACSAC, Association for Computing Machinery, p. 28–39.

[42] Fog, A. Calling Conventions for Different C++ Compilers and Operating Systems. In *Software Optimization Resources.* Technical University of Denmark, 2014. `https://www.agner.org/optimize/calling_conventions.pdf`.

[43] Gawlik, R., and Holz, T. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC* (2014), C. N. Payne, A. Hahn, K. R. B. Butler, and M. Sherr, Eds., ACM, pp. 396–405.

[44] GCC Project. GNU Compiler Collection (GCC). `https://github.com/gcc-mirror/gcc` Accessed: 22-11-2021.

[45] Gibbs, M., and Stroustrup, B. Fast dynamic casting. *Softw. Pract. Exp. 36*, 2 (2006), 139–156.

[46] Godbolt, M. Compiler Explorer. `https://godbolt.org/` Accessed: 02-03-2023.

[47] Goldthwaite, L. Technical Report on C++ Performance. Tech. Rep. TR 18015:2004, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), 2004. `http://www.speling.org/JTC1/SC22/WG21/docs/papers/2004/n1666.pdf`.

[48] Gray, J. C++: Under the Hood, March 1994. (Archived Visual C++ Technical Article) `http://www.openrce.org/articles/files/jangrayhood.pdf` Accessed: 22-04-2023.

[49] Haller, I., Jeon, Y., Peng, H., Payer, M., Giuffrida, C., Bos, H., and Van Der Kouwe, E. TypeSan: Practical type confusion detection. In *Conference on Computer and Communications Security* (2016), Association for Computing Machinery, pp. 517–528.

[50] Henning, J. L. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News 34*, 4 (2006), 1–17.

[51] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy* (2016), IEEE Computer Society, pp. 969–986.

[52] Ingerman, P. Z. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Commun. ACM 4*, 1 (1961), 55–58.

[53] Intel Corporation. *Align and Organize Data for Better Performance.* Intel Official Online Articles for Development Topics & Technologies. https://software.intel.com/content/www/us/en/develop/articles/align-and-organize-data-for-better-performance.html Accessed: 20-08-2021.

[54] Intel Corporation. *Intel Itanium Processor-specific Application Binary Interface (ABI)*, 2001. https://www.uclibc.org/docs/psABI-ia64.pdf.

[55] Intel Corporation. Introduction to Intel memory protection extensions, 2013. Web Archive https://web.archive.org/web/20190116155131/http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions Accessed: 11-11-2021.

[56] Intel Corporation. Intel Memory Protection Extensions Enabling Guide, 2016. https://www.intel.com/content/www/us/en/developer/articles/guide/intel-memory-protection-extensions-enabling-guide.html Accessed: 11-11-2021.

[57] ISO/IEC. *Working Draft, Standard for Programming Language C++ (C++23).* International Organization for Standardization/International Electrotechnical Commission, 2020. https://github.com/cplusplus/draft/releases/tag/n4868.

[58] Jang, D., Tatlock, Z., and Lerner, S. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *21st Annual Network*

and Distributed System Security Symposium, NDSS (2014), The Internet Society.

[59] Jeffers, J., Reinders, J., and Sodani, A. Chapter 4 - Knights Landing architecture. In *Intel Xeon Phi Processor High Performance Programming*, second ed. Morgan Kaufmann, 2016, pp. 63–84.

[60] Jeon, Y., Biswas, P., Carr, S., Lee, B., and Payer, M. HexType: Efficient Detection of Type Confusion Errors for C++. In *Conference on Computer and Communications Security* (2017), CCS '17, Association for Computing Machinery, p. 2373–2387.

[61] Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook: The art of automatic memory management.* Chapman and Hall / CRC Applied Algorithms and Data Structures Series. CRC Press, 2011.

[62] Khandaker, M., Naser, A., Liu, W., Wang, Z., Zhou, Y., and Cheng, Y. Adaptive Call-Site Sensitive Control Flow Integrity. In *IEE European Symposium on Security and Privacy, EuroS&P* (2019), IEEE, pp. 95–110.

[63] Khandaker, M. R., Liu, W., Naser, A., Wang, Z., and Yang, J. Origin-sensitive Control Flow Integrity. In *28th USENIX Security Symposium, USENIX Security* (2019), N. Heninger and P. Traynor, Eds., USENI Association, pp. 195–211.

[64] Kirzner, O., and Morrison, A. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *30th USENIX Security Symposium (USENIX Security)* (August 2021), USENIX Association, pp. 2399–2416.

[65] Krall, A., and Grafl, R. CACAO - A 64-bit JavaVM Just-in-Time Compiler. *Concurr. Pract. Exp. 9*, 11 (1997), 1017–1030.

[66] Krall, A., Vitek, J., and Horspool, R. N. Near Optimal Hierarchical Encoding of Types. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings* (1997), M. Aksit and S. Matsuoka, Eds., vol. 1241 of *Lecture Notes in Computer Science*, Springer, pp. 128–145.

[67] Kusswurm, D. *Modern X86 Assembly Language Programming: Covers X86 64-bit, AVX, AVX2, and AVX-512*. Apress, 2018.

[68] Kwon, A., Dhawan, U., Smith, J. M., Knight, T. F., and DeHon, A. Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security. In *ACM SIGSAC Conference on Computer & Communications Security* (2013), Association for Computing Machinery, p. 721–732.

[69] Lan, B., Li, Y., Sun, H., Su, C., Liu, Y., and Zeng, Q. Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses. In *IEEE TrustCom/BigDataSE/ISPA* (2015), IEEE, pp. 190–197.

[70] Larabel, M. Intel MPX Support Will Be Removed From Linux - Memory Protection Extensions Appear Dead - Phoronix Media, 2018. `https://www.phoronix.com/scan.php?page=news_item&px=Intel-MPX-Kernel-Removal-Patch` Accessed: 11-11-2021.

[71] Le Goc, Y., and Donzé, A. EVL: A framework for multi-methods in C++. *Sci. Comput. Program. 98* (2015), 531–550.

[72] Lee, B., Song, C., Kim, T., and Lee, W. Type Casting Verification: Stopping an Emerging Attack Vector. In *24th USENIX Security Symposium* (Aug. 2015), USENIX Association, pp. 81–96.

[73] Lenkov, D., Mehta, M., and Unni, S. Type Identification in C++. In *Proceedings of the C++ Conference* (1991), USENIX Association, pp. 103–118.

[74] Lippman, S. B. *Inside the C++ Object Model.* Addison-Wesley, 1996.

[75] LLVM Developer Group. The LLVM Project. `https://github.com/llvm/llvm-project` Accessed: 22-11-2021.

[76] LLVM Project. How to set up LLVM-style RTTI for your class hierarchy. `https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html` Accessed: 03-05-2023.

[77] Luo, B., Yang, Y., Zhang, C., Wang, Y., and Zhang, B. A survey of code reuse attack and defense. In *Advances in Intelligent, Interactive Systems and Applications* (Cham, 2019), F. Xhafa, S. Patnaik, and M. Tavana, Eds., Springer International Publishing, pp. 782–788.

[78] Martin, R. C. Java and C++ A critical comparison. *Technical Note, Object Mentor* (March 1997).

[79] Matz, M., Hubička, J., Jaeger, A., and Mitchell, M. *System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.7.* SUSE Software Solutions Germany and CodeSourcery, 2014. `https://www.uclibc.org/docs/psABI-x86_64.pdf`.

[80] Meyers, S. The most important c++ software...ever, 2006. `https://www.artima.com/articles/the-most-important-c-softwareemeverem` Accessed: 28-10-2021).

[81] Microsoft. *x64 Software Conventions.* Microsoft official online documentation, 2018. `https://docs.microsoft.com/en-gb/cpp/build/x64-software-conventions` Accessed: 07-06-2021.

[82] Microsoft. *Calling Conventions.* Microsoft official online documentation, 2019. `https://learn.microsoft.com/en-us/cpp/cpp/calling-conventions` Accessed: 02-03-2023.

[83] Microsoft. *Microsoft C/C++ Change History 2003-2015.* Microsoft official online documentation, 2019. `https://docs.microsoft.com/en-gb/cpp/porting/visual-cpp-change-history-2003-2015` Accessed: 08-06-2021.

[84] Microsoft. *Overview of Potential Upgrade Issues (Visual C++).* Microsoft official online documentation, 2019. `https://docs.microsoft.com/en-us/cpp/porting/overview-of-potential-upgrade-issues-visual-cpp` Accessed: 06-08-2021.

[85] Microsoft. *Data Execution Prevention.* Microsoft official online documentation, 2021. `https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention` Accessed: 10-20-2021.

[86] Microsoft. *Microsoft C/C++ Documentation.* Microsoft official online documentation, 2021. `https://docs.microsoft.com/en-gb/cpp/` Accessed: 06-07-2021.

[87] Miller, M. R., Johnson, K. D., and Burrell, T. W. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities, 2014. US Patent 8,683,583B2.

[88] Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K. W., and Franz, M. Opaque Control-Flow Integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015), The Internet Society.

[89] Moreno, C., and Fischmeister, S. Accurate Measurement of Small Execution Times — Getting Around Measurement Errors. *IEEE Embedded Systems Letters 9*, 1 (2017), 17–20.

[90] Muntean, P., Wuerl, S., Grossklags, J., and Eckert, C. Cast-San: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS Barcelona, Spain* (2018), J. López, J. Zhou, and M. Soriano, Eds., vol. 11098, Springer, pp. 3–25.

[91] NASA and Caltech. Jet Propulsion Laboratory (JPL) - California Institute of Technology. `https://www.jpl.nasa.gov/` Accessed: 10-12-202.

[92] Nergal. Advanced return-into-lib(c) exploits (PaX case study). *Phrack Magazine 11*, 58 (2001), 4–14. `http://phrack.org/issues/58/4.html` Accessed: 10-20-2021.

[93] Niu, B., and Tan, G. Modular Control-Flow Integrity. In *SIGPLAN Conference on Programming Language Design and Implementation,PLDI* (2014), M. F. P. O'Boyle and K. Pingali, Eds., ACM, pp. 577–587.

[94] Niu, B., and Tan, G. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), I. Ray, N. Li, and C. Kruegel, Eds., ACM, pp. 914–926.

[95] Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., and Fetzer, C. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst. 2*, 2 (2018), 28:1–28:30.

[96] One, A. Smashing The Stack For Fun And Profit. *Phrack Magazine 7*, 49 (1996), 14–16. `http://phrack.org/issues/49/14.html` Accessed: 10-20-2021.

