Systematic Testing of C++ Abstraction Recovery Systems by Iterative Compiler Model Refinement

Edward J. Schwartz
Carnegie Mellon University
edmcman@cmu.edu

Cory F. Cohen
Carnegie Mellon University
cfc@cert.org

Stephanie Schwartz
Millersville University
stephanie.schwartz@millersville.edu

Abstract

C++ source code abstractions such as classes and methods greatly assist human analysts and automated algorithms alike when analyzing C++ programs. These abstractions are lost during the compilation process, but researchers have been developing tools to recover them using program analysis. Despite promising advances, this difficult problem remains unsolved, with state-of-the-art solutions self-reporting accuracies of 78% [32] and 77.5% [10] for different types of abstractions.

In this paper, we address this problem by proposing a new model-based approach for systematically testing C++ abstraction recovery systems. Our high-level approach is to both jointly and iteratively refine the abstraction recovery system *and* a compiler model that introspects the compilation process. We built EmCee, a model of Microsoft's Visual C++ compiler, to apply our technique to the popular C++ abstraction recovery systems VirtAnalyzer [10] and OOAnalyzer [32]. EmCee "parses" input files by interpreting them as answers to a series of multiple choice questions (inspired by the game "twenty questions"), which makes it very amenable to fuzzing. We use an off-the-shelf grey-box fuzzer to automatically generate test cases for EmCee that represent a variety of program structures and optimizations. We then use these test cases to evaluate the reasoning in VirtAnalyzer and OOAnalyzer for soundness problems, and correct any violations. Using our approach, we identified 27 soundness problems in OOAnalyzer and three in VirtAnalyzer.

ACM Reference Format:
Edward J. Schwartz, Cory F. Cohen, and Stephanie Schwartz. 2025. Systematic Testing of C++ Abstraction Recovery Systems by Iterative Compiler Model Refinement. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering (SURE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3733822.3764669

1 Introduction

As a language of choice for complex systems that must be performant such as web browsers, database engines, productivity suites, and operating systems, the security of C++ programs has become an important topic. The proliferation of C++ has spurred specialized research in many areas of security, including decompilation [13, 14], reverse-engineering [29], vulnerability detection [7], and runtime security protections [1, 9, 23, 24, 37, 38]. Many of these applications require knowledge of the program's *object-oriented* (OO) abstractions

ACM ISBN 979-8-4007-1910-3/25/10
https://doi.org/10.1145/3733822.3764669

```
mov [ecx], vftable
ret 0
```

(a) Compiled code

```
struct D {
  virtual ~D() {}
};
```

(b) Likely source code

Figure 1: Method `D::D`

from the source code. For example, early runtime protection systems for C++ programs [1, 37, 38] used the class hierarchy information in source code to determine whether a method is allowed to call a method on a different class.

Unfortunately, in many security scenarios, the software we need to analyze or protect does not include source code. This is often the case when working with malware and commercial off-the-shelf (COTS) software. Because source code was traditionally the only method we had to obtain OO abstractions, this often meant that otherwise valuable security research could not be applied in practice. Over time, researchers have begun to address this problem by analyzing programs at the executable level to recover the OO abstractions they need, rather than assuming they can obtain them from source code [7–10, 13, 14, 19, 20, 23, 33, 36]. For example, several newer C++ protection schemes only require access to a program's executable to function [9, 23, 24].

The recovery of OO abstractions from executables has been studied for well over a decade. Early work typically focused on extracting information from a single source of information: virtual function tables (vftables), which are used to implement dynamic dispatch (i.e., virtual functions) in C++ [7–9, 13, 14, 20, 23, 36]. More recent approaches to OO abstraction recovery also analyze constructor code, destructor code, or examine the paths in which object pointers are propagated throughout the program [10, 32]. Despite much promising work, the core problem remains unsolved, with the state-of-the-art solutions self-reporting accuracies of 78% [32] and 77.5% [10] for different types of abstractions.

In our experience, it's (relatively) easy to formulate an inference rule that can leverage observations of executable code to correctly make conclusions about the original C++ source code on 80% of programs. But soundly reasoning about the corner cases is *hard.* As one pathological example, we found an example program in which a derived class is *smaller* than its base class, which we strongly believed was impossible (Section 6.1)!

Let's examine another example that seems straight-forward, but presents some unexpected complexity. One of the most important parts of OO analysis is determining the class that each vftable belongs to by looking at the compiler-generated code in constructors and destructors that installs them into objects. Fortunately, the structure of this automatically generated code is well-known [16], and in many cases this is straight-forward.

```
struct B {};
struct D : B {
  virtual ~D() {}
};
```

(a) Source program a

```
struct B {
  virtual ~B () {}
};
struct D : B {};
```

(b) Source program b

Figure 2: Two C++ programs that are indistinguishable at the executable level but install vftables for different classes. (a) installs the derived class's vftable; (b) installs the base class's vftable. This example demonstrates the difficulty that inlining and vftable elimination can pose.

Imagine that we are analyzing the assembly code for method `D::D` as shown in Figure 1a. From a cursory analysis, an analyst can see that the method is installing a vftable to the `this` object (in %ecx) at offset 0. Because there are no signs of another class, most analysts would conclude that `D` has no base classes, as in Figure 1b, and as a result, that the vftable must belong to `D`. And in many cases, those analysts would be correct. Unfortunately, over time we have learned that multiple compiler optimizations (Section 5.1) can interact and produce very misleading assembly code. For example, Figure 2 shows two programs that also compile to the assembly code in Figure 1a, but in which class `D` inherits from a base class. Although the program in Figure 2a will install `D`'s vftable, as most analysts would expect, the program in Figure 2b actually installs `B`'s vftable, even though there is no evidence of `B`'s existence in `D`'s constructor. This is surprising because constructors usually install the vftables of their base class, which is a clue that there is an inheritance relationship. Unfortunately, as Figure 2 shows, this clue is not always present.

In this paper, we attempt to address such problems by proposing a method to systematically test systems that recover C++ abstractions from executables, such as VirtAnalyzer [10] and OOAnalyzer [32]. Our high-level approach is to jointly and iteratively refine both an OO analysis system *and* an abstract compiler model in tandem. The compiler model introspects the compilation process and detects mistakes made by the analysis system, while the analysis system identifies gaps in the compiler model. To apply our approach to OOAnalyzer and VirtAnalyzer, we built EmCee, a model of Microsoft's Visual C++ compiler. In each iteration of our process, we use an automatic test case generator—a grey-box fuzzer [12, 22]—to produce test cases for EmCee. Each test case is an answer to multiple choice questions (see "twenty questions" below or Section 4.2) from which EmCee generates a source-level program and the low-level executable it compiles to. Because EmCee has perfect knowledge about both the source and executable representations, it can run the system under test on the executable representation and detect any incorrect conclusions made about the source program, which constitute potential soundness violations. An analyst examines each violation and determines if the problem can be reproduced in the real compiler. If it can, this is proof of a soundness problem, and the analyst refines the OO recovery system to fix the problem. (If not, it suggests the analyst must refine the model compiler.)

At a high level, we build a compiler model (rather than use a real compiler) to expose internal details of the compilation process that are not revealed by regular compilers (e.g., MSVC), even with debug flags. These internal details are important because they provide ground truth for intermediate observations made by the system

```
Enter the general maximum size (static max is 5): 1
Create a class? 0: false 1: true 1
Add a method? 0: false 1: true 1
Type of method? 0: Normal 1: Constructor 2: Destructor 1
Add local variable? 0: false 1: true 0
Add class member? 0: false 1: true 1
Type? 0: AbstractType, 1: PtrType, 2: DeletablePtrType,
    ↪ 3: ClassType 3
Cannot create a ClassType because there are no classes.
Selected AbstractType. Creating initializer expression.
    ↪ 0: Rvalue 1: Lvalue 0
0: Literal 1: Nullptr 2: New 3: This 4: Call 5: AddrOf 6:
    ↪  Deref 7: ObjPtrCast 0
```

Figure 3: An example session of executing EmCee. Input appears like this. The model compiler's output is only present during interactive (i.e., human) use.

under test, such as whether a vftable installation was inlined into a caller method. Obtaining this ground truth allows EmCee to determine when the system under test concludes an intermediate fact that is not true. This in turn allows EmCee to test each rule in the system under test in isolation for soundness (Section 3).

Similar to a real compiler, EmCee transforms a high-level C++ program into a low-level intermediate representation (IR) that is amenable to analysis and optimizations. However, that is where the similarities to real compilers end. EmCee purposefully abstracts away details that are not relevant to OO recovery, such as control flow, templating, and the specifics of the C++ syntax. EmCee "parses" its input as answers to a series of multiple choice questions, which is inspired by the game "twenty questions". The answers to these questions indirectly describe the source-code structure of the input program (see Figure 3 for an example). EmCee then outputs a set of *facts* or observations about the input program's structure and behavior at both the source and executable levels (see Figure 4 for an example). In fact, one way to think about EmCee is that it compiles programs *to facts* rather than machine code. Collectively, these differences make EmCee very amenable to automatic test case generation, ensuring that advanced but rare OO structures, such as virtual inheritance, multiple inheritance, and empty base classes, are represented in the tests. For example, using EmCee, test case generation easily identified the programs in Figures 1b and 2 as all compiling to the code in Figure 1a.

We applied our iterative refinement process on two state-of-the-art OO abstraction recovery systems, OOAnalyzer [32] and VirtAnalyzer [10]. For OOAnalyzer, an analyst created 39 patches until the process no longer identified any problems. 27 of these patches corrected a soundness violation, nine patches modified rules to apply more broadly, and the remainder were small refinements or scaffolding code. In total, the process identified and corrected at least one soundness violation in 24% of OOAnalyzer's reasoning rules (19 of 80). It also fixed two rules which would never conclude any facts. When applying the process to VirtAnalyzer, which is notably simpler, an analyst created three patches to address soundness violations, and one patch to allow VirtAnalyzer to make conclusions more broadly.

