Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables

Edward J. Schwartz
Carnegie Mellon University
Software Engineering Institute