[97] OPEN RCE - IGOTSK. Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI, September 2006. (Online Blog) `http://www.openrce.org/articles/full_view/23` Accessed: 22-04-2023.

[98] OPENSIM LTD. OMNeT++ Discrete Event Simulator. `https://omnetpp.org/` Accessed: 02-03-2023.

[99] OPENSIM LTD. What is OMNeT++? `https://omnetpp.org/intro` Accessed: 02-03-2023.

[100] PADHYE, R., AND SEN, K. Efficient Fail-Fast Dynamic Subtype Checking. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (2019), D. Bonetta and Y. D. Liu, Eds., ACM, pp. 32–37.

[101] PANG, C., DU, Y., MAO, B., AND GUO, S. Mapping to Bits: Efficiently Detecting Type Confusion Errors. In *34th Annual Computer Security Applications Conference* (2018), ACSAC '18, Association for Computing Machinery, p. 518–528.

[102] PARR, T. ANTLR. `https://www.antlr.org/` Accessed: 28-04-2023.

[103] PARR, T. *The definitive ANTLR 4 reference*. The Pragmatic Bookshelf, 2013.

[104] PAWLOWSKI, A., VAN DER VEEN, V., ANDRIESSE, D., VAN DER KOUWE, E., HOLZ, T., GIUFFRIDA, C., AND BOS, H. VPS: excavating high-level C++ constructs from low-level binaries to protect dynamic dispatching. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC* (2019), D. Balenson, Ed., ACM, pp. 97–112.

[105] PAX. Address space layout randomization, 2001. `https://pax.grsecurity.net/docs/aslr.txt` Accessed: 11-18-2021.

[106] Prakash, A., Hu, X., and Yin, H. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015), The Internet Society.

[107] Qt Group . Cross-platform Software Design and Development Tools. `https://www.qt.io/` Accessed: 22-04-2023.

[108] Raymond, E. S. The Lost Art of Structure Packing, January 2014. (Online Blog) `http://www.catb.org/esr/structure-packing/` Accessed: 22-04-2023.

[109] Reeves, J. W. Multiple Inheritance Considered Useful, February 2006. (Online Blog) `https://www.drdobbs.com/cpp/multiple-inheritance-considered-useful/184402074` Accessed: 22-04-2023.

[110] Sadeghi, A., Niksefat, S., and Rostamipour, M. Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. *J. Comput. Virol. Hacking Tech. 14*, 2 (2018), 139–156.

[111] Schubert, L. K., Papalaskaris, M. A., and Taugher, J. Determining Type, Part, Color and Time Relationships. *Computer 16*, 10 (1983), 53–60.

[112] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A., and Holz, T. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE Computer Society, pp. 745–762.

[113] Shacham, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS* (2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM, pp. 552–561.

[114] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS* (2004), V. Atluri, B. Pfitzmann, and P. D. McDaniel, Eds., ACM, pp. 298–307.

[115] Skochinsky, I. Compiler Internals: Exceptions and RTTI, 2012. (Online Blog) `http://www.hexblog.com/wp-content/uploads/2012/06/Recon-2012-Skochinsky-Compiler-Internals.pdf` Accessed: 22-04-2023.

[116] Sleator, D. D., and Tarjan, R. E. Self-Adjusting Binary Trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA* (1983), ACM, pp. 235–245.

[117] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy, SP* (2013), IEEE Computer Society, pp. 574–588.

[118] Solar Designer. *Getting around non-executable stack (and fix)*. Bugtraq mailing list archives, 1997. `https://seclists.org/bugtraq/1997/Aug/63` Accessed: 10-20-2021.

[119] Solodkyy, Y., Reis, G. D., and Stroustrup, B. Mach7: Pattern Matching for C++. `https://github.com/solodon4/Mach7` Accessed: 10-03-2021.

[120] SONG, D., LETTNER, J., RAJASEKARAN, P., NA, Y., VOLCKAERT, S., LARSEN, P., AND FRANZ, M. SoK: Sanitizing for Security. *IEEE Symposium on Security and Privacy, San Francisco* (2019), 1275–1295.

[121] STACK OVERFLOW. MSVC Object Layout Quirk, February 2010. (Online Forum) `https://stackoverflow.com/questions/2250931/msvc-object-layout-quirk` Accessed: 22-04-2023.

[122] STANDARD C++ FOUNDATION. Programming Languages–C++ (C++98). Tech. Rep. 14882, International Organization for Standardization, International Organization for Standardization/International Electrotechnical Commission, 1998. `https://www.iso.org/standard/25845.html`.

[123] STANDARD PERFORMANCE EVALUATION CORPORATION. *SPEC CPU2006*, 2018. `https://www.spec.org/cpu2006/` Accessed: 09-09-2021.

[124] STROUSTRUP, B. Classes: An Abstract Data Type Facility for the C Language. *SIGPLAN Not. 17*, 1 (Jan. 1982), 42–51.

[125] STROUSTRUP, B. Adding Classes to the C Language: An Exercise in Language Evolution. *Software: Practice and Experience 13*, 2 (1983), 139–161.

[126] STROUSTRUP, B. A History of C++: 1979-1991. In *Proceedings of History of Programming Languages Conference (HOPL-II)* (1993), J. A. N. Lee and J. E. Sammet, Eds., Association for Computing Machinery, pp. 271–297.

[127] STROUSTRUP, B. *The Design and Evolution of C++*. Addison-Wesley, New Jersey, 1994.

[128] STROUSTRUP, B., AND LENKOV, D. Run Time Type Identification for C++. In *Proceedings of the C++ Conference* (1992), USENIX Association, pp. 313–340.

[129] Taylor, I. *64-bit PowerPC ELF Application Binary Interface Supplement*, 1.7.1 ed. Zembu Labs, 2004. `https://www.uclibc.org/docs/psABI-ppc64.pdf`.

[130] The Clang Team. *Clang 13 documentation*. Official Online Documentation for LLVM & Clang. `https://clang.llvm.org/docs/ClangTools.html` Accessed: 13-08-2021.

[131] The Clang Team. *Control Flow Integrity*. Official Online Documentation for LLVM & Clang, 2021. `https://clang.llvm.org/docs/ControlFlowIntegrity.html` Accessed: 04-01-2021.

[132] The Clang Team. *Undefined Behavior Sanitizer*. Official Online Documentation for LLVM & Clang, 2021. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html` Accessed 04-01-2021.

[133] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium* (2014), K. Fu and J. Jung, Eds., USENIX Association, pp. 941–955.

[134] V8. Retiring octane, April 2017. Official V8 online blog archive: `https://v8.dev/blog/retiring-octane` Accessed: 10-08-2021.

[135] Van Der Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., and Giuffrida, C. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *IEEE Symposium on Security and Privacy, SP* (2016), IEEE Computer Society, pp. 934–953.

[136] Vishnyakov, A. V., and Nurmukhametov, A. R. Survey of methods for automated code-reuse exploit generation. *Program. Comput. Softw. 47*, 4 (2021), 271–297.

[137] Vitek, J., Horspool, R. N., and Krall, A. Efficient Type Inclusion Tests. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)* (1997), M. E. S. Loomis, T. Bloom, and A. M. Berman, Eds., ACM, pp. 142–157.

[138] Wang, C., Chen, B., Liu, Y., and Wu, H. Layered object-oriented programming: Advanced vtable reuse attacks on binary-level defense. *IEEE Transactions on Information Forensics and Security 14*, 3 (2019), 693–708.

[139] Wirth, N. 3. *ACM Transactions on Programming Languages and Systems 10*, 2 (1988), 204–214.

[140] Zhang, C., Song, C., Chen, K. Z., Chen, Z., and Song, D. VTint: Protecting Virtual Function Tables' Integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015), The Internet Society.

[141] Zhang, C., Song, D., Carr, S. A., Payer, M., Li, T., Ding, Y., and Song, C. VTrust: Regaining Trust on Virtual Calls. In *23rd Annual Network and Distributed System Security Symposium, NDSS* (2016), The Internet Society.

[142] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy, SP* (2013), IEEE Computer Society, pp. 559–573.

[143] Zibin, Y., and Gil, J. Efficient Subtyping Tests with PQ-Encoding. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA* (2001), L. M. Northrop and J. M. Vlissides, Eds., ACM, pp. 96–107.

[144] Zou, C., Sui, Y., Yan, H., and Xue, J. TCD: Statically Detecting Type Confusion Errors in C++ Programs. In *30th IEEE International Symposium on Software Reliability Engineering* (2019), K. Wolter, I. Schieferdecker, B. Gallina, M. Cukier, R. Natella, N. Ramezani Ivaki, and N. Laranjeiro, Eds., IEEE, pp. 292–302.

Appendix A

Behind Object Abstraction

A.1 RTTI Hierarchy

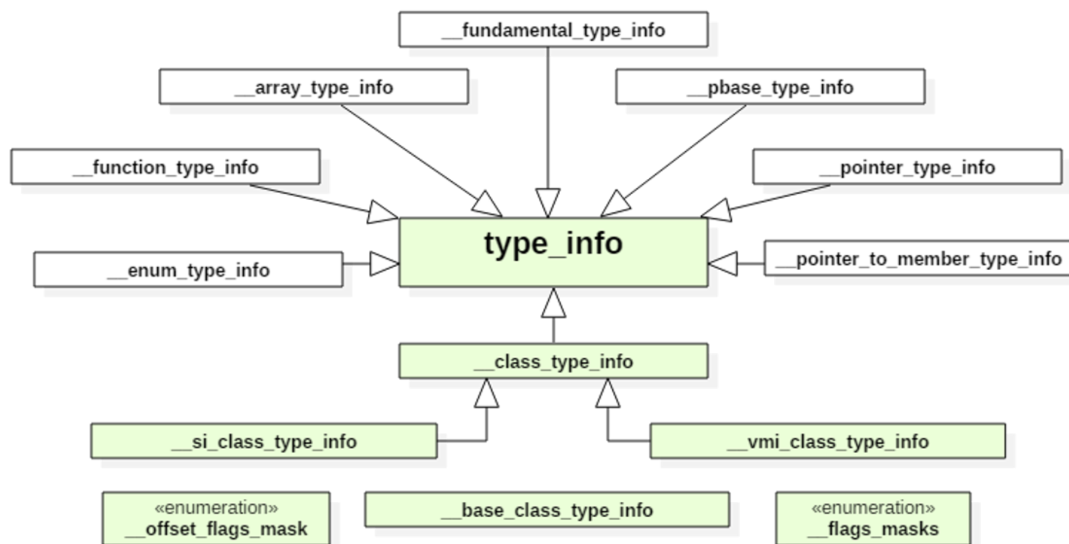


Figure 85: Full Inheritance Hierarchy for RTTI in Itanium ABI [23]