Contributions In this paper, we present a new model-based technique for systematically testing C++ abstraction recovery systems.

```
% function(1) returns the object pointer that was passed as an argument
returnsSelf(function(1)).
% function(0) receives the object pointer thisptr(8) in ecx
funcParameter(function(0), ecx, thisptr(8)).
% the call at 0x11003 in function(1) passes thisptr(25) in ecx
callParameter(0x11003, function(1), ecx, thisptr(25)).
% the call at 0x11003 in function(1) targets function(9)
callTarget(0x11003, function(1), function(9)).
% thisptr(25) points to 24 bytes allocated on the heap by 0x11001 in function(1)
thisPtrAllocation(0x11001, function(1), thisptr(25), type_Heap, 24).
% thisptr(35) points 8 bytes beyond thisptr(14) in function(4)
thisPtrOffset(function(4), thisptr(14), 0x8, thisptr(35)).
```

(a) Initial (binary analysis) facts

```
% function(4) is an OO method
factMethod(function(4)).
% function(4) is defined on class(0)
findint(function(4), class(0)).
% Methods in class(0) are able to call method function(4)
factClassCallsMethod(class(0), function(4)).
% function(12) is a real destructor
factRealDestructor(function(12)).
% entry 0 of vftable(0) points to function(4)
factVFTableEntry(vftable(0), 0, function(4)).
% class(1) inherits virtually from class(0) at offset 8
factDerivedClass(class(1), class(0), 0x8, virtual).
```

(b) Entity (ground-truth) facts

Figure 4: Example OOAnalyzer facts emitted by EmCee

```
struct B {
 int b_mem;
 B() { b_mem = 2; }
 virtual void b_vf();
};
struct VB {
 char vb_mem;
 VB(){ vb_mem = '3'; }
};
```

```
struct D : B, virtual VB {
  B b_embedded;
  virtual void b_vf() override;
  virtual void d_vf();
};
```

Figure 5: Working example program.

```
class D size(21):
  +---
   | +--- (base B)
 0|  | {vfptr}
 4|  | b_mem
   | +---
 8| {vbptr}
12| B b_embedded
  +---
  +--- (vbase VB)
20| vb_mem
  +---
```

Figure 6: Data layout of D from Figure 5

We create EmCee, a model compiler that closely mimics the behavior of Microsoft's Visual C++ compiler on 32-bit x86 executables, which we publish in support of open science.[1] Finally, we demonstrate that our technique is effective in practice by discovering 27 soundness violations in OOAnalyzer and three in VirtAnalyzer.

2 Background

In this section, we will introduce the necessary background on inference rules and how C++ programs are compiled.

2.1 Inference rules

Modern OO analysis tools such as OOAnalyzer reason about programs *iteratively*, by making a *series* of smaller conclusions. Each step of the reasoning sequence begins at a *knowledge state*, σ, which is the set of all conclusions about the program at a point in time. A tool may make a reasoning step by applying an internal rule to the current state σ to form a new knowledge state σ', which contains new facts from the rule. We denote this as $\sigma \overset{rule}{\rightsquigarrow} \sigma'$. The new state σ' then becomes the new current state, and the process is repeated until no new conclusions can be made. We denote the sequence of zero or more reasoning steps as $\sigma \rightsquigarrow^* \sigma'$.

Each rule is actually an *inference rule*:

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_n}{C}$$

where P_i represents the ith premise of the rule, and C represents the conclusion. In essence, the rule asserts that if all the premises are true, then the conclusion must be true as well. For example, this is an example rule from OOAnalyzer that states *if a method is directly callable by a base class, it cannot be defined on the derived class*:

$$\frac{\text{ClassCallsMethod}(Cl_d, M) \qquad \text{ClassCallsMethod}(Cl_b, M) \\ \text{DerivedClass}(Cl_d, Cl_b, \_) \qquad M \in Cl_m}{Cl_m \neq Cl_d}$$

OOAnalyzer implements each inference rule as a clause in Prolog, a declarative logic programming language. The meaning of each type of fact, such as ClassCallsMethod and DerivedClass are defined in the OOAnalyzer paper [32].

2.2 Data layout

We assume that the reader has a working knowledge of C++, including features such as classes, inheritance, and virtual functions. Understanding how C++ programs are compiled to executables is required to understand the nature of OO analysis rules, and why it can be difficult to know if they are correct. We focus here on Visual C++ and its ABI (Application Binary Interface), but many of the details are similar for other compilers and ABIs. For more details about Visual C++ compilation, we recommend reading Jan Gray's excellent description of the topic [16]. We will use the program in Figure 5 as our working example, which is inspired by Gray [16]. Throughout this paper, we will include relevant links as footnotes to Compiler Explorer [15] sessions, such as this link for the working example program[2]. Compiler Explorer is a web service that compiles a source code program specified by the user and displays the corresponding assembly code.

We will now explain how class objects are laid out in memory using a class layout visualization[3] of class D, which is shown in

[1]Available at https://github.com/sei-eschwartz/emcee

[2]https://godbolt.org/z/K5qv49Tj5

[3]Visual C++ will emit these diagrams when passed the undocumented flag /d1reportAllClassLayout.

```
b_vf: &B::b_vf
```

(a) `B`'s vftable

```
b_vf: &D::b_vf
d_vf: &D::d_vf
```

(b) `D`'s vftable

Figure 7: Vftables for B and D

Figure 6. Such visualizations are extremely helpful for thinking about the validity of OO inference rules, and we will utilize them to quickly summarize examples and counter-examples throughout the paper. Object offsets are one of the fundamental observations made at the executable level, and these diagrams concisely represent offsets and connect them to the higher-level class structure.

Class layout begins with the special `vfptr` member, if it is needed. As we will discuss in Sections 2.3 and 2.4, the compiler uses special `vfptr` and `vbptr` fields to implement virtual functions and virtual inheritance respectively. If a class needs its own `vfptr` member, it will be located at offset `0`, such as in class `B`. When possible, Visual C++ will reuse a member inherited from a base class. For example, `D` reuses the `vfptr` from `B`. Following the `vfptr` field (if it is needed), Visual C++ then lays out each base class that has been inherited non-virtually; in `D`'s case, that is `B` at offset `0`. Classes are laid out in source-code order, *except* that if a base class's `vfptr` field is reused, that class is always laid out first.[4] After this, the compiler includes a `vbptr` member, if needed, such as in `D`, and then the user-defined class members, such as `b_embedded`.

Virtual bases are included last. This includes direct virtual bases, which are listed in the source code declaration, but also any classes that are virtually inherited by an ancestor. This is a major difference between virtual and non-virtual inheritance.

2.3 Virtual functions

Visual C++'s implementation of two features, virtual functions and virtual inheritance, provide some of the most important information in OO analysis. Virtual functions are a form of dynamic dispatch that selects the implementation of a method based on the type of object the method is invoked on, rather than the type of pointer or reference it is invoked through. Like most compilers, Visual C++ implements virtual functions using compiler-generated *virtual function tables* (vftables).

At a high level, a vftable maps each virtual function on that class to the actual implementation. When that class is inherited by another class, the derived class is responsible for installing its own vftables, which may override the implementations. For example, Figure 7 shows the vftables for classes `B` and `D`, and `D`'s vftable overrides `B`'s implementation of `b_vf`. From a reverse-engineering perspective, a virtual function table belonging to class `C` is a list of methods that are guaranteed to be on `C` or one of its ancestors. Because of this, vftables were the main source for many early OO reverse-engineering systems [7–9, 13, 14, 20, 23, 36].

When a derived class overrides a virtual function in a base, Visual C++ constructs the derived method with a *thisptr adjustment*, which is the offset from the base object to the derived object. A thisptr adjustment A on `C::M` indicates that method `M` expects to be called with a thisptr pointing A bytes past the start of a `C` object. This means that if `M` accesses offset O of its thisptr, it is actually accessing offset $O + A$ of a `C` object. This can be problematic because the method is defined on the derived class, and so can access derived members and methods, but when the method is called it looks like it is on the base class. This can result in strange-looking executable code, such as a method that appears to access members at negative offsets.

[4]Small details like this can be important; see Section 6.6.

2.4 Virtual inheritance

Virtual inheritance is a solution to the so-called "diamond problem" in OO programming, in which a class inherits multiple copies of an ancestor through different inheritance paths. Virtual inheritance is a form of inheritance that indicates a class is willing to share a copy of a base class in derived classes. In the example program, class `D` virtually inherits from class `VB`, indicating it is willing to share a copy of `VB`. Due to space restrictions, the example program does not contain any sharing of `VB`. But if a class inherited from `D` and virtually inherited from `VB`, the same instance of `VB` would be shared.

The primary implementation challenge of virtual inheritance is that the offset to virtual bases is not statically known, since it may need to share another class's copy. Visual C++ handles this by installing *virtual base tables* (vbtables) which contain the offset from the current class's `vbptr` to each virtual base. From a reverse engineering perspective, vbtables reveal important information about the virtual inheritance hierarchy [10].

2.5 Constructors and destructors

Constructors and destructors are compiled into executable functions in a manner similar to that of normal methods. Unlike normal methods, constructors and destructors perform a range of automatic operations, such as installing vftables, vbtables, initializing base classes, and so on. The Visual C++ ABI also dictates several types of special helper functions for destructors. For this reason, the "main" destructor at the executable level is called the real destructor. There is also a deleting destructor, which calls the real destructor before invoking `delete`. Finally, a special vbase destructor is created to properly destruct virtual bases. This level of detail may seem unnecessary, but the details matter: EmCee found a subtle flaw in which a vbase destructor causes unsoundness in an OOAnalyzer rule (Section 6.5). While destructors can provide important observations about the source program, they also pose challenges to reasoning accurately.