A.2 Full Virtual Inheritance Constructor Call

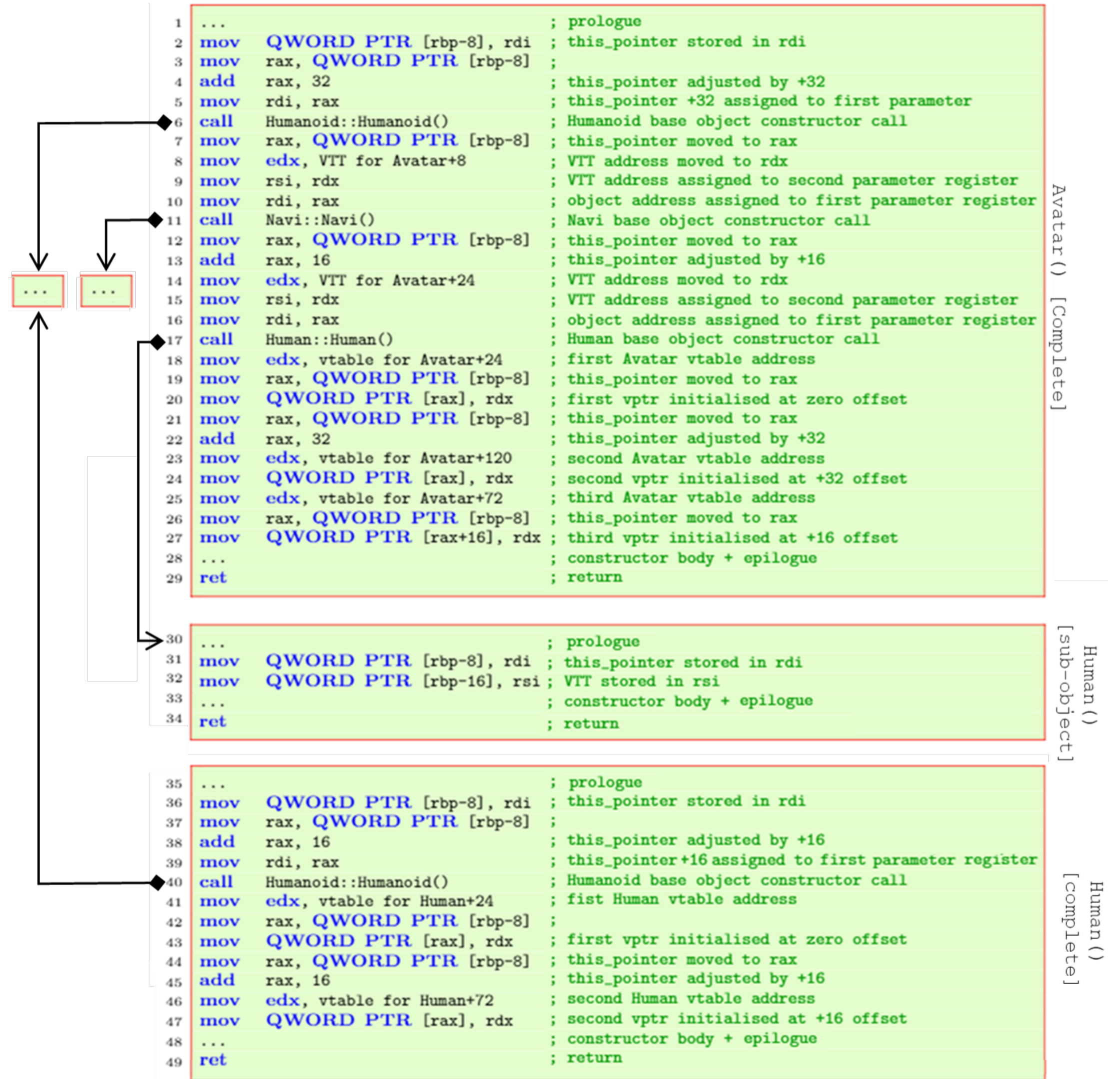


Figure 86: Avatar nested constructor call with virtual inheritance

Appendix B

Deall.II Full Results

Here we present the full results of our analysis of the Deal.II library and test programs, detailing all the dynamic down-cast locations within the code and their stability values in each test case.

B.1 Dynamic Down-Cast Locations

Table 22 displays the location of every dynamic down-cast found in the Deal.II library (version 9.2.0 [5]), alongside each step-x example program the cast is featured in.

Table 22: Deal.II dynamic down-cast locations and featured step-x programs

Cast Number	Source Location : Line Number	Features in Step-x
1	source/dofs/dof_accessor_get.cc:58	3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 20, 21, 23, 24, 25, 26, 27, 30, 38, 39, 47, 48, 51, 52, 61, 65, 67
2	include/deal.II/lac/affine_constraints.templates.h:3695	6, 8, 9, 11, 12, 16, 21, 23, 24, 25, 26, 27, 47, 51, 52, 61, 65
3	source/fe/fe_q.cc:189	6, 7, 8, 9, 13, 14, 15, 16, 26, 27, 48
4	source/fe/fe_q.cc:198	6, 7, 8, 9, 13, 14, 15, 16, 26, 27, 48
5	source/dofs/dof_accessor_set.cc:55	15, 26
6	include/deal.II/fe/fe_poly.templates.h:254	14, 39, 47
7	include/deal.II/fe/fe_poly.h:258	3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 20, 21, 23, 24, 25, 26, 27, 30, 38, 39, 47, 48, 51, 52, 61, 65, 67

Continued on next page

Table 22 – continued from previous page

Dynamic cast Number	Source Location : Line Number	Features in Step-x
8	source/fe/mapping_q_generic.cc:2679	3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 21, 23, 24, 25, 26, 27, 30, 38, 39, 47, 48, 51, 52, 61, 65, 67
9	source/fe/mapping_q_generic.cc:2695	3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 21, 23, 24, 26, 27, 30, 38, 39, 47, 51, 52, 61, 65
10	include/deal.II/fe/fe_poly_tensor.h:252	20, 21, 61
11	source/fe/fe_q_base.cc:594	6, 7, 8, 9, 13, 14, 15, 16, 26, 27, 48
12	source/fe/fe_system.cc:927	8, 20, 21, 51, 61, 67
13	source/fe/fe_system.cc:990	8, 20, 21, 51, 61
14	include/deal.II/numerics/vector_tools_interpolate.templates.h:179	26, 48
15	source/dofs/dof_handler.cc:1287	2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 21, 23, 24, 25, 26, 30, 38, 39, 47, 48, 51, 52, 61, 65, 67
16	source/fe/mapping_q_generic.cc:2713	6, 7, 8, 12, 13, 14, 15, 16, 26, 27, 30, 39, 47, 51
17	source/fe/fe_q_base.cc:477	14
18	source/dofs/dof_handler.cc:843	2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 21, 23, 24, 25, 26, 30, 38, 39, 47, 48, 51, 52, 61, 65, 67
19	source/dofs/dof_handler.cc:849	2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 21, 23, 24, 25, 26, 30, 38, 39, 47, 48, 51, 52, 61, 65, 67
20	include/deal.II/matrix_free/shape_info.templates.h:140	10, 11, 23, 24, 25, 38, 48, 65, 67
21	include/deal.II/matrix_free/shape_info.templates.h:148	10, 11, 23, 24, 25, 38, 48, 65, 67
22	include/deal.II/matrix_free/shape_info.templates.h:154	10, 11, 23, 24, 25, 38, 48, 65, 67
23	include/deal.II/matrix_free/shape_info.templates.h:156	10, 11, 23, 24, 25, 38, 48, 65, 67
24	include/deal.II/numerics/vector_tools_project.templates.h:966	21, 23, 24, 25
25	include/deal.II/numerics/vector_tools_project.templates.h:968	21, 23, 24, 25
26	include/deal.II/numerics/vector_tools_project.templates.h:971	21, 23, 24, 25
27	source/fe/mapping_cartesian.cc:99	20, 21
28	include/deal.II/lac/la_parallel_vector.templates.h:1312	23, 24, 25
29	include/deal.II/lac/la_parallel_vector.templates.h:1580	23, 24, 25
30	include/deal.II/matrix_free/mapping_info.templates.h:2510	23, 24, 25, 48, 67
31	include/deal.II/matrix_free/mapping_info.templates.h:359	23, 24, 25, 48, 67
32	include/deal.II/matrix_free/matrix_free.templates.h:558	23, 24, 25
33	include/deal.II/fe/fe_poly.h:258	23, 24, 25, 48, 67, 12b, 16b
34	include/deal.II/fe/fe_poly.templates.h:254	20, 21, 16b
49	include/deal.II/base/utilities.h:784	10, 11, 24, 38, 47
54	include/deal.II/lac/affine_constraints.templates.h:3695	16b
70	include/deal.II/fe/fe_poly_face.h:116	51, 61
115	include/deal.II/lac/la_parallel_vector.templates.h:1379	24, 25
121	include/deal.II/lac/la_parallel_vector.templates.h:1489	24, 25, 67
123	include/deal.II/lac/la_parallel_vector.templates.h:1557	24, 25, 48
126	include/deal.II/lac/la_parallel_vector.templates.h:1631	24, 25
128	include/deal.II/lac/la_parallel_vector.templates.h:1841	24, 25
130	include/deal.II/lac/la_parallel_vector.templates.h:1844	24, 25
141	include/deal.II/meshworker/integration_info.h:608	12b, 16b
142	include/deal.II/meshworker/integration_info.h:610	12b, 16b
143	include/deal.II/meshworker/integration_info.h:612	12b, 16b
161	include/deal.II/meshworker/integration_info.h:608	39
162	include/deal.II/meshworker/integration_info.h:610	39
163	include/deal.II/meshworker/integration_info.h:612	39

Continued on next page

Table 22 – continued from previous page

Dynamic cast Number	Source Location : Line Number	Features in Step-x
224	source/dofs/dof_accessor_get.cc:58	12b, 16b
227	source/dofs/dof_handler.cc:1287	12b, 16b
228	source/dofs/dof_handler.cc:1319	16b
234	source/dofs/dof_handler.cc:843	12b, 16b
235	source/dofs/dof_handler.cc:849	12b, 16b
241	source/dofs/dof_handler.cc:1319	16, 39
324	source/fe/fe_face.cc:175	51
330	source/fe/fe_face.cc:449	51
346	source/fe/fe_q.cc:189	16b
347	source/fe/fe_q.cc:198	16b
349	source/fe/fe_nothing.cc:198	10
351	source/fe/fe_q_base.cc:594	16b
359	source/fe/fe_q_base.cc:695	27
361	source/fe/fe_q_base.cc:735	27
403	source/fe/fe_system.cc:1053	8, 51
410	source/fe/fe_system.cc:2007	8
412	source/fe/fe_system.cc:2247	8
455	source/fe/mapping_q.cc:156	10, 11, 38, 47
456	source/fe/mapping_q.cc:188	10, 11, 38, 47
457	source/fe/mapping_q.cc:221	47
461	source/fe/mapping_q.cc:393	11, 38, 47
465	source/fe/mapping_q.cc:442	47
468	source/fe/mapping_q_generic.cc:2679	12b, 16b
469	source/fe/mapping_q_generic.cc:2695	12b, 16b
470	source/fe/mapping_q_generic.cc:2713	12b, 16b
479	source/fe/mapping_q_generic.cc:4023	6, 10, 11, 65, 67
511	source/hp/dof_handler.cc:1019	27
513	source/hp/dof_handler.cc:1645	27
514	source/hp/dof_handler.cc:1721	27
515	source/hp/dof_handler.cc:1755	27
516	source/hp/dof_handler.cc:980	27
522	source/multigrid/mg_level_global_transfer.cc:87	16, 39, 16b
525	source/multigrid/mg_transfer_internal.cc:218	16, 39, 16b

B.2 Stability of Each Dynamic Down-Cast Site

Table 23 displays the stability of every cast site found in the Deal.II library (listed in Table 22) and which step-x example it features in.

Table 23: Stability of every casts site for each step-x program

Cast Num	Step	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
1	3	1023	0	100.00%
1	4	4351	1	99.98%
1	6	23111	0	100.00%
1	7	492791	0	100.00%
1	8	148451	0	100.00%
1	9	212628	0	100.00%
1	11	40949	0	100.00%
1	12	14009	0	100.00%
1	13	226809	0	100.00%
1	14	187559	0	100.00%
1	15	10799477	0	100.00%
1	16	47101	0	100.00%
1	20	3071	0	100.00%
1	21	691967	0	100.00%
1	23	10485759	0	100.00%
1	24	35915999	0	100.00%
1	25	49983	0	100.00%
1	26	668860	0	100.00%
1	27	37841	0	100.00%
1	30	42644	0	100.00%
1	38	3839	0	100.00%
1	39	1259	0	100.00%
1	47	5439	0	100.00%
1	48	1109663	0	100.00%
1	51	720771	0	100.00%
1	52	58879	0	100.00%
1	61	12287	0	100.00%
1	65	132351	0	100.00%
1	67	45055	0	100.00%
2	6	7527	0	100.00%
2	8	47509	0	100.00%
2	9	57564	0	100.00%
2	11	20474	0	100.00%
2	12	9468	0	100.00%
2	16	34729	0	100.00%
2	21	1023	0	100.00%
2	23	32767	0	100.00%
2	24	163839	0	100.00%
2	25	127	0	100.00%
2	26	142037	0	100.00%
2	27	7589	0	100.00%

Cast Num	Steps	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
2	47	4199	0	100.00%
2	51	46471	0	100.00%
2	52	511	0	100.00%
2	61	1023	0	100.00%
2	65	13311	0	100.00%
3	6	5007	0	100.00%
3	7	22551	0	100.00%
3	8	5295	0	100.00%
3	9	32623	0	100.00%
3	13	11635	0	100.00%
3	14	27419	0	100.00%
3	15	336289	0	100.00%
3	16	5419	0	100.00%
3	26	31465	0	100.00%
3	27	15323	0	100.00%
3	48	783	0	100.00%
4	6	5007	0	100.00%
4	7	22551	0	100.00%
4	8	5295	0	100.00%
4	9	32623	0	100.00%
4	13	11635	0	100.00%
4	14	27419	0	100.00%
4	15	336289	0	100.00%
4	16	5419	0	100.00%
4	26	31465	0	100.00%
4	27	15323	0	100.00%
4	48	783	0	100.00%
5	15	709999	0	100.00%
5	26	62417	0	100.00%
6	14	14495	0	100.00%
6	39	1259	0	100.00%
6	47	8159	0	100.00%
7	3	2	0	100.00%
7	4	5	1	80.00%
7	6	43	0	100.00%
7	7	217	0	100.00%
7	8	43	0	100.00%
7	9	67	0	100.00%
7	11	89	0	100.00%
7	12	1609	0	100.00%
7	13	126	0	100.00%

Continued on next page

Table 23 – continued from previous page

Cast Num	Step	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)	Cast Num	Steps	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
7	14	135	0	100.00%	9	4	1	1	0.00%
7	15	289	0	100.00%	9	6	21	0	100.00%
7	16	67	0	100.00%	9	7	87	0	100.00%
7	20	4	0	100.00%	9	8	21	0	100.00%
7	21	523	0	100.00%	9	9	19	0	100.00%
7	23	1603	0	100.00%	9	10	6	0	100.00%
7	24	35262	0	100.00%	9	11	29	0	100.00%
7	25	419	0	100.00%	9	12	23	0	100.00%
7	26	1259	0	100.00%	9	13	40	0	100.00%
7	27	215	0	100.00%	9	14	49	0	100.00%
7	30	5586	0	100.00%	9	15	109	0	100.00%
7	38	3	0	100.00%	9	16	21	0	100.00%
7	39	616	0	100.00%	9	20	0	0	N/A
7	47	59	0	100.00%	9	21	125	0	100.00%
7	48	73	0	100.00%	9	23	639	0	100.00%
7	51	951	0	100.00%	9	24	0	0	N/A
7	52	7475	0	100.00%	9	26	351	0	100.00%
7	61	7	0	100.00%	9	27	153	0	100.00%
7	65	13	0	100.00%	9	30	16	0	100.00%
7	67	10	0	100.00%	9	38	0	0	N/A
8	3	0	0	N/A	9	39	335	0	100.00%
8	4	1	1	0.00%	9	47	19	0	100.00%
8	6	7	0	100.00%	9	51	46639	0	100.00%
8	7	111	0	100.00%	9	52	0	0	N/A
8	8	7	0	100.00%	9	61	6	0	100.00%
8	9	37	0	100.00%	9	65	5	0	100.00%
8	10	6	0	100.00%	10	20	18	0	100.00%
8	11	113	0	100.00%	10	21	275	0	100.00%
8	12	1555	0	100.00%	10	61	17	0	100.00%
8	13	56	0	100.00%	11	6	13	0	100.00%
8	14	60	0	100.00%	11	7	31	0	100.00%
8	15	153	0	100.00%	11	8	27	0	100.00%
8	16	31	0	100.00%	11	9	17	0	100.00%
8	20	14	0	100.00%	11	13	21	0	100.00%
8	21	146	0	100.00%	11	14	75	0	100.00%
8	23	645	0	100.00%	11	15	25	0	100.00%
8	24	105123	0	100.00%	11	16	13	0	100.00%
8	25	367	0	100.00%	11	26	101	0	100.00%
8	26	605	0	100.00%	11	27	86	0	100.00%
8	27	20	0	100.00%	11	48	1	0	100.00%
8	30	5523	0	100.00%	12	8	15	0	100.00%
8	38	2	0	100.00%	12	20	3	0	100.00%
8	39	120	0	100.00%	12	21	135	0	100.00%
8	47	19	0	100.00%	12	51	269	0	100.00%
8	48	38	0	100.00%	12	61	3	0	100.00%
8	51	269	0	100.00%	12	67	10	0	100.00%
8	52	7244	0	100.00%	13	8	21	0	100.00%
8	61	17	0	100.00%	13	20	0	0	N/A
8	65	7	0	100.00%	13	21	125	0	100.00%
8	67	11	0	100.00%	13	51	137	0	100.00%
9	3	0	0	N/A	13	61	3	0	100.00%

Continued on next page

Table 23 – continued from previous page

Cast Num	Step	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
14	26	107	0	100.00%
14	48	30823	0	100.00%
15	2	0	0	N/A
15	3	0	0	N/A
15	4	1	1	0.00%
15	6	7	0	100.00%
15	7	27	0	100.00%
15	8	7	0	100.00%
15	9	9	0	100.00%
15	10	47	0	100.00%
15	11	17	0	100.00%
15	12	5	0	100.00%
15	13	18	0	100.00%
15	14	17	0	100.00%
15	15	13	0	100.00%
15	16	7	0	100.00%
15	20	0	0	N/A
15	21	0	0	N/A
15	23	0	0	N/A
15	24	0	0	N/A
15	25	0	0	N/A
15	26	51	0	100.00%
15	30	11	0	100.00%
15	38	0	0	N/A
15	39	11	0	100.00%
15	47	3	0	100.00%
15	48	0	0	N/A
15	51	89	0	100.00%
15	52	0	0	N/A
15	61	1	0	100.00%
15	65	1	0	100.00%
15	67	0	0	N/A
16	6	5	0	100.00%
16	7	13	0	100.00%
16	8	5	0	100.00%
16	12	23	0	100.00%
16	13	9	0	100.00%
16	14	15	0	100.00%
16	15	11	0	100.00%
16	16	5	0	100.00%
16	26	49	0	100.00%
16	27	19	0	100.00%
16	30	16	0	100.00%
16	39	119	0	100.00%
16	47	15	0	100.00%
16	51	7	0	100.00%
17	14	32	0	100.00%
18	2	0	0	N/A
18	3	0	0	N/A
18	4	1	1	0.00%
18	6	0	0	N/A

Cast Num	Steps	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
18	7	3	0	100.00%
18	8	0	0	N/A
18	9	0	0	N/A
18	10	7	0	100.00%
18	11	2	0	100.00%
18	12	0	0	N/A
18	13	1	0	100.00%
18	14	1	0	100.00%
18	15	0	0	N/A
18	16	0	0	N/A
18	20	0	0	N/A
18	21	0	0	N/A
18	23	0	0	N/A
18	24	0	0	N/A
18	25	0	0	N/A
18	26	0	0	N/A
18	30	1	0	100.00%
18	38	0	0	N/A
18	39	0	0	N/A
18	47	0	0	N/A
18	48	0	0	N/A
18	51	8	0	100.00%
18	52	0	0	N/A
18	61	1	0	100.00%
18	65	0	0	N/A
18	67	0	0	N/A
19	2	0	0	N/A
19	3	0	0	N/A
19	4	1	1	0.00%
19	6	0	0	N/A
19	7	3	0	100.00%
19	8	0	0	N/A
19	9	0	0	N/A
19	10	7	0	100.00%
19	11	2	0	100.00%
19	12	0	0	N/A
19	13	1	0	100.00%
19	14	1	0	100.00%
19	15	0	0	N/A
19	16	0	0	N/A
19	20	0	0	N/A
19	21	0	0	N/A
19	23	0	0	N/A
19	24	0	0	N/A
19	25	0	0	N/A
19	26	0	0	N/A
19	30	1	0	100.00%
19	38	0	0	N/A
19	39	0	0	N/A
19	47	0	0	N/A
19	48	0	0	N/A