3 Models of Correctness

In this section, we discuss what it means for a tool such as OOAnalyzer or VirtAnalyzer to be correct. From a user's point of view, correctness is straightforward. First, we assume that a developer compiled the executable β from a C++ source code program α^{dev} with a C++ compiler, using some optimization settings γ^{dev}. We denote this as $\texttt{compilesto}(\alpha^{dev}, \gamma^{dev}, \beta)$. The user runs the tool on executable β, and the tool is correct if, for *some* C++ source code program α and optimizations γ such that $\texttt{compilesto}(\alpha, \gamma, \beta)$, (1) all of the facts output by the tool are true with respect to α, β, and γ (this is *soundness*) and (2) the tool reports all relevant facts about α, β, and γ (this is *completeness* or 100% recall).

It may seem at first glance like a better definition of correctness would be to return an answer equivalent to the original source file α^{dev}, but there are two problems with this. First, as we showed in Figures 1 and 2, multiple source programs can compile to exactly the

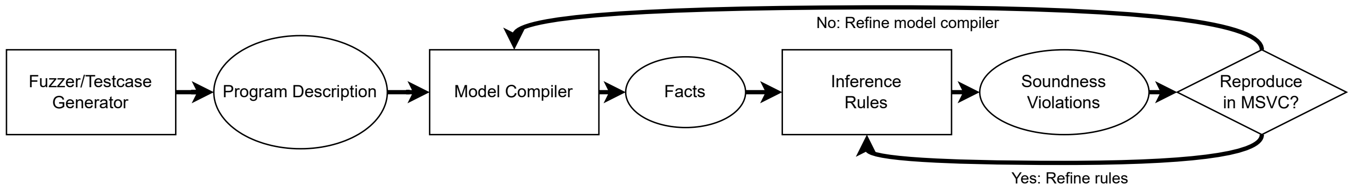

Figure 8: The iterative testing and refinement process

same binary, making perfect recovery impossible. Second, in almost all cases, the user does not know the original source program α^{dev}, so pragmatically it does not matter to her which source program is described as long as it reflects the behavior seen in the executable β.

3.1 Local soundness under the perfect knowledge model

Unfortunately, establishing global correctness is very hard. Since OOAnalyzer and VirtAnalyzer build conclusions by chaining together applications of reasoning rules (Section 2.1), in this paper we pursue a more tractable goal: validating each inference rule in isolation. By ensuring that each step preserves soundness, we move toward the larger goal of system-wide correctness. Specifically, we establish a weaker form of correctness that we call *local soundness under the perfect knowledge model*.

The perfect knowledge model simply means that we only consider reasoning from a correct (sound and complete) knowledge state. A rule is locally sound under the perfect knowledge model iff, for all α, β, and γ such that `compilesto`(α, γ, β), any conclusion made by the rule from any correct knowledge state σ is true (with respect to α, β, and γ). Since the starting state σ was correct and thus sound by assumption, and any new conclusion is true, then any state σ' such that $\sigma \overset{rule}{\rightsquigarrow} \sigma'$ must be sound as well by definition. In essence, this weaker form of correctness demonstrates that the rule preserves soundness under ideal circumstances (i.e., perfect knowledge).

We focus on the perfect knowledge model in this paper because it allows us to reason about each rule in isolation. OOAnalyzer in particular contains dozens of different rules that interact in extremely complex ways. This makes it challenging to predict which conclusions could plausibly be present in a knowledge state σ when a rule is evaluated. The perfect knowledge model allows us to side-step this question and the complications that arise from these interactions. In practice, of course, OOAnalyzer and VirtAnalyzer do not reason from a correct knowledge state. Indeed, their iterative nature implies that they often reason from *incomplete* knowledge states. Despite this, we have found testing under the perfect knowledge model to be extremely effective, as we demonstrate in the remainder of the paper.

4 Iterative testing and refinement process overview

The cornerstone of our approach is an iterative process that uses the compiler model to identify and correct defects in OO reasoning rules, and as a byproduct, uses the OO rules to identify and correct defects in the compiler model. This process is based on the following two assumptions about OO rules. First, when an analyst writes a rule, she expects that there is a program that can trigger it. Analysts do not (intentionally) waste their time creating rules that are useless, though it can happen accidentally. Second, as we discuss in Section 2.1, if the premises to a rule are satisfied by a program, then the conclusion of the rule should be true for that program. We call these the *coverage* and *soundness criteria*, respectively. The coverage criteria ensures that the testing process triggers each rule at least once, and as a consequence that the testing process is sufficient to model many different behaviors and functionality of Visual C++. Since we expect all rules to be triggered, failing to trigger a rule is a sign that something went wrong. The soundness criteria simply ensures that only true conclusions are made.

The process is depicted in Figure 8, and starts with automatically generating test cases for the model compiler. Each test case consists of answers to a series of multiple choice questions "asked" by the model compiler. Figure 3 shows an example of such questions and answers. The model compiler uses the answers in each test case to instantiate a model of a source code program and its compiled executable. Each analysis rule is then tested in isolation to ensure that no soundness violations are detected for the program described in that test case. If a soundness violation occurs, the analyst manually replicates the problem in the real Visual C++ compiler. If the problem can be replicated, then the rule is unsound and the analyst refines the rule to correct the problem. If the problem cannot be replicated, then there is a fidelity problem in the compiler model, and the analyst refines the compiler model in order to address the problem. The process repeats until the coverage and soundness criteria are met.

In the following sections, we describe in greater detail how we applied the process to our model compiler EmCee, and the OO recovery systems OOAnalyzer and VirtAnalyzer.

4.1 Test case generation

Each iteration starts by running automatic test case generation tools on the most recent revision of the model compiler. In this paper, we used the AFL++ fuzzer (version 4.02c) [12] as our test case generator. AFL++ is a grey-box mutational fuzzer [22]. Mutational fuzzers perform mutations (e.g., bit flipping) on a set of seed inputs and run the program on these mutated inputs. A grey-box fuzzer uses code coverage to detect when a mutated input reaches a new program behavior, and adds that input to the set of seed inputs that are mutated. In this way grey-box fuzzers like AFL++ can learn to automatically trigger different code behaviors. In EmCee, these different behaviors roughly correspond to different source-code program structures. One of the strengths of this approach is that the fuzzer is not bound by coding conventions or limited by imagination and therefore generates some truly bizarre and tricky test cases. Because we designed EmCee's input format to be very simple (Section 4.2), AFL++ was able to quickly establish a large corpus of high-coverage test cases for the first iteration even though we gave it no seed inputs to start with. For later iterations, we used the test cases from the previous iteration as seed inputs to speed up the testing process.

4.2 Model Compiler: Front-end

A compiler's front-end is responsible for reading an input specification of a program, and converting it to an internal representation of the program's structure. In a real C++ compiler, this input specification is determined by the C++ language. But we have no such requirement with EmCee, and there is good reason *not* to do so. C++ syntax is complicated, so many inputs are *malformed* and will be rejected by the parser. Since these malformed inputs are not compiled, they do not help us test OO abstraction recovery systems. Instead of using C++ syntax, our front-end is designed around a simple principle: *as long as the input is sufficiently long, always produce a high-level program representation.* To accomplish this, our compiler's front-end plays a game inspired by twenty questions.

In a game of *twenty questions*, one person thinks of an object, and the other person attempts to guess the object by asking twenty yes or no questions. (It is basically a gamified version of binary search.) In our case, the user (i.e., input to the compiler) chooses an abstracted C++ source code program, and EmCee asks multiple-choice questions about the program until it has enough information to instantiate a complete representation of the corresponding program. Figure 3 in the introduction displays an example session of this process. Note that the grey-box fuzzer we employ in this paper for test case generation does not *read* the questions; it instead mutates the answers randomly and uses code coverage to detect when a test-case does something interesting and is worth keeping (Section 4.1). Even though it is unaware of the questions it is "answering", it still benefits from the very simple input structure the questions impose.

4.3 Model Compiler: Back-end

The back-end of a compiler is responsible for translating the source-level program representation to an "executable level" representation. Unlike a standard compiler, EmCee's primary output is not assembly code, but rather a set of *facts* that describe the properties of the program at both the source and executable levels. Rather than reinventing the wheel, we chose to express these facts using the same fact language as OOAnalyzer. We chose OOAnalyzer's fact language because it was the first system we tested, but we later found that it subsumed the types of observations that VirtAnalyzer makes too.

At a high level, OOAnalyzer contains two types of facts: initial facts and entity facts. Figure 4 in the introduction depicts examples of both. *Initial facts* (Figure 4a) describe properties that can be observed from the compiled executable using binary analysis. For example, these include facts about data-flow, calling conventions, and control flow. *Entity facts* (Figure 4b), in contrast, largely describe properties of the original source code abstractions, such as class membership, inheritance, and so on. They also include facts about compiler-generated structures, such as vftables and whether a method is a constructor.

EmCee was designed from the ground up to be able to emit these facts. Similar to a conventional compiler, EmCee first translates the source program to a compiler intermediate representation (IR). EmCee differs from a typical compiler in that many operations in its IR directly correspond to OO facts that it can emit. Although a conventional compiler may have similar operations, EmCee's operations carry metadata to be able to correlate information about executable-level objects with their corresponding source-level objects, even after optimizations. After the input program is translated to IR form, the model compiler then (optionally, depending on the input) optimizes the IR (Section 5.1). The final step of EmCee's back-end is a lightweight analysis that computes the set of facts describing both the final optimized low-level code and the corresponding source-code program.

EmCee creates a Prolog test harness for each test case. The test harness loads the tools' rules in Prolog form, and contains all the facts produced from the model compiler, which are expressed as Prolog facts. For each rule, the harness contains two *check* predicates: one for the soundness criteria, and the other for the coverage criteria. The soundness predicate enumerates over every conclusion made by a rule, and checks whether that conclusion is true (sound) according to EmCee. The coverage predicate outputs the information needed to compute recall: it enumerates over every observation in the ground truth, and determines whether that observation is concluded by the rule. For example, if a program has 10 constructors, and rule ConstructorB identifies 3 of them, ConstructorB has a recall of 3/10 for this program. The coverage criteria requires that each rule has a recall greater than zero when computed over all test cases.