Continued on next page

Table 23 – continued from previous page

Cast Num	Step	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)		Cast Num	Steps	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
19	51	8	0	100.00%		26	24	0	0	N/A
19	52	0	0	N/A		26	25	0	0	N/A
19	61	1	0	100.00%		27	20	2	0	100.00%
19	65	0	0	N/A		27	21	2	0	100.00%
19	67	0	0	N/A		28	23	1	0	100.00%
20	10	2	0	100.00%		28	24	0	0	N/A
20	11	47	0	100.00%		28	25	0	0	N/A
20	23	3	3	0.00%		29	23	1	0	100.00%
20	24	1	1	0.00%		29	24	1	0	100.00%
20	25	1	1	0.00%		29	25	1	0	100.00%
20	38	2	0	100.00%		30	23	1	0	100.00%
20	48	1	1	0.00%		30	24	0	0	N/A
20	65	7	0	100.00%		30	25	0	0	N/A
20	67	15	3	80.00%		30	48	0	0	N/A
21	10	2	0	100.00%		30	67	0	0	N/A
21	11	47	0	100.00%		31	23	1	0	100.00%
21	23	3	3	0.00%		31	24	0	0	N/A
21	24	1	1	0.00%		31	25	0	0	N/A
21	25	1	1	0.00%		31	48	0	0	N/A
21	38	2	0	100.00%		31	67	0	0	N/A
21	48	1	1	0.00%		32	23	1	0	100.00%
21	65	7	0	100.00%		32	24	0	0	N/A
21	67	15	3	80.00%		32	25	0	0	N/A
22	10	2	0	100.00%		33	23	1	0	100.00%
22	11	47	0	100.00%		33	24	0	0	N/A
22	23	3	3	0.00%		33	25	0	0	N/A
22	24	1	1	0.00%		33	48	0	0	N/A
22	25	1	1	0.00%		33	67	0	0	N/A
22	38	2	0	100.00%		33	12b	1603	0	100.00%
22	48	1	1	0.00%		33	16b	195	0	100.00%
22	65	7	0	100.00%		34	20	0	0	N/A
22	67	15	3	80.00%		34	21	0	0	N/A
23	10	2	0	100.00%		34	16b	9361	0	100.00%
23	11	47	0	100.00%		49	10	13	0	100.00%
23	23	3	3	0.00%		49	11	143	0	100.00%
23	24	1	1	0.00%		49	24	105119	0	100.00%
23	25	1	1	0.00%		49	38	3	0	100.00%
23	38	2	0	100.00%		49	47	51	0	100.00%
23	48	1	1	0.00%		54	16b	4017	0	100.00%
23	65	7	0	100.00%		70	51	46621	0	100.00%
23	67	15	3	80.00%		70	61	3	0	100.00%
24	21	0	0	N/A		115	24	36	0	100.00%
24	23	1	0	100.00%		115	25	15	0	100.00%
24	24	0	0	N/A		121	24	35	0	100.00%
24	25	0	0	N/A		121	25	14	0	100.00%
25	21	0	0	N/A		121	67	10	0	100.00%
25	23	1	0	100.00%		123	24	36	0	100.00%
25	24	0	0	N/A		123	25	15	0	100.00%
25	25	0	0	N/A		123	48	6824	0	100.00%
26	21	0	0	N/A		126	24	73	0	100.00%
26	23	1	1	0.00%		126	25	31	0	100.00%

Continued on next page

Table 23 – continued from previous page

Cast Num	Step	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)		Cast Num	Steps	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
128	24	36	0	100.00%		469	12b	35	0	100.00%
128	25	15	0	100.00%		455	10	3	0	100.00%
130	24	36	0	100.00%		455	11	65	0	100.00%
130	25	15	0	100.00%		455	38	2	0	100.00%
141	12b	29	13	55.17%		455	47	19	0	100.00%
141	16b	79	33	58.23%		456	10	3	0	100.00%
142	12b	29	12	58.62%		456	11	17	0	100.00%
142	16b	79	32	59.49%		456	38	0	0	N/A
143	12b	29	12	58.62%		456	47	15	0	100.00%
143	16b	79	32	59.49%		457	47	15	0	100.00%
161	39	299	121	59.53%		461	11	131183	0	100.00%
162	39	299	120	59.87%		461	38	23039	0	100.00%
163	39	299	120	59.87%		461	47	49067	0	100.00%
224	12b	10967	0	100.00%		465	47	49031	0	100.00%
224	16b	12601	0	100.00%		468	12b	1549	0	100.00%
227	12b	5	0	100.00%		468	16b	31	0	100.00%
227	16b	7	0	100.00%		469	16b	117	0	100.00%
228	16b	7	0	100.00%		470	12b	11	0	100.00%
234	12b	0	0	N/A		470	16b	37	0	100.00%
234	16b	0	0	N/A		479	6	2295	0	100.00%
235	12b	0	0	N/A		479	10	2999	0	100.00%
235	16b	0	0	N/A		479	11	2519	0	100.00%
241	16	7	0	100.00%		479	65	60928	0	100.00%
241	39	11	0	100.00%		479	67	12287	0	100.00%
324	51	17	0	100.00%		511	27	11	0	100.00%
330	51	3717	0	100.00%		513	27	5	0	100.00%
346	16b	1467	0	100.00%		514	27	0	0	N/A
347	16b	1467	0	100.00%		515	27	0	0	N/A
349	10	39	0	100.00%		516	27	11	0	100.00%
351	16b	13	0	100.00%		522	16	7	0	100.00%
359	27	30	0	100.00%		522	39	11	0	100.00%
361	27	27	0	100.00%		522	16b	7	0	100.00%
403	8	5	0	100.00%		525	16	7	0	100.00%
403	51	7	0	100.00%		525	39	11	0	100.00%
410	8	13	0	100.00%		525	16b	7	0	100.00%
412	8	5295	0	100.00%						

Appendix C

OMNet++ Full Results

Here we present the full results of our analysis of the OMNET++ library and test programs, detailing all the dynamic down-cast locations within the code and their stability values in each test case.

C.1 Dynamic Down-Cast Locations

Table 24 displays the location of every dynamic down-cast found in the OMNET++ library and test cases.

Table 24: OMNet++ dynamic down-casts locations and featured programs

Cast Number	Source Location : Line Number	Featured in						
		fifo$_1$	**routing**	**dyna**	**fifo$_2$**	**aloha**	**cqn**	**histograms**
2	include/omnetpp/ccanvas.h:559					✓		✓
36	src/sim/cboolparimpl.cc:208		✓	✓		✓	✓	
37	src/sim/ccanvas.cc:1123					✓		
38	src/sim/ccanvas.cc:3960							✓
42	src/sim/ccomponent.cc:128							✓
43	src/sim/ccomponent.cc:248	✓	✓		✓	✓		✓
48	src/sim/ccomponenttype.cc:359	✓	✓	✓	✓	✓	✓	✓
49	src/sim/ccomponenttype.cc:365	✓	✓	✓	✓	✓	✓	✓
50	src/sim/ccomponenttype.cc:388	✓	✓	✓	✓	✓	✓	✓
51	src/sim/ccomponenttype.cc:414		✓	✓			✓	
53	src/sim/ccomponenttype.cc:94	✓	✓	✓	✓	✓	✓	
57	src/sim/cdoubleparimpl.cc:213		✓			✓	✓	
64	src/sim/cgate.cc:249					✓		
68	src/sim/cintparimpl.cc:210		✓				✓	
71	src/sim/cmessage.cc:339	✓	✓	✓	✓	✓	✓	✓
73	src/sim/cmodule.cc:437	✓	✓	✓	✓	✓	✓	✓
75	src/sim/cnedfunction.cc:242	✓	✓	✓	✓	✓	✓	✓
77	src/sim/cnedmathfunction.cc:130							✓
79	src/sim/cobjectfactory.cc:40	✓	✓	✓	✓	✓	✓	✓
90	src/sim/cpar.cc:127	✓	✓	✓	✓	✓	✓	✓
92	src/sim/cresultfilter.cc:284					✓		
93	src/sim/cresultlistener.cc:112					✓		
94	src/sim/cresultrecorder.cc:247	✓	✓		✓	✓		✓
95	src/sim/csimplemodule.cc:173			✓				
98	src/sim/csimulation.cc:568			✓				
101	src/sim/csoftowner.cc:78		✓	✓		✓		
102	src/sim/cstatistic.cc:90							✓
112	src/envir/envirbase.cc:1924	✓	✓	✓	✓	✓	✓	✓
113	src/envir/envirbase.cc:1933	✓	✓	✓	✓	✓	✓	✓
114	src/envir/envirbase.cc:199	✓	✓	✓	✓	✓	✓	✓
115	src/envir/envirbase.cc:354	✓	✓	✓	✓	✓	✓	✓
175	src/envir/eventlogfilemgr.cc:193	✓	✓	✓	✓	✓	✓	✓
192	src/common/expression.cc:174	✓	✓	✓	✓	✓	✓	✓
193	src/common/expression.cc:186	✓	✓	✓	✓	✓	✓	✓
194	src/common/expression.cc:233	✓	✓	✓	✓	✓	✓	✓
195	src/common/expression.cc:234	✓	✓	✓	✓	✓	✓	✓
196	src/common/expression.cc:235	✓	✓	✓	✓	✓	✓	✓
197	src/common/expression.cc:236	✓	✓	✓	✓	✓	✓	✓
198	src/common/expression.cc:237	✓	✓	✓	✓	✓	✓	✓
199	src/common/expression.cc:238	✓	✓	✓	✓	✓	✓	✓
212	src/envir/filesnapshotmgr.cc:43	✓	✓	✓	✓	✓	✓	✓
252	src/nedxml/nedresourcecache.cc:204	✓	✓	✓	✓	✓	✓	✓
253	src/nedxml/nedresourcecache.cc:491	✓	✓	✓	✓	✓	✓	
254	src/nedxml/nedresourcecache.cc:66	✓	✓	✓	✓	✓	✓	✓
255	src/sim/nedsupport.cc:100	✓	✓	✓	✓	✓	✓	✓
256	src/sim/nedsupport.cc:119		✓	✓		✓		
257	src/sim/nedsupport.cc:147		✓				✓	
259	src/sim/nedsupport.cc:279		✓	✓		✓		
260	src/sim/nedsupport.cc:422		✓	✓			✓	
266	src/envir/omnetppoutscalarmgr.cc:161		✓			✓		✓
267	src/envir/omnetppoutscalarmgr.cc:62	✓	✓	✓	✓	✓	✓	✓
268	src/envir/omnetppoutvectormgr.cc:64	✓	✓	✓	✓	✓	✓	✓
345	src/envir/resultfileutils.cc:74	✓	✓	✓		✓	✓	

C.2 Stability of Each Dynamic Down-Cast Site

Table 25 displays the stability of every cast site (Listed in Table 24) found in the OMNET++ library and test cases.