Since OOAnalyzer is written in Prolog, this all works out nicely. VirtAnalyzer, on the other hand, is written in IDAPython. As we describe in Section 7, we manually extracted the abstract rules that were implemented in VirtAnalyzer's code and expressed them in Prolog.

4.4 Manual analysis

If there were any soundness errors, we picked one of the failing test cases to analyze. Because the test cases were often quite large and complex, we used the creduce [26] and lithium [28] file minimizers to remove program structure unrelated to the bug in question. We then manually examined the minimized test case using EmCee's introspection capabilities. The model compiler is able to emit the source-code program in C++, and can output the IR before, during, and after it is optimized. If the test case appeared to be a valid counter-example of a rule, we attempted to reproduce the problem in Visual C++. If we could reproduce the problem, this proved the rule was unsound. Consequently, we would refine the rule to fix the identified problem, and start the next iteration of the refinement process. If we could not reproduce the behavior in Visual C++, then EmCee's model did not match the behavior of the real compiler. In this case, we would fix EmCee and begin the next iteration.

After many iterations, we ran out of soundness violations and began investigating rules that were never triggered. For these, we would manually study the rule and think of a program that should trigger it according to the rule's preconditions. We would then manually create a test case for that program. If the rule was triggered, we did not let the fuzzer run long enough and simply had a coverage issue. If the rule was not triggered, we manually debugged the model to understand why the rule did not activate. We fixed the problem, and began the next iteration.

5 Model Compiler Design Decisions

5.1 Optimizations

We have found that many challenges in OO analysis are caused by a combination of two optimizations: inlining and unnecessary vftable elimination.

5.1.1 Inlining. Inlining is an optimization that replaces a function or method call with a copy of the callee's code. This can be very beneficial because the copy of the code that is made for each call-site can be optimized for that specific calling context, unlocking optimizations that would otherwise not have been possible. It also avoids the overhead of performing a function call. From an analyst's point of view, however, inlining is a nightmare. Whenever we see code in a function, we cannot simply assume that the action actually originated in that function. Instead, we must always consider if that action could be from an inlined callee. This is most problematic in constructors and destructors. Their regular structure provides a great deal of information, but it can be difficult to tell if a particular action originated in that constructor, or from an inlined call to another constructor. To make matters worse, the compiler makes a separate decision about whether to perform inlining at each callsite. So even if we know that inlining is enabled in a particular executable, it can be difficult to tell whether a particular instruction is from an inlined call.

Visual C++ employs a policy to decide whether inlining would be beneficial, but only the basic design is public knowledge [21]. Generally speaking, the compiler will inline small amounts of code, and will avoid inlining larger amounts of code. Thus, by artificially adjusting the amount of code in the callee, it is fairly easy to control whether Visual C++ inlines a particular call. Since inlining is relatively easy to manipulate, EmCee simply asks whether each call should be inlined (instead of attempting to reverse engineer Visual C++'s inlining policy). This level of control is beneficial in the model compiler, because in our experience any call is capable of being inlined if the callee is sufficiently small. We have yet to discover any case where a rule's correctness hinges on this size limit.

5.1.2 Unused vftable removal. Another optimization that affects OO analysis is unused vftable removal. This optimization observes that in some cases there may be vftable installations that are not needed in constructors and destructors after inlining. For example, if a derived constructor inlines a call to a base constructor that has no user-defined code, the vftable installation for the base class may be optimized away. Similarly, if a trivial derived destructor inlines a call to a base destructor, the vftable installation for the derived class may be optimized away. Although these rules are not complicated in isolation, the implications can be difficult to grasp, especially when combined with multiple layers of inlining.

5.2 Fact generation and modeling

Most facts generated by EmCee are straight-forward to emit, but in this section we describe a few interesting design choices that we made regarding some of the facts.

```
funcParameter(0x4014c0, ecx, sv_2655).
thisPtrOffset(sv_2655, 0xc, sv_630).
callParameter(0x4014cc, 0x4014c0, ecx, sv_630).
callTarget(0x4014cc, 0x4014c0, 0x401160).
```

Figure 9: OOAnalyzer facts stating that function 0x4014c0 receives a thisptr in register %ecx, and passes thisptr + 0xc in a function call to 0x401160.

5.2.1 Thisptrs. One of the primary innovations of OOAnalyzer's predecessor, ObjDigger [19], was to statically track the data flow of potential object pointers throughout the program. These object pointers are called *thisptrs*, and are used to describe important behaviors such as calling a method on a sub-object and installing a vftable into an object. OOAnalyzer uses a symbolic analysis to track the movement of thisptrs through a function from their creation (e.g., as a function argument, a returned value from a function call, or a newly allocated object). OOAnalyzer labels each thisptr by a hash of its symbolic expression. e.g., sv_2655. OOAnalyzer also tracks and describes simple relationships when one thisptr is a constant offset from another. These constant offsets are sufficient to describe the locations of sub-objects, but they are limited and cannot express locations inside of virtual bases, which do not have constant offsets (Section 2.4). For example, Figure 9 shows several thisptr-related facts that demonstrate a call to a sub-object of `this`.

We wanted EmCee to mimic OOAnalyzer's ability to describe thisptr relationships that are constant offsets, but we also wanted EmCee to track all relationships internally so that it can eventually emit facts for non-constant offsets when OOAnalyzer supports it. To accomplish this, we observe that thisptrs represent the computations of *locations* inside of an object. The most general case is when the location is inside of a virtual base. To support this, EmCee models object locations as a tuple consisting of an (optional) virtual base and a constant offset into that virtual base (or the offset from the object's address if no virtual base is specified). When these locations are compiled to instructions, the constant offset is computed using a special instruction, ThisPtrConstOffset. For every use of this instruction, EmCee produces a thisPtrOffset fact that mimics OOAnalyzer's current ability. When the referenced location is inside of a virtual base, the computation of the address of that virtual base is expressed using a different instruction, ThisPtrVarOffset. Instances of this instruction are currently exported to a new fact that OOAnalyzer does not use (yet).

5.2.2 ClassCallsMethod. ClassCallsMethod is a fact in OOAnalyzer that did not have a clear definition. In some rules, it was used to describe methods that can be called on a class's `this` pointer, i.e., methods inherited from base classes. This definition makes sense from a source-code perspective. In other cases, ClassCallsMethod was used to represent any method that appeared to be passed a sub-object. For example, if M1 passes `this` + 8 to M2, then M1's class can call method M2. This is a more executable-oriented definition. The second definition differs primarily in that it includes methods on embedded objects (class objects that are members instead of base classes). More rules used the first definition, so we adopted that definition. This inconsistency serves as a positive example that came simply from trying to formalize the definition of an existing fact. See Section 6.3 for a related case study.

5.3 Limited control-flow modeling

EmCee abstracts away many details of C++ compilers that are immaterial to OO analysis, such as templating and C++ syntax. Much of OO analysis does not require analyzing control flow structures such as conditional branches or loops. This is because OO analysis mostly consists of analyzing compiler-generated code, which usually has straight-line control flow. As such, we decided not to model branches or loops in the model compiler. We did find it necessary to implement support for one case of conditional branching, however. Virtual base constructors (constructors of classes that contain virtual bases) take an argument indicating whether the current constructor is the "most derived" constructor, and if so, invokes the constructor for each virtual base. When such a constructor is called to initialize a base object and the call is inlined, this argument will be false and the calls to virtual bases will be optimized away. This optimization is quite important, so we added a *condition* field to represent the conditions required for compiled assembly instructions to execute. If this condition becomes false, the instructions are dead and EmCee removes them. This allows us to ignore almost all control flow, but still emulate this important effect of inlining virtual base constructors.

6 Case studies: soundness problems in OOAnalyzer

While applying the iterative testing and refinement process on OOAnalyzer, we identified 27 soundness problems in 19 reasoning rules and remedied all of them. In this section, we study a few cases of the most interesting and common types of problems.

6.1 A class must be at least as large as its bases

ClassSizeGTEC says that a class must be at least as large as any of its bases. This is such an intuitive rule that it is hard to imagine that it could be incorrect. After all, if an inner class is *n* bytes, and an outer class contains it, how could the outer class possibly be smaller than *n* bytes? Much to our disbelief, EmCee found that it is possible because of a quirk of virtual inheritance and alignment. Figure 10 shows the counter-example[5] that EmCee generated. `C2`, the inner class, contains two virtual bases, which because of their contents require seven bytes of alignment padding to be placed between them. `C3`, the outer class, inherits virtually from `C0`, and non-virtually from `C2`. Although `C0` is already a virtual base of `C2`, this causes `C0` to be laid out before `C1` in `C3`, and in this order requires no padding. Thus, the derived class `C3` is seven bytes smaller than its base `C2`.

6.2 VftableBelongsToClass

VftableBelongsToClass is one of the most important rules in OOAnalyzer. It is responsible for linking two important concepts in OO executables: classes and virtual function tables. Specifically, VftableBelongsToClass examines constructors and destructors for installations of vftables into the current object, and attempts to determine whether the installed vftable belongs to the same class as the constructor or destructor.

For example, in Figure 11b, which shows the assembly code for a constructor `C1::C1`, VftableBelongsToClass should determine that `C1::vftable` belongs to `C1::C1`'s class, but `C0::vftable` does not. (Of course we can tell which class the vftables belong to by looking at their names, but in a real reverse-engineering scenario they are not labeled.)

```
class C2    size(12)
0 +---
0 | {vbptr}
4 +---
4 +--- (vbase C1)
4 | var0: ClassType{0}
5 +---
5 | alignment
8 | alignment
12+--- (vbase C0)
12+---
```

(a) Inner class `C2`

```
class C3    size(5)
0 +---
0 | +--- (base C2)
0 | | {vbptr}
4 | +---
4 +---
4 +--- (vbase C0)
4 +---
4 +--- (vbase C1)
4 | var0: ClassType{0}
5 +---
```

(b) Outer class `C3`

Figure 10: Outer class `C3` virtually inherits from inner class `C2`, but `C3` is unexpectedly seven bytes *smaller* than `C2`. As can be seen in (a), when virtual base `C1` preceeds virtual base `C0`, Visual C++ adds seven alignment bytes. Because `C3` virtually inherits from `C0`, as can be seen in (b), `C0` is laid out before `C1`, which requires no padding.

VftableBelongsToClass is a complex rule, and it took us *three* attempts to produce an implementation that is sound under perfect knowledge. Here, we'll only describe the most interesting problem that EmCee discovered. Figure 11 shows the counter-example[6] that EmCee reported, but it requires some explanation. The root problem has to do with a clause in VftableBelongsToClass that detects when vftables are overridden, and how that logic interacts with the `InitVBases` branch of constructors (which we will explain shortly). Recall from Section 2.3 that a derived class may override its base's vftable to override a virtual function or add new ones. VftableBelongsToClass contains an important clause that detects overridden vftables and handles them appropriately. Specifically, if a constructor `C::C` first installs a vftable and then installs a new vftable to the same location, this indicates the first vftable was inherited and does not belong to `C`.

As Figure 11b shows, Visual C++ adds a hidden `InitVBases` argument to constructors of classes that contain a virtual base. As the name suggests, the `InitVBases` argument controls whether virtual bases are constructed. The compiler sets this flag when constructing a most-derived object (i.e., the object is *not* being constructed as a base for another object). Such constructors consist of a branch that only executes when `InitVBases` is set (line 3 in Figure 11b) and an unconditional branch that always executes (line 7).

EmCee detected that when a vftable is installed for a virtual base, the clause that detects overwritten vftables will not work as intended. On line 5 of Figure 11b, the constructor installs `C0`'s vftable at offset 4. Because virtual bases are only initialized when constructing a most-derived object, the compiler statically knows the offset at which to install the vftable. In contrast, on line 10 the constructor installs `C1`'s vftable to the same location but without using a static offset. Because this installation occurs on the unconditional branch, the compiler does not statically know the location of `C0`, and must compute it using a vbtable (Section 2.4). Because both vftables are installed to the same location, `C0`'s vftable is overwritten and cannot belong to `C1`.

[5] https://godbolt.org/z/4febr1szK

[6] https://godbolt.org/z/nTvsf3jjW

```
class C1 size(8):
    +---
 0  | {vbptr}
    +---
    +--- (vbase C0)
 4  | {vfptr}
    +---
```

(a) `C1`'s layout

```
1   cmp $initVBases$[esp-4], 0
2   je SHORT $LN5@C1
3   ; initVBases branch
4   mov [ecx], C1::vbtable
5   mov [ecx+4], C0::vftable
6   $LN5@C1:
7   ; unconditional
8   mov eax, [ecx]
9   mov eax, [eax+4]
10  mov [ecx+eax], C1::vftable
```

(b) `C1::C1` assembly snippet

Figure 11: VftableBelongsToClass counter-example that demonstrates how two different vftables are installed to the same location using different mechanisms.

Unfortunately, because lines 5 and 10 reference that location in two different ways, VftableBelongsToClass failed to detect the override, and incorrectly determined that `C0::vftable` belongs to the constructor's class, `C1`. To fix this problem, we modified VftableBelongsToClass so that it would disregard any vftable installed on the `InitVBases` branch, since OOAnalyzer cannot currently determine if they are overridden. A longer term solution would be to analyze the vbtable lookups and compute the relative offsets to determine if they override any vftables installed in the `InitVBases` branch.

6.3 ClassCallsMethodC

The ClassCallsMethodC rule is interesting because it shows the importance of precisely defining the meaning of each fact. We also made several mistakes when "fixing" the rule, but EmCee caught them quickly, which demonstrates the iterative nature of rule development that EmCee encourages. Originally, the rule claimed that if a method `m1` appears to call method `m2` at the executable level (e.g., `m1` receives a thisptr in %ecx, and then passes the same thisptr in %ecx to `m2`), then `m1`'s class can call `m2`. On the face of it, this rule seems trivially correct. After all, we seem to have an example of `m1`'s class calling `m2`.

```
class C1
0 +---
0 | v0: C0
0 +---
```

Figure 12: Counter-example to ClassCallsMethodC

EmCee discovered a simple counter-example to this rule, shown in Figure 12, in which `m1` is on `C1`, and `m2` lives on `C0`, which is embedded in class `C1` at offset 0. As we discuss in Section 5.2.2, "callability" can be defined in a source-oriented definition, or an executable-oriented one. In this paper, we adopted the source-oriented definition, which does not consider methods on embedded classes to be callable by the outer class, and under this definition the rule incorrectly concluded that `m1`'s class could call `m2`.

We first tried to apply a "band-aid fix"; we added a constraint that `m1`'s class did not embed an object at offset 0. Testing immediately revealed an updated counter-example, in which `m1`'s class inherits from a middle-man class at offset 0, which in turn embeds `m2`'s class at offset 0. As a fairly constraining solution, we changed the rule to only apply when `m1`'s class contained no sub-objects at all.

We found that this formulation did not have any soundness violations, but unexpectedly had zero recall, meaning it was not triggered during testing. In retrospect, the rule was designed to make observations about method calling in the case of inheritance, and the very conservative "fix" excluded this case. When we revisited the rule, we formulated the more precise restriction that there cannot be any embedded object at offset 0 in the layout of `m1`'s class, including any combination of inheritance and embedding. The newly formulated rule required a bit of prolog machinery to implement, but it proved to be sound and was actually triggered. As an extra bonus, we also noticed that the new rule applies to calls at *any* offset, not just at offset zero as it was originally formulated.

```
class C1    size(16)
0 +---
0 | {vfptr}
4 | {vbptr}
8 +---
8 +--- (vbase C0)
8 | {vfptr}
12+---
```

(a) `C1`'s class layout

```
1  lea esi, [ecx+8]
2  mov ecx, esi
3  ; Call to offset 8
4  call virtual C1::~C1
5  ; Install at offset 8
6  mov [esi], C0::vftable
```

(b) `C1`'s vbase destructor assembly code fragment

Figure 13: CtorDtorSpecial counter-example

6.4 Thisptr adjustments

EmCee identified several rules that failed to correctly handle thisptr adjustments. As we discuss in Section 2.3, Visual C++ may assign a thisptr adjustment to virtual methods. A thisptr adjustment A on `C::M` indicates that method `M` expects to be called with a thisptr pointing A bytes past the start of a `C` object. This means that if `M` accesses offset O of its thisptr, it is actually accessing offset $O + A$ of a C object.

This is potentially a problem for any rule that infers something about the layout of an object by observing some behavior at a particular thisptr offset. For example, ObjectInObjectF originally would conclude (under some conditions) that if method `M` installs a vftable into offset O, there must be an embedded object at O. However, the rule failed to consider that `M` might have a thisptr adjustment. In these cases, EmCee reported that rules such as ObjectInObjectF would (unsoundly) conclude inner objects exist at *negative* offsets, which does not make sense. In reality, the offsets had to be adjusted by `M`'s thisptr adjustment.

6.5 CtorDtorSpecial

The CtorDtorSpecial rule uses an inheritance-related trick to distinguish whether a method is a constructor or destructor based on the ordering of its actions. Specifically, when a derived class inherits from a base class and overrides the base's vftable, a derived constructor will install the derived vftable *after* calling the base constructor, whereas the derived destructor will do so *before*. The rule only works when the call to the base constructor or destructor is not inlined. The rule presumes there is no other way for such call-and-install pairs to be produced.

Figure 13 shows the counter-example[7] that EmCee discovered, which demonstrates that call-and-install pairs can also be produced by a vbase destructor under certain conditions. As can be seen in Figure 13b on lines 4 and 6, there appears to be a method call at offset

[7]https://godbolt.org/z/x8zh1WPMG

8 followed by a vftable install to the same offset. Since the vftable install follows the method call, it fools the CtorDtorSpecial rule into concluding that the vbase destructor is a constructor.

You may be wondering why the call on line 4 appears to be at offset 8, when both the caller and callee methods are on `C1`. The callee is `C1::~C1`, which is a virtual function that overrides the inherited virtual function `C0::~C0`. As a result, `C1::~C1` has a thisptr adjustment of 8, which explains why it appears to be called at offset 8. The vftable install on line 6 comes from the inlined call to `C0::~C0`.

This is a another example of how multiple behaviors which are understood in isolation can interact in very complex ways. Specifically, this counter-example requires `C1::~C1` to have a thisptr adjustment; simultaneously, the vbase destructor's call to `C0::~C0` must be inlined, but the call to `C1::~C1` must not be. This is a lot for a human analyst to consider.

6.6 NOTConstructorG

NOTConstructorG is an interesting rule for a few reasons. First, it is simple to state, but the justification for the rule is quite complex. Second, its behavior depends on some very low-level details about how Visual C++ lays out classes. Because of these reasons, this rule demonstrates how cognitively difficult it can be to reason about the correctness of rules. Perhaps unsurprisingly, EmCee uncovered a subtle flaw that occurs when some calls in the program are inlined, and others are not.