Table 25: Stability of every casts site within the OMNet++ tests

Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
2	aloha	2	0	100.00%
2	histograms	10	5	50.00%
36	aloha	37	0	100.00%
36	cqn	315	0	100.00%
36	dyna	33	0	100.00%
36	routing	51	0	100.00%
37	aloha	0	0	N/A
37	histograms	1	0	100.00%
38	histograms	0	0	N/A
42	histograms	3	3	0.00%
43	aloha	28	8	71.43%
43	fifo$_1$	9	9	0.00%
43	fifo$_2$	16	16	0.00%
43	histograms	3	3	0.00%
43	routing	278	273	1.80%
48	aloha	21	2	90.48%
48	cqn	156	6	96.15%
48	dyna	24050	4	99.98%
48	fifo$_1$	3	3	0.00%
48	fifo$_2$	4	3	25.00%
48	histograms	0	0	N/A
48	routing	29	15	48.28%
49	aloha	20	1	95.00%
49	cqn	152	5	96.71%
49	dyna	24049	3	99.99%
49	fifo$_1$	2	2	0.00%
49	fifo$_2$	3	2	33.33%
49	histograms	0	0	N/A
49	routing	23	14	39.13%
50	aloha	0	0	N/A
50	cqn	0	0	N/A
50	dyna	1	0	100.00%
50	fifo$_1$	0	0	N/A
50	fifo$_2$	0	0	N/A
50	histograms	0	0	N/A
50	routing	0	0	N/A
51	cqn	158	0	100.00%
51	dyna	17	0	100.00%
51	routing	13	0	100.00%
53	aloha	1	0	100.00%
53	cqn	165	7	95.76%
53	dyna	20	1	95.00%
53	fifo$_1$	2	0	100.00%
53	fifo$_2$	3	0	100.00%
53	routing	29	2	93.10%
57	aloha	414	0	100.00%
57	cqn	472	0	100.00%
57	routing	260	0	100.00%
64	aloha	1	0	100.00%
68	cqn	5	0	100.00%
68	routing	47	0	100.00%
71	aloha	16569886	6279546	62.10%
71	cqn	194091	3822	98.03%
71	dyna	926437	639782	30.94%
71	fifo$_1$	14384515	12111427	15.80%
71	fifo$_2$	1077243	657703	38.95%
71	histograms	1000000	0	100.00%
71	routing	24468	18435	24.66%
73	aloha	22	3	86.36%
73	cqn	152	5	96.71%
73	dyna	24050	4	99.98%
73	fifo$_1$	3	3	0.00%
73	fifo$_2$	4	3	25.00%
73	histograms	0	0	N/A
73	routing	29	19	34.48%
75	aloha	223	0	100.00%
75	cqn	43	0	100.00%
75	dyna	181	0	100.00%
75	fifo$_1$	43	0	100.00%
75	fifo$_2$	87	0	100.00%
75	histograms	261	2	99.23%
75	routing	209	0	100.00%
77	histograms	89	1	98.88%
79	aloha	61	0	100.00%
79	cqn	325	0	100.00%
79	dyna	24076	0	100.00%
79	fifo$_1$	21	0	100.00%
79	fifo$_2$	29	0	100.00%
79	histograms	13	0	100.00%
79	routing	330	0	100.00%
90	aloha	221	5	97.74%
90	cqn	796	15	98.12%
90	dyna	81	8	90.12%
90	fifo$_1$	3	3	0.00%

Continued on next page

Table 25 – continued from previous page

Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)	Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
90	$fifo_2$	4	3	25.00%	192	aloha	16	6	62.50%
90	histograms	7	0	100.00%	192	cqn	25	14	44.00%
90	routing	140	19	86.43%	192	dyna	17	6	64.71%
92	aloha	5	0	100.00%	192	$fifo_1$	1	1	0.00%
93	aloha	3139771	0	100.00%	192	$fifo_2$	1	0	100.00%
94	aloha	28	0	100.00%	192	histograms	4	2	50.00%
94	$fifo_1$	9	0	100.00%	192	routing	18	7	61.11%
94	$fifo_2$	16	0	100.00%	193	aloha	616	20	96.75%
94	histograms	3	0	100.00%	193	cqn	2795	64	97.71%
94	routing	278	0	100.00%	193	dyna	24821	22	99.91%
95	dyna	24048	1	100.00%	193	$fifo_1$	84	2	97.62%
98	dyna	24037	0	100.00%	193	$fifo_2$	102	4	96.08%
101	aloha	80	60	25.00%	193	histograms	113	12	89.38%
101	dyna	48120	2	100.00%	193	routing	1014	22	97.83%
101	routing	9	9	0.00%	194	aloha	616	20	96.75%
102	histograms	8	0	100.00%	194	cqn	2783	51	98.17%
112	aloha	0	0	N/A	194	dyna	24817	18	99.93%
112	cqn	0	0	N/A	194	$fifo_1$	84	2	97.62%
112	dyna	0	0	N/A	194	$fifo_2$	102	4	96.08%
112	$fifo_1$	0	0	N/A	194	histograms	106	4	96.23%
112	$fifo_2$	0	0	N/A	194	routing	1013	20	98.03%
112	histograms	0	0	N/A	195	aloha	10	3	70.00%
112	routing	0	0	N/A	195	cqn	37	19	48.65%
113	aloha	0	0	N/A	195	dyna	9	4	55.56%
113	cqn	0	0	N/A	195	$fifo_1$	0	0	N/A
113	dyna	0	0	N/A	195	$fifo_2$	1	0	100.00%
113	$fifo_1$	0	0	N/A	195	histograms	1	0	100.00%
113	$fifo_2$	0	0	N/A	195	routing	17	7	58.82%
113	histograms	0	0	N/A	196	aloha	10	3	70.00%
113	routing	0	0	N/A	196	cqn	37	19	48.65%
114	aloha	7	6	14.29%	196	dyna	9	4	55.56%
114	cqn	9	6	33.33%	196	$fifo_1$	0	0	N/A
114	dyna	7	6	14.29%	196	$fifo_2$	1	0	100.00%
114	$fifo_1$	7	6	14.29%	196	histograms	1	0	100.00%
114	$fifo_2$	7	6	14.29%	196	routing	17	7	58.82%
114	histograms	7	6	14.29%	197	aloha	10	3	70.00%
114	routing	7	6	14.29%	197	cqn	37	19	48.65%
115	aloha	0	0	N/A	197	dyna	9	4	55.56%
115	cqn	0	0	N/A	197	$fifo_1$	0	0	N/A
115	dyna	0	0	N/A	197	$fifo_2$	1	0	100.00%
115	$fifo_1$	0	0	N/A	197	histograms	1	0	100.00%
115	$fifo_2$	0	0	N/A	197	routing	17	7	58.82%
115	histograms	0	0	N/A	198	aloha	10	3	70.00%
115	routing	0	0	N/A	198	cqn	37	19	48.65%
175	aloha	0	0	N/A	198	dyna	9	4	55.56%
175	cqn	0	0	N/A	198	$fifo_1$	0	0	N/A
175	dyna	0	0	N/A	198	$fifo_2$	1	0	100.00%
175	$fifo_1$	0	0	N/A	198	histograms	1	0	100.00%
175	$fifo_2$	0	0	N/A	198	routing	17	7	58.82%
175	histograms	0	0	N/A	199	aloha	10	3	70.00%
175	routing	0	0	N/A	199	cqn	37	19	48.65%

Continued on next page

Table 25 – continued from previous page

Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)	Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
199	dyna	9	4	55.56%	255	histograms	4000003	0	100.00%
199	$fifo_1$	0	0	N/A	255	routing	6692	0	100.00%
199	$fifo_2$	1	0	100.00%	256	aloha	40	0	100.00%
199	histograms	1	0	100.00%	256	cqn	370	0	100.00%
199	routing	17	7	58.82%	256	dyna	4	0	100.00%
212	aloha	0	0	N/A	256	routing	34	0	100.00%
212	cqn	0	0	N/A	257	cqn	164	12	92.68%
212	dyna	0	0	N/A	257	routing	24	0	100.00%
212	$fifo_1$	0	0	N/A	259	aloha	40	0	100.00%
212	$fifo_2$	0	0	N/A	259	cqn	205	0	100.00%
212	histograms	0	0	N/A	259	dyna	4	0	100.00%
212	routing	0	0	N/A	259	routing	9	0	100.00%
252	aloha	4	0	100.00%	260	cqn	377	0	100.00%
252	cqn	6	0	100.00%	260	dyna	15	0	100.00%
252	dyna	6	0	100.00%	260	routing	83	0	100.00%
252	$fifo_1$	6	0	100.00%	266	aloha	1	0	100.00%
252	$fifo_2$	6	0	100.00%	266	histograms	11	3	72.73%
252	histograms	2	0	100.00%	266	routing	32	0	100.00%
252	routing	16	0	100.00%	267	aloha	0	0	N/A
253	aloha	1	0	100.00%	267	cqn	0	0	N/A
253	cqn	6	0	100.00%	267	dyna	0	0	N/A
253	dyna	2	0	100.00%	267	$fifo_1$	0	0	N/A
253	$fifo_1$	2	0	100.00%	267	$fifo_2$	0	0	N/A
253	$fifo_2$	3	0	100.00%	267	histograms	0	0	N/A
253	routing	29	0	100.00%	267	routing	0	0	N/A
254	aloha	0	0	N/A	268	aloha	0	0	N/A
254	cqn	0	0	N/A	268	cqn	0	0	N/A
254	dyna	0	0	N/A	268	dyna	0	0	N/A
254	$fifo_1$	0	0	N/A	268	$fifo_1$	0	0	N/A
254	$fifo_2$	0	0	N/A	268	$fifo_2$	0	0	N/A
254	histograms	0	0	N/A	268	histograms	0	0	N/A
254	routing	0	0	N/A	268	routing	0	0	N/A
255	aloha	5000060	0	100.00%	345	aloha	1	0	100.00%
255	cqn	96192	0	100.00%	345	cqn	1	0	100.00%
255	dyna	132325	0	100.00%	345	dyna	1	0	100.00%
255	$fifo_1$	3596208	0	100.00%	345	$fifo_1$	1	0	100.00%
255	$fifo_2$	538623	0	100.00%	345	routing	3	3	0.00%

Appendix D

Antlr4 Full Results

Here we present the full results of our analysis of the Antlr4 library and test programs, detailing all the dynamic down-cast locations within the code and their stability values in each test case.