The rule itself claims that if method *caller* calls method *callee* at offset 0, and *caller* installs a vftable at offset 0, then *callee* can only be a constructor if it too installs a vftable at offset 0. This rule is notable in that it does *not* require *callee* to be on *caller*'s class, and can be applied even if it is unknown whether the two methods are on the same class. As we said earlier, the justification for the rule is complex, and has three cases depending on the callee. (1) *Callee is the constructor for the caller's base class at offset 0.* In this case, the *callee*'s class must have a `vfptr` at offset 0. If it did not, *caller*'s class would have its `vfptr` at offset 0, which means that the base class could not also be at offset 0. Because its class has a `vfptr`, *callee* must install a vftable if it is a constructor. (2) *Callee and caller are on the same class.* We know from *caller*'s vftable installation that it has a vfptr. Therefore, *callee* must install a vftable if it is a constructor. (3) *Callee is on an embedded class at offset 0.* This case is not possible. The *callee*'s class must own the `vfptr`. The *caller*'s class cannot own the `vfptr` because *caller*'s class is already embedded at offset 0. The only way that *caller*'s class could install a vftable into an embedded class is through inlining, but the call to *callee* is visible, so it is not inlined.

During the testing process, EmCee made us aware of a special subcase that we did not think of: when *callee* is a constructor for an *empty* base class (Figure 14). The rule happened to handle this correctly, but again the justification is complex. Say that `C2` inherits from an empty base class `C1`, and then `C0`, which has a vftable[8]. If laid out in order, *both* `C1` and `C0` would be located at offset 0 of `C2`, and the rule would unsoundly conclude that the constructor of the empty base class `C1` was *not* a constructor because it did not install a vftable at offset 0. However, because `C0` has a `vfptr` field that it can reuse, Visual C++ will place it first (Section 2.2), regardless of the source-code order. The empty base class `C1` will be located at offset 4, and the rule will (correctly) not apply. This is an example where the correctness of a rule depends on a very fine detail of the compiler (i.e., re-ordering base classes when a `vfptr` field is reused).

```
class C2    size(4)
0 +---
0 | +--- (base C0)
0 | | {vfptr}
4 | +---
4 | +--- (base C1)
4 | +---
4 +---
```

Figure 14: Complex case in NOTConstructorG when the base class is empty.

Although the rule handled the above example correctly, EmCee found that a combination of inheritance, embedding, and inlining can still trigger a problem with empty base classes (Figure 15). Specifically, if `C2` inherits from an empty base class `C0` and then embeds `C1`, which contains a vftable, inlining can cause `C2::C2` to consist of a call to `C0::C0` and C1's vftable to be installed at offset 0[9]. For this to happen, `C1::C1` must be inlined into `C2::C2`, but `C0::C0` cannot be.

```
class C2    size(4):
  +---
0 | +--- (base C0)
0 | +---
0 | C1 c1
  +---
```

Figure 15: Counter-example to NOTConstructorG when the base class is empty.

6.7 NOTRealDestructorG

NOTRealDestructorG is the analog of NOTConstructorG (Section 6.6) but for destructors instead of constructors. It claims that if a caller installs a vftable to offset 0, and calls a callee at offset 0, then the callee may only be a real destructor if it also installs a vftable to offset 0. Unfortunately, the destructor version of this rule is fundamentally flawed because although default constructors install vftables, default destructors do not. Thus, if the callee is a default destructor, it can be a destructor but not install a vftable. Lacking a better solution, we disabled this rule.

7 Case Study: VirtAnalyzer

In this section, we explore the generality of EmCee by applying it to another popular system for OO analysis, VirtAnalyzer [10]. Rather than being constructed as a set of cooperating rules, VirtAnalyzer consists of several algorithms that are executed sequentially. The core algorithm, which identifies virtual inheritance relationships, is based on a simple premise: if VirtAnalyzer finds a constructor or destructor that appears to install a vbtable into the current object, it analyzes the vbtable as follows. If the vbtable reveals that the object has a virtual base at offset O, VirtAnalyzer looks for a method call at offset O. If it finds one, it reasons that the called method is a constructor or destructor for the virtual base, and concludes there is a

[8]https://godbolt.org/z/T9bWfhhPo

[9]https://godbolt.org/z/hT74G3Gzj

virtual inheritance between the outer constructor (or destructor) and the inner constructor (or destructor). At a high level, this reasoning is valid, and OOAnalyzer contains a rule inspired by the same idea. But following the theme of this paper, the devil is in the details.

We tested VirtAnalyzer's reasoning algorithm by converting it into an OOAnalyzer rule that implements the same logic, which can be found in Section A. We based our rule on the open-source VirtAnalyzer implementation for Visual C++ binaries [2]. We verified each flaw[10] by producing an executable compiled by MSVC that reproduced the flaw in the original VirtAnalyzer implementation [2].

7.1 Magic Offsets

```
class C3    size(8)
0 +---
0 | {vbptr}
4 | var0: Class{1}
5 | alignment
8 +---
8 +--- (vbase C0)
8 +---
```

Figure 16: Magic offsets counter-example

EmCee quickly found a problem (shown in Figure 16) with the way that VirtAnalyzer reasons about `vbptr` offsets, the offset between the start of an object and its `vbptr`. Each offset O in a vbtable is actually an offset from the `vbptr`, which is not always the start of the object (Section 2.4). VirtAnalyzer addresses this with a list of *magic offsets* that it considers as possible `vbptr` offsets. To be more specific, when VirtAnalyzer sees an offset O in a vbtable, it adds each magic offset MO to O, and if there appears to be a method call at $O + MO$, it concludes the callee is on the virtual base. EmCee found that this can cause VirtAnalyzer to mistake an embedding relationship for virtual inheritance. For example, in this model,[11] `C3` virtually inherits from `C0` at offset 8, so `C3`'s vbtable contains an entry of 8 (since `C0` is located at offset 8 from the `vbptr` at offset 0). `C3` also embeds a `C1` object at offset 4; as a result, `C3`'s constructor calls `C1`'s constructor at offset 4. Unfortunately, because VirtAnalyzer includes a magic offset value of −4, VirtAnalyzer mistakenly associates this call with `C0` instead of `C1` because $8 + -4 = 4$, which matches the observed method call offset. As a result, VirtAnalyzer incorrectly concludes that `C3` virtually inherits from `C1`.

Fortunately, there is a simple solution. Rather than trying each magic value as a possible offset, MSVC stores the `vbptr`'s offset in the first entry of every vbtable. One can simply use this stored value as the offset. It also applies more generally, because it allows for `vbptr` offsets that are not covered by the static list of magic offsets.

7.2 Constructor inlining

The next problem that EmCee found was that VirtAnalyzer's reasoning algorithm fails to soundly reason about inlined constructors when multiple objects start at the same offset. For example, in Figure 17,[12] `C2` virtually inherits from class `C1` at offset 4, but `C1` also embeds a `C0` object at the same location.

[10]This excludes the inherited vbptrs flaw (Section 7.3), since we inadvertently introduced that ourselves when attempting to fix the magic offsets flaw (Section 7.1).
[11]https://godbolt.org/z/zxneK7aob
[12]https://godbolt.org/z/Eoozajchx

```
class C2    size(7)
0 +---
0 | {vbptr}
4 +---
4 +--- (vbase C1)
4 | var0: Class{0}
5 | var1: Class{0}
6 | var2: Class{0}
7 +---
```

Figure 17: Inlined constructors counter-example

Without inlining, `C2::C2` will call `C1::C1` at offset 4, which will call `C0::C0` at `C1`'s offset 0. The problem occurs when `C1::C1` is inlined into `C2::C2`, but `C0::C0` is not inlined into `C1::C1`. When this happens, `C2::C2` calls `C0::C0` at offset 4, which VirtAnalyzer incorrectly interprets as `C0::C0` being a constructor for `C2`'s virtual base class `C1`, since it is stored at the same offset. We fixed this rule by ensuring that only one object starts at the relevant offset, which ensures there is no ambiguity about which object's constructor is invoked. Unfortunately, while this type of constraint is possible in OOAnalyzer, we are not aware of a practical solution for the actual VirtAnalyzer implementation.

7.3 Inherited `vbptrs`

```
class C2   size(8)
0 +---
0 | {vfptr}
4 | +--- (base C1)
4 | | {vbptr}
8 | +---
8 +---
8 +--- (vbase C0)
8 +---
```

Figure 18: Inherited `vbptrs` counter-example

In Section 7.1, we fixed a soundness problem by leveraging the offset from the `vbptr` to the start of the object, which is always stored in the first entry of a vbtable. But this offset is to the object *that owns the* `vbptr`, which *is not always the class that owns the vbtable.* This is further exacerbated by inlining, which obscures the object that owns the `vbptr`.

In Figure 18,[13] `C1` owns the `vbptr`, which is located at offset 0 of `C1`, and so the first entry of `C1`'s vbtable is zero. Since `C2`'s vbtable is also installed in `C1`'s `vbptr`, the first entry of `C2`'s vbtable is *also* zero, even though the `vbptr` is at offset 4 of `C2`. EmCee found that when `C1`'s constructor is inlined into `C2`'s constructor, the rule thinks that `C2` owns the `vbptr`. This is a problem, because it interprets the zero offset as being the start of `C2` rather than `C1`, which is located at offset 4 in `C2`. Thus, it wrongly concludes that `C2` inherits from `C0` at offset 4 (instead of at offset 8).

The solution is to use the offset at which the vbtable is installed to, which is visible from the constructor's assembly code, rather than the offset in the first entry of the vbtable. We want to emphasize that *we* introduced this soundness violation in our first "fix" to VirtAnalyzer, and that VirtAnalyzer's authors did not make this mistake. But we included it here because it demonstrates another example of how EmCee encourages an iterative approach to rule development by quickly identifying faulty changes.

[13]https://godbolt.org/z/bKx3Ws586

```
// Part 1: Class definitions
class C0 {...}; class C1 : C0 {...};

// Part 2a: verify class size
const C1* obj = new C1(special_constructor());
const size_t vc_size = sizeof(C1);
assert(vc_size == 4);
// Part 2b: verify base offset
const C0* baseobj = (C0*) obj;
size_t c = (uintptr_t)baseobj - (uintptr_t)obj;
assert(c == 4);
```

Figure 19: Example validation program

8 Model Validation

Because our approach to testing is model-based, it is critical for the model to accurately reflect the behavior of the real Visual C++ compiler. As we note in Section 4, the refinement process itself identified many bugs in the model. In this section, we describe additional validation (and repair) that we performed after the refinement process finished. Specifically, we explicitly validate EmCee's ability to construct a class's data layout (Section 2.2).