D.1 Dynamic Down-Cast Locations

Table 26 displays the location of every dynamic down-cast found in the Antlr4 library and test cases.

Table 26: Antlr4 dynamic down-casts locations and input files. Key: cpp_1: avrc_api.cc, cpp_2: function_lib.cc, cpp_3: data_out_base.cc, html_1: antlr.html, html_2: gnu.html, html_3: github.html.

Cast Number	Source Location : Line Number	Featured in					
		cpp_1	ccp2	ccp3	html_1	html_2	html_3
1492	runtime/src/DefaultErrorStrategy.cpp:312	✓	✓				
1493	runtime/src/DefaultErrorStrategy.cpp:312	✓	✓				
1494	runtime/src/DefaultErrorStrategy.cpp:312	✓					
1502	runtime/src/Parser.cpp:353	✓	✓	✓	✓	✓	✓
1503	runtime/src/Parser.cpp:381	✓	✓	✓	✓	✓	✓
1521	runtime/src/atn/ATNDeserializer.cpp:148	✓	✓	✓	✓	✓	✓
1522	runtime/src/atn/ATNDeserializer.cpp:292	✓	✓	✓	✓	✓	✓
1523	runtime/src/atn/ATNDeserializer.cpp:295	✓	✓	✓	✓	✓	✓
1524	runtime/src/atn/ATNDeserializer.cpp:306	✓	✓	✓	✓	✓	✓
1525	runtime/src/atn/ATNDeserializer.cpp:315	✓	✓	✓	✓	✓	✓
1526	runtime/src/atn/ATNDeserializer.cpp:321	✓	✓	✓			
1527	runtime/src/atn/ATNDeserializer.cpp:332	✓	✓	✓	✓	✓	✓
1528	runtime/src/atn/ATNDeserializer.cpp:346	✓	✓	✓	✓	✓	✓
1529	runtime/src/atn/ATNDeserializer.cpp:358	✓	✓	✓	✓	✓	✓
1530	runtime/src/atn/ATNDeserializer.cpp:395	✓	✓	✓	✓	✓	✓
1531	runtime/src/atn/ATNDeserializer.cpp:410	✓	✓	✓	✓	✓	✓
1532	runtime/src/atn/ATNDeserializer.cpp:422	✓	✓	✓	✓	✓	✓
1533	runtime/src/atn/ATNDeserializer.cpp:426	✓	✓	✓	✓	✓	✓
1534	runtime/src/atn/ATNDeserializer.cpp:430	✓	✓	✓	✓	✓	✓
1535	runtime/src/atn/ATNDeserializer.cpp:434	✓	✓	✓	✓	✓	✓
1536	runtime/src/atn/ATNDeserializer.cpp:438	✓	✓	✓	✓	✓	✓
1537	runtime/src/atn/ATNDeserializer.cpp:451	✓	✓	✓	✓	✓	✓
1538	runtime/src/atn/ATNDeserializer.cpp:581	✓	✓	✓	✓	✓	✓
1539	runtime/src/atn/ATNDeserializer.cpp:585	✓	✓	✓	✓	✓	✓
1540	runtime/src/atn/ATNDeserializer.cpp:590	✓	✓	✓	✓	✓	✓
1541	runtime/src/atn/ATNDeserializer.cpp:606	✓	✓	✓	✓	✓	✓
1542	runtime/src/atn/ATNDeserializer.cpp:610	✓	✓	✓	✓	✓	✓
1543	runtime/src/atn/ATNDeserializer.cpp:614	✓	✓	✓	✓	✓	✓
1544	runtime/src/atn/ATNDeserializer.cpp:618	✓	✓	✓	✓	✓	✓
1545	runtime/src/atn/ATNDeserializer.cpp:622	✓	✓	✓	✓	✓	✓
1546	runtime/src/atn/ATNDeserializer.cpp:96	✓	✓	✓			
1547	runtime/src/atn/ArrayPredictionContext.cpp:77	✓	✓	✓	✓	✓	✓
1548	runtime/src/atn/LexerATNConfig.cpp:66	✓	✓	✓	✓	✓	✓
1549	runtime/src/atn/LexerATNSimulator.cpp:244	✓	✓	✓	✓	✓	✓
1550	runtime/src/atn/LexerATNSimulator.cpp:250	✓	✓	✓	✓	✓	✓
1551	runtime/src/atn/LexerATNSimulator.cpp:543	✓	✓	✓	✓	✓	✓
1559	runtime/src/atn/ParserATNSimulator.cpp:918	✓	✓	✓			
1560	runtime/src/atn/PredictionContext.cpp:147	✓	✓	✓	✓	✓	✓
1561	runtime/src/atn/PredictionContext.cpp:148	✓	✓	✓	✓	✓	✓
1562	runtime/src/atn/PredictionContext.cpp:200	✓	✓	✓	✓	✓	
1563	runtime/src/atn/PredictionContext.cpp:206	✓	✓	✓	✓	✓	✓
1564	runtime/src/atn/PredictionContext.cpp:91	✓	✓	✓	✓	✓	✓
1574	runtime/src/atn/SemanticContext.cpp:62	✓	✓				
1577	runtime/src/atn/SingletonPredictionContext.cpp:70	✓	✓	✓	✓	✓	✓
1579	runtime/src/dfa/DFA.cpp:27	✓	✓	✓	✓	✓	✓
1587	runtime/src/tree/Trees.cpp:101	✓	✓	✓	✓	✓	✓
1588	runtime/src/tree/Trees.cpp:101	✓	✓	✓	✓	✓	✓
1589	runtime/src/tree/Trees.cpp:101	✓	✓	✓	✓	✓	✓
1590	runtime/src/tree/Trees.cpp:103	✓	✓	✓	✓	✓	✓
1591	runtime/src/tree/Trees.cpp:111	✓	✓	✓	✓	✓	✓
1592	runtime/src/tree/Trees.cpp:111	✓	✓	✓	✓	✓	✓
1625	runtime/src/Recognizer.h:72	✓	✓	✓	✓	✓	✓
1628	runtime/src/BufferedTokenStream.cpp:98	✓	✓	✓	✓	✓	✓

D.2 Stability of Each Dynamic Down-Cast Site

Table 27 displays the stability of every cast site (listed in Table 26) found in the Antlr4 library and test cases.

Table 27: Stability of every casts site within the Antlr4 tests program. Key: cpp$_1$: avrc_api.cc, cpp$_2$: function_lib.cc, cpp$_3$: data_out_base.cc, html$_1$: antlr.html, html$_2$: gnu.html, html$_3$: github.html.

Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
1492	cpp$_2$	7	0	100.00%
1492	cpp$_3$	449	0	100.00%
1493	cpp$_2$	1	0	100.00%
1493	cpp$_3$	14	8	42.86%
1494	cpp$_3$	4	0	100.00%
1502	cpp$_1$	28522	28225	1.04%
1502	cpp$_2$	100850	100099	0.74%
1502	cpp$_3$	208484	206419	0.99%
1502	html$_1$	677	298	55.98%
1502	html$_2$	1298	544	58.09%
1502	html$_3$	3899	1540	60.50%
1503	cpp$_1$	28523	28308	0.75%
1503	cpp$_2$	100851	100699	0.15%
1503	cpp$_3$	208485	206876	0.77%
1503	html$_1$	678	445	34.37%
1503	html$_2$	1299	816	37.18%
1503	html$_3$	3900	2558	34.41%
1521	cpp$_1$	550	0	100.00%
1521	cpp$_2$	550	0	100.00%
1521	cpp$_3$	550	0	100.00%
1521	html$_1$	26	0	100.00%
1521	html$_2$	26	0	100.00%
1521	html$_3$	26	0	100.00%
1522	cpp$_1$	66	0	100.00%
1522	cpp$_2$	66	0	100.00%
1522	cpp$_3$	66	0	100.00%
1522	html$_1$	29	0	100.00%
1522	html$_2$	29	0	100.00%
1522	html$_3$	29	0	100.00%
1523	cpp$_1$	373	103	72.39%
1523	cpp$_2$	373	103	72.39%
1523	cpp$_3$	373	103	72.39%
1523	html$_1$	47	20	57.45%
1523	html$_2$	47	20	57.45%
1523	html$_3$	47	20	57.45%
1524	cpp$_1$	373	0	100.00%
1524	cpp$_2$	373	0	100.00%
1524	cpp$_3$	373	0	100.00%
1524	html$_1$	47	0	100.00%
1524	html$_2$	47	0	100.00%
1524	html$_3$	47	0	100.00%
1525	cpp$_1$	5	1	80.00%
1525	cpp$_2$	5	1	80.00%
1525	cpp$_3$	5	1	80.00%
1525	html$_1$	12	0	100.00%
1525	html$_2$	12	0	100.00%
1525	html$_3$	12	0	100.00%
1526	cpp$_1$	5	0	100.00%
1526	cpp$_2$	5	0	100.00%
1526	cpp$_3$	5	0	100.00%
1527	cpp$_1$	361	0	100.00%
1527	cpp$_2$	361	0	100.00%
1527	cpp$_3$	361	0	100.00%
1527	html$_1$	43	0	100.00%
1527	html$_2$	43	0	100.00%
1527	html$_3$	43	0	100.00%
1528	cpp$_1$	361	0	100.00%
1528	cpp$_2$	361	0	100.00%
1528	cpp$_3$	361	0	100.00%
1528	html$_1$	43	0	100.00%
1528	html$_2$	43	0	100.00%
1528	html$_3$	43	0	100.00%
1529	cpp$_1$	0	0	N/A
1529	cpp$_2$	0	0	N/A
1529	cpp$_3$	0	0	N/A
1529	html$_1$	4	0	100.00%
1529	html$_2$	4	0	100.00%
1529	html$_3$	4	0	100.00%
1530	cpp$_1$	550	0	100.00%
1530	cpp$_2$	550	0	100.00%
1530	cpp$_3$	550	0	100.00%
1530	html$_1$	26	0	100.00%
1530	html$_2$	26	0	100.00%
1530	html$_3$	26	0	100.00%
1531	cpp$_1$	373	103	72.39%
1531	cpp$_2$	373	103	72.39%
1531	cpp$_3$	373	103	72.39%
1531	html$_1$	47	20	57.45%