EmCee can produce a C++ program that is then compiled with Visual C++ and executed to validate that the model's layout matches the one produced by Visual C++. The program consists of two parts, as shown in Figure 19. The first portion includes a reconstruction of the test case's source code program. The second portion consists of automatically generated checks which compares the data layout assigned by the Visual C++ compiler with the layout determined by the model. For each class `C`, the program computes the size of the class using `sizeof(C)` and compares it to the model's computed size of the class, which is hard-coded into the program. Similarly, for each base `B` of `C`, the program computes the offset from the start of `C` to the embedded location of base `B`. This indirectly validates the model's understanding of padding and alignment, and the offsets of individual fields. The resulting executable will emit an error message for any disagreement.

8.1 Discovered discrepancies

While validating the data layout of our model compiler, we found many programs where the model's layout did not match Visual C++'s layout. In the process, we (re)discovered several bizarre data layout behaviors in the Visual C++ ABI.

The empty base optimization (EBO) allows empty base classes to take zero space in an object. We rediscovered an old bug in Visual C++[14], which is that it only applies EBO to the *last* empty base on a class. In Figure 20, classes `EB0` and `EB1` are both empty, but `EB0` takes one byte while `EB1` takes zero because of EBO. Normally, EBO would allocate zero bytes for both classes. Knowledge of this bug dates back to at least 2006 but it remains unfixed in Visual C++ 2019 unless one uses a special class declaration. The problem is that the bug has effectively become encoded into Visual C++'s ABI.

We also encountered a confusing condition where Visual C++ unexpectedly adds more padding than is necessary (Figure 21). We are not the first to notice this unusual problem [18]. One of Visual C++'s authors explained that the padding is needed because `vfptr` and `vbptr` fields are added *after* other fields are already laid out. The

[14]https://stackoverflow.com/a/12714226/670527

```
class C1   size(2):
  +---
0 | +--- (base EB0)
  | +---
1 | +--- (base EB1)
  | +---
1 | bool first_mem
  +---
```

Figure 20: Empty base optimization counter-example

extra alignment is added to preserve each fields natural alignment. For example, if a field must be 8-byte aligned and a 4-byte `vfptr` field is added, the 8-byte alignment would be broken without additional padding.

```
class C1   size(24):
  +---
 0| {vfptr}
 8| long x
12| <alignment> (4)
16| long long y
  +---
```

Figure 21: Unexpected padding counter-example

The final unexpected feature was that Visual C++ adds one byte of padding between bases to avoid aliasing between zero-sized bases. For example, in Figure 22[15], there is a byte of padding at offset 4 between classes `C2` and `C0`.

```
class C3    size(8):
  +---
 0| +--- (base C2)
 0| | C1 v1
 0| +---
 5| +--- (base C0)
  | +---
  | <alignment> (sz=3)
```

Figure 22: Padding between bases counter-example

9 Discussion

9.1 Tension Between Soundness and Accuracy

The original motivation of this project was to proactively prevent soundness failures from interfering with OOAnalyzer's execution (as opposed to improving accuracy). OOAnalyzer can detect unsoundness in certain cases. When it does, it intentionally terminates and prompts the user to file an issue. Before EmCee, this was how we learned about most soundness problems in OOAnalyzer. Over time, we found that some rules would repeatedly cause problems. One of the most notorious problems was the VftableBelongsToClass rule (Section 6.2). We would diligently analyze the issue, refine our understanding, and implement fixes, only to learn (in a new issue) that we had not fully addressed the underlying problem. It became apparent that we were approaching the limits of what we could do without having some form of assistance. EmCee has allowed us to largely eliminate these types of problems.[16]

[15]https://godbolt.org/z/1brshxGG7

[16]As a timely but anecdotal example, as we were writing this paper we received a report of a new soundness issue in one of the few rules we had not tested in this paper.

Despite this success, when we embarked on this project, we also expected that addressing the discovered issues would lead to improved accuracy. However, we found that addressing soundness failures generally had a negative impact on accuracy. There are a few reasons for this, but fundamentally there is a tension between soundness and accuracy because accuracy is weighted by frequency and soundness is not. To illustrate this tension, imagine a rule that is correct for 99.9% of real-world programs, and improves overall accuracy by 5% on a representative benchmark. When viewed through the lens of accuracy, the rule is clearly beneficial, since it improves accuracy by a substantial margin, and is only incorrect for a small fraction of programs. But when viewed through the lens of soundness, it does not matter that it is only incorrect for a small fraction of programs: it is unsound, full stop. The real tension depends on how difficult it is to repair the unsound rule. In the best case scenario, the unsound condition can be easily characterized and filtered out by adding an additional clause to the rule that prevents only the unsound conclusions. In this case, we can eliminate the unsoundness but keep the improvement to accuracy; everyone wins! In the worst case, however, it is not possible to distinguish sound from unsound conclusions, and we need to disable the rule in its entirely. This would "cost" us 5% accuracy, even though it might only negatively impact a small subset of programs.

Thisptr adjustments are a good example of this tension (Sections 2.3 and 6.4) because they impact many rules, are not easily detected, and are only present in (relatively) complex inheritance hierarchies. Remember that thisptr adjustments occur when a derived class overrides a virtual function in a base class. For there to be a non-zero thisptr adjustment, the program must contain multiple inheritance. Unfortunately, it is difficult to know whether there is a thisptr adjustment. If we know a class does not inherit from any bases, we know there is not a thisptr adjustment for that class. But this is tricky, because the best evidence for not having base classes is simply a lack of evidence *for* base classes. This type of problem leads to negation and ordering challenges: if we did not find evidence for inheritance, is it because there is no inheritance, or because we have not yet found the evidence? In OOAnalyzer, we addressed this by implementing a *hypothetical reasoning rule* that, in the absence of evidence to the contrary, assumes that there is no inheritance, and thus no thisptr adjustment. Unlike the reasoning rules we discuss in this paper, hypothetical reasoning rules are allowed to make untrue conclusions by design, but happen later in the overall reasoning process. Unfortunately, delaying such decisions can substantially change the order in which OOAnalyzer's rules interact, which in turn can alter accuracy by itself.

9.2 Future Work

EmCee is a proof of concept implementation of our more general iterative testing and refinement process (Section 4). Although the *process* is not specific to any architecture, EmCee itself is limited in that it only models the MSVC compiler for 32-bit x86 code. This is largely because we first applied EmCee to refine OOAnalyzer, which has the same limitation. The largest obstacle to adding support for 64-bit and the Itanium ABI (e.g., Linux) is actually a lack of support from pre-existing OO abstraction recovery tools. Although VirtAnalyzer supports 64-bit and the Itanium ABIs, it consists of far fewer rules than OOAnalyzer. This is problematic because, in our iterative approach, the OO analysis tools also help identify inconsistencies in the model compiler. Intuitively, the more reasoning that a tool performs, the more the model compiler will be stress tested. Based on our experience with EmCee, we believe the best way to add support for both 64-bit and the Itanium ABI executables would be to begin the iterative process using VirtAnalyzer, and then to implement the new features in OOAnalyzer. Our experience with repairing the faulty rules in this paper leads us to believe that employing our proposed refinement process will make implementing these features significantly easier by providing an immediate feedback mechanism. Specifically, we found that when EmCee immediately detected problems in rules we "fixed," it naturally encouraged rapid iterative development (Sections 6.2 and 7.3).

10 Related Work

10.1 Recovery of C++ Abstractions

The problem of recovering C++ abstractions from executables has been studied for years. Early work focused on leveraging obvious sources of data such as vftables and RTTI metadata, but more recently, researchers have been studying how to recover abstractions for non-polymorphic classes (i.e., classes without virtual functions) using other sources of data. ObjDigger [19], OOAnalyzer's predecessor, pioneered the idea of statically tracking the data flow of object pointers throughout the program, and roughly collecting a map between object pointers and the methods they appear to be invoked on using static binary analysis. With this data, ObjDigger was able to assign methods to classes in the presence of relatively simple class hierarchies. OOAnalyzer [32] expanded on the idea of tracking the data flow of object pointers, but observed that accurately making conclusions from such observations can be very difficult in larger programs. To address this, OOAnalyzer added a Prolog-based rule system to help make additional conclusions from the set of existing conclusions. OOAnalyzer's authors also observed that some properties are ambiguous, and devised a system of hypothetical reasoning that backtracks when the reasoning process detects a contradiction.

Lego [33] is another early system that is notable in its ability to recover information about non-polymorphic classes. Like ObjDigger and OOAnalyzer, Lego gathers information about the methods that are invoked on various object pointers. Unlike those systems, Lego gathers this information by employing *dynamic* binary analysis. That is, it runs the program on test cases and records a trace of method calls.

Several systems are able to recover the class hierarchy for only *polymorphic* classes (i.e., classes with a virtual function) by either (1) utilizing RTTI (run-time type information) metadata that is sometimes included in executables [13, 14, 36], or (2) analyzing virtual function tables, constructors, and destructors [10, 13, 14]. SmartDec [13, 14] is an executable to C++ decompiler that can use RTTI metadata to recover the polymorphic class hierarchy. If RTTI is unavailable, it will analyze vftables, constructors, and destructors to recover the hierarchy. More recently, the VirtAnalyzer system [10] was introduced, which also recovers the class hierarchy using a combination of vftable, constructor and destructor analysis. The VirtAnalyzer system focuses on correctly recovering instances of *virtual* inheritance (Section 2.4) in the class hierarchy, which they show to be pervasive and important for security protections. They note that most existing abstraction recovery systems do not handle virtual inheritance correctly. Our experience in this paper supports the notion that reasoning about virtual inheritance can be challenging and unintuitive.