Continued on next page

Table 27 – continued from previous page

Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)	Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
1531	$html_2$	47	20	57.45%	1540	cpp_2	41	0	100.00%
1531	$html_3$	47	20	57.45%	1540	cpp_3	41	0	100.00%
1532	cpp_1	373	103	72.39%	1540	$html_1$	11	0	100.00%
1532	cpp_2	373	103	72.39%	1540	$html_2$	11	0	100.00%
1532	cpp_3	373	103	72.39%	1540	$html_3$	11	0	100.00%
1532	$html_1$	47	20	57.45%	1541	cpp_1	66	0	100.00%
1532	$html_2$	47	20	57.45%	1541	cpp_2	66	0	100.00%
1532	$html_3$	47	20	57.45%	1541	cpp_3	66	0	100.00%
1533	cpp_1	19	0	100.00%	1541	$html_1$	29	0	100.00%
1533	cpp_2	19	0	100.00%	1541	$html_2$	29	0	100.00%
1533	cpp_3	19	0	100.00%	1541	$html_3$	29	0	100.00%
1533	$html_1$	4	0	100.00%	1542	cpp_1	361	0	100.00%
1533	$html_2$	4	0	100.00%	1542	cpp_2	361	0	100.00%
1533	$html_3$	4	0	100.00%	1542	cpp_3	361	0	100.00%
1534	cpp_1	19	0	100.00%	1542	$html_1$	43	0	100.00%
1534	cpp_2	19	0	100.00%	1542	$html_2$	43	0	100.00%
1534	cpp_3	19	0	100.00%	1542	$html_3$	43	0	100.00%
1534	$html_1$	4	0	100.00%	1543	cpp_1	373	103	72.39%
1534	$html_2$	4	0	100.00%	1543	cpp_2	373	103	72.39%
1534	$html_3$	4	0	100.00%	1543	cpp_3	373	103	72.39%
1535	cpp_1	46	0	100.00%	1543	$html_1$	47	20	57.45%
1535	cpp_2	46	0	100.00%	1543	$html_2$	47	20	57.45%
1535	cpp_3	46	0	100.00%	1543	$html_3$	47	20	57.45%
1535	$html_1$	24	0	100.00%	1544	cpp_1	373	0	100.00%
1535	$html_2$	24	0	100.00%	1544	cpp_2	373	0	100.00%
1535	$html_3$	24	0	100.00%	1544	cpp_3	373	0	100.00%
1536	cpp_1	46	0	100.00%	1544	$html_1$	47	0	100.00%
1536	cpp_2	46	0	100.00%	1544	$html_2$	47	0	100.00%
1536	cpp_3	46	0	100.00%	1544	$html_3$	47	0	100.00%
1536	$html_1$	24	0	100.00%	1545	cpp_1	441	186	57.82%
1536	$html_2$	24	0	100.00%	1545	cpp_2	441	186	57.82%
1536	$html_3$	24	0	100.00%	1545	cpp_3	441	186	57.82%
1537	cpp_1	379	110	70.98%	1545	$html_1$	82	69	15.85%
1537	cpp_2	379	110	70.98%	1545	$html_2$	82	69	15.85%
1537	cpp_3	379	110	70.98%	1545	$html_3$	82	69	15.85%
1537	$html_1$	53	22	58.49%	1546	cpp_1	5	0	100.00%
1537	$html_2$	53	22	58.49%	1546	cpp_2	5	0	100.00%
1537	$html_3$	53	22	58.49%	1546	cpp_3	5	0	100.00%
1538	cpp_1	19	0	100.00%	1547	cpp_1	877783	0	100.00%
1538	cpp_2	19	0	100.00%	1547	cpp_2	8882925	0	100.00%
1538	cpp_3	19	0	100.00%	1547	cpp_3	14767104	0	100.00%
1538	$html_1$	4	0	100.00%	1547	$html_1$	269773	0	100.00%
1538	$html_2$	4	0	100.00%	1547	$html_2$	859563	0	100.00%
1538	$html_3$	4	0	100.00%	1547	$html_3$	6890381	0	100.00%
1539	cpp_1	46	0	100.00%	1548	cpp_1	3119	1880	39.72%
1539	cpp_2	46	0	100.00%	1548	cpp_2	2490	1595	35.94%
1539	cpp_3	46	0	100.00%	1548	cpp_3	5781	3573	38.19%
1539	$html_1$	24	0	100.00%	1548	$html_1$	793	738	6.94%
1539	$html_2$	24	0	100.00%	1548	$html_2$	1029	979	4.86%
1539	$html_3$	24	0	100.00%	1548	$html_3$	1004	954	4.98%
1540	cpp_1	41	0	100.00%	1549	cpp_1	1164	0	100.00%

Continued on next page

Table 27 – continued from previous page

Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
1549	cpp$_2$	953	0	100.00%
1549	cpp$_3$	1968	0	100.00%
1549	html$_1$	798	0	100.00%
1549	html$_2$	1060	0	100.00%
1549	html$_3$	791	0	100.00%
1550	cpp$_1$	1164	0	100.00%
1550	cpp$_2$	953	0	100.00%
1550	cpp$_3$	1968	0	100.00%
1550	html$_1$	798	0	100.00%
1550	html$_2$	1060	0	100.00%
1550	html$_3$	791	0	100.00%
1551	cpp$_1$	338	0	100.00%
1551	cpp$_2$	393	0	100.00%
1551	cpp$_3$	724	0	100.00%
1551	html$_1$	193	0	100.00%
1551	html$_2$	247	0	100.00%
1551	html$_3$	224	0	100.00%
1559	cpp$_1$	385	0	100.00%
1559	cpp$_2$	12671	0	100.00%
1559	cpp$_3$	32711	0	100.00%
1560	cpp$_1$	12836	12829	0.05%
1560	cpp$_2$	168860	168772	0.05%
1560	cpp$_3$	211075	210981	0.04%
1560	html$_1$	3321	3321	0.00%
1560	html$_2$	3397	3397	0.00%
1560	html$_3$	32011	32011	0.00%
1561	cpp$_1$	12836	0	100.00%
1561	cpp$_2$	168860	0	100.00%
1561	cpp$_3$	211075	0	100.00%
1561	html$_1$	3321	0	100.00%
1561	html$_2$	3397	0	100.00%
1561	html$_3$	32011	0	100.00%
1562	cpp$_2$	12442	0	100.00%
1562	cpp$_3$	5016	0	100.00%
1562	html$_1$	8003	0	100.00%
1562	html$_2$	41307	0	100.00%
1562	html$_3$	105141	0	100.00%
1563	cpp$_1$	535882	0	100.00%
1563	cpp$_2$	5304247	0	100.00%
1563	cpp$_3$	8915228	0	100.00%
1563	html$_1$	146206	0	100.00%
1563	html$_2$	451764	0	100.00%
1563	html$_3$	3535499	0	100.00%
1564	cpp$_1$	2973	0	100.00%
1564	cpp$_2$	17315	0	100.00%
1564	cpp$_3$	36676	0	100.00%
1564	html$_1$	2	0	100.00%
1564	html$_2$	2	0	100.00%
1564	html$_3$	2	0	100.00%
1574	cpp$_2$	11	0	100.00%
1574	cpp$_3$	11	0	100.00%
1577	cpp$_1$	6411091	0	100.00%
1577	cpp$_2$	42544562	0	100.00%
1577	cpp$_3$	79362682	0	100.00%
1577	html$_1$	2470218	0	100.00%
1577	html$_2$	8474368	0	100.00%
1577	html$_3$	61567880	0	100.00%
1579	cpp$_1$	46	0	100.00%
1579	cpp$_2$	46	0	100.00%
1579	cpp$_3$	46	0	100.00%
1579	html$_1$	24	0	100.00%
1579	html$_2$	24	0	100.00%
1579	html$_3$	24	0	100.00%
1587	cpp$_1$	30613	30101	1.67%
1587	cpp$_2$	109718	107317	2.19%
1587	cpp$_3$	225845	221904	1.75%
1587	html$_1$	678	549	19.03%
1587	html$_2$	1299	1028	20.86%
1587	html$_3$	3900	2886	26.00%
1588	cpp$_1$	35775	33198	7.20%
1588	cpp$_2$	130715	118280	9.51%
1588	cpp$_3$	268878	245781	8.59%
1588	html$_1$	2263	1229	45.69%
1588	html$_2$	4444	2355	47.01%
1588	html$_3$	14108	7168	49.19%
1589	cpp$_1$	5161	0	100.00%
1589	cpp$_2$	20996	3	99.99%
1589	cpp$_3$	43032	37	99.91%
1589	html$_1$	1584	0	100.00%
1589	html$_2$	3144	0	100.00%
1589	html$_3$	10207	0	100.00%
1590	cpp$_1$	30613	30101	1.67%
1590	cpp$_2$	109718	107317	2.19%
1590	cpp$_3$	225845	221904	1.75%
1590	html$_2$	1299	1028	20.86%
1590	html$_3$	3900	2886	26.00%
1591	cpp$_1$	5161	0	100.00%
1591	cpp$_2$	20994	0	100.00%
1591	cpp$_3$	42976	0	100.00%
1591	html$_2$	3144	0	100.00%
1591	html$_3$	10207	0	100.00%
1592	cpp$_1$	5161	0	100.00%
1592	cpp$_2$	20994	0	100.00%
1592	cpp$_3$	42976	0	100.00%
1592	html$_2$	3144	0	100.00%
1592	html$_3$	10207	0	100.00%
1625	cpp$_1$	93539	2	100.00%
1625	cpp$_2$	341759	2	100.00%
1625	cpp$_3$	728299	24	100.00%
1625	html$_2$	21339	2	99.99%
1625	html$_3$	72529	2	100.00%
1628	cpp$_1$	5181	0	100.00%

Continued on next page

Table 27 – continued from previous page

Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)		Cast Num	Program	Total Casts Performed -1 (P)	Total Source Type Changes (C)	Stability (S)
1628	cpp_2	21004	0	100.00%		1628	$html_2$	3485	0	100.00%
1628	cpp_3	43159	0	100.00%		1628	$html_3$	12149	0	100.00%