10.2 Security protections for C++ binaries

Another body of research that utilizes C++ abstractions is the work on security protections such as control-flow integrity (CFI). Early CFI systems would automatically infer expected control-flow transitions from source code, and would include runtime checks to ensure that the program was safely terminated if any unexpected control-flow transitions were taken [1, 4]. Subsequently, researchers demonstrated that similar protections can be added to executables without requiring access to source code [35, 39]. Several attacks demonstrated that some CFI systems were permitting too many executions in the name of lowering overhead [5, 6, 11, 17, 31], and in response, several systems were proposed that incorporate specific protections for C++ features [9, 23, 24, 37] to achieve stronger protections. vfGuard [24] and VTint [37] both recover information about virtual call sites and vftables. More recent work has also focused on recovering information about the inheritance hierarchy, including VCI [9] and MARX [23]. Improved recovery of OO abstractions from executables allows for improved security in such systems without the aid of source code for the protected program.

10.3 Provable decompilation

This paper was inspired by one of the first papers on provably correct decompilation and type recovery by Robbins et al. [27]. Although we do not think of OOAnalyzer as a decompiler, the OO abstractions it recovers can be thought of as a subset of the decompilation process. In Robbins, the authors formally define a simple programming language based on C, MinC, and a low-level assembly code based on x86, MinX. They formally define a structural semantics for each language, which they then use to define a decompiler relation from a low-level MinX program back to a corresponding MinC program. They formally prove that their decompilation relation is conservative. One of the high-level ideas we borrow from their paper is that it is first necessary to have a definition of forward compilation in order to define the correctness of *decompilation*. In their case, a definition of compilation naturally arises for pairs of MinX and MinC programs that have equivalent semantics. In other words, their decompiler does not leverage knowledge about the compiler's behavior. In our work, we focus on a particular compiler; by using our knowledge of how the compiler implements certain high-level program structures, we are better able to relate those implementations back to high-level structure.

There are several other papers relating to provably correct decompilation. FoxDec [34] is a sound x86-64 to C decompiler. Rather than targeting a specific compiler, they prove that each phase of the decompilation process is sound (i.e., does not change the semantics of the program). A very different approach to correct decompilation is *perfect* or *exact decompilation* [3, 30], in which the goal is to recover source code that, when compiled, produces binary code that is syntactically identical to the original (and thus trivially semantically equivalent). C++ object layout has also been studied in the context of provably correct *compilation* [25].

11 Conclusion

OO abstraction recovery is an extremely challenging problem, and even state-of-the-art tools are unsound. In this paper, we proposed a new model-based technique for systematically testing C++ abstraction recovery systems. We created EmCee, a model compiler that closely mimics the behavior of Microsoft's Visual C++ compiler on 32-bit x86 executables. We used EmCee to systematically test and refine the rules of the two leading C++ abstraction recovery systems, OOAnalyzer and VirtAnalyzer. Although we applied our technique retroactively, we believe that the iterative nature of rule development that our approach encourages will accelerate the creation of reasoning rules for new architectures and compilers, as well as open the door to rules that were previously too difficult for a human to understand.

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[2] Erinfolami Rukayat Ayomide. 2020. VirtAnalyzer Implementation for Visual C++ Executables. Retrieved August 29, 2025 from https://github.com/bingseclab/VirtAnalyzer/blob/c8e0bd22b4ea5c3036d8b79b23a1b044520ac378/recover_virtual_inheritance_msvc.py

[3] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. 2022. Decomperson: How Humans Decompile and What We Can Learn From It. In *Proceedings of the USENIX Security Symposium.*

[4] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50, 1, Article 16 (April 2017). doi:10.1145/3054924

[5] Nicholas Carlini and David Wagner. 2015. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the USENIX Security Symposium.*

[6] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the USENIX Security Symposium.*

[7] David Dewey and Jonathon Giffin. 2012. Static Detection of C++ Vtable Escape Vulnerabilities in Binary Code. In *Proceedings of the Network and Distributed System Security Symposium.*

[8] David Dewey, Bradley Reaves, and Patrick Traynor. 2015. Uncovering Use-After-Free Conditions in Compiled Code. In *Proceedings of the IEEE Conference on Availability, Reliability and Security.*

[9] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. 2017. Strict Virtual Call Integrity Checking for C++ Binaries. In *Proceedings of the ACM Asia Conference on Computer and Communications Security.*

[10] Rukayat Ayomide Erinfolami and Aravind Prakash. 2020. Devil is Virtual: Reversing Virtual Inheritance in C++ Binaries. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[11] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the USENIX Workshop on Offensive Technologies.*

[13] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE).*

[14] Alexander Fokin, Katerina Troshina, and Alexander Chernov. 2010. Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *Proceedings of the Software Maintenance and Reengineering Conference*.
[15] Matt Godbolt. 2024. Compiler Explorer. http://www.godbolt.org.
[16] Jan Gray. 1994. *C++: Under the Hood*. Technical Report. Microsoft. Retrieved August 29, 2025 from https://web.archive.org/web/20250614023525/https://www.openrce.org/articles/files/jangrayhood.pdf
[17] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[18] Sam Hocevar. 2012. The stolen bytes: Visual Studio, virtual methods and data alignment. Retrieved August 29, 2025 from https://web.archive.org/web/20230107202959/http://lolengine.net/blog/2012/10/21/the-stolen-bytes
[19] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. 2014. Recovering C++ Objects From Binaries Using Inter-procedural Data-Flow Analysis. In *Proceedings of the Program Protection and Reverse Engineering Workshop*.
[20] Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. Estimating Types in Binaries Using Predictive Modeling. In *Proceedings of the Symposium on Principles of Programming Languages*.
[21] Terry Mahaffey. 2019. Inlining Decisions in Visual Studio. Retrieved August 29, 2025 from https://web.archive.org/web/20250710025931/https://devblogs.microsoft.com/cppblog/inlining-decisions-in-visual-studio/
[22] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47 (2021).
[23] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX: Uncovering Class Hierarchies in C++ Programs. In *Proceedings of the Network and Distributed System Security Symposium*.
[24] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the Network and Distributed System Security Symposium*.
[25] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. 2011. Formal Verification of Object Layout for C++ Multiple Inheritance. In *Proceedings of the Symposium on Principles of Programming Languages*.
[26] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*.
[27] Ed Robbins, Andy King, and Tom Schrijvers. 2016. From MinX to MinC: semantics-driven decompilation of recursive datatypes. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*.
[28] Jesse Ruderman. 2008. Lithium: Line-based testcase reducer. Retrieved August 29, 2025 from https://github.com/MozillaSecurity/lithium
[29] Paul Vincent Sabanal and Mark Vincent Yason. 2007. Reversing C++. In *Proceedings of Black Hat USA*.
[30] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. 2018. Evolving Exact Decompilation. In *Proceedings of the Workshop on Binary Analysis Research*.
[31] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[32] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. 2018. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[33] Venkatesh Srinivasan and Thomas Reps. 2014. Recovery of Class Hierarchies and Composition Relationships from Machine Code. In *Proceedings of the International Conference on Compiler Construction*.
[34] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. 2020. Sound C Code Decompilation for a Subset of x86-64 Binaries. In *Software Engineering and Formal Methods*.
[35] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. 2015. Binary Code Continent: Finer-grained Control Flow Integrity for Stripped Binaries. In *Proceedings of the Annual Computer Security Applications Conference*.
[36] Kyungjin Yoo and Rajeev Barua. 2014. Recovery of Object Oriented Features from C++ Binaries. In *Proceedings of the IEEE Asia-Pacific Software Engineering Conference*.
[37] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity.. In *Proceedings of the Network and Distributed System Security Symposium*.
[38] Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls.. In *Proceedings of the Network and Distributed System Security Symposium*.
[39] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the USENIX Security Symposium*.

A VirtAnalyzer Rule Modeling

Figure 23 displays the implementation of the primary reasoning rule of VirtAnalyzer [2, 10] in OOAnalyzer's rule language, which we test in Section 7. We based our rule on the open-source VirtAnalyzer implementation for Visual C++ binaries [2].

```
reasonDerivedClass_VirtAnalyzer(DerivedClass, BaseClass, AdjustedOffset, virtual) :-
  % We see the outer method installing a vbtable
  factVBTableWrite(_Insn, OuterMethod, _UnusedObjVBPtrOffset, VBTableAddress),

  % And a vftable
  factVFTableWrite(_Insn2, OuterMethod, _UnusedObjVFPtrOffset, _UnusedVFTableAddress),

  % VirtAnalyzer uses "vbase magic" offsets as a signature to detect vbtables
  % (0, 0xffffff20, 0xffffffc0, 0xfffffe28, 0xfffffffc, 0xfffffff8)
  MagicOffsets = [0, 4, 8, 64, 224, 472],
  factVBTableEntry(VBTableAddress, 0, NegativeVBPtrOffset),
  VBPtrOffset is -NegativeVBPtrOffset,
  member(VBPtrOffset, MagicOffsets),

  % The signature passed.  Now they look at a non-zero vbtable entry
  factVBTableEntry(VBTableAddress, TableOffset, ObjOffset),
  TableOffset > 0,

  % VirtAnalyzer computes an adjusted offset, trying each of the magic offsets
  member(MagicOffset, MagicOffsets),
  AdjustedOffset is ObjOffset - MagicOffset,

  % We see a call at AdjustedOffset
  callTarget(CallInsn, OuterMethod, InnerMethod),
  funcParameter(OuterMethod, ecx, ThisPtr),
  thisPtrOffset(ThisPtr, AdjustedOffset, CalleeThisPtr),
  callParameter(CallInsn, OuterMethod, ecx, CalleeThisPtr),

  find(OuterMethod, DerivedClass),
  find(InnerMethod, BaseClass).
```

Figure 23: Re-implementation of VirtAnalyzer's main reasoning rule for Visual C++ executables in OOAnalyzer's rule language. Our implementation is based on this code [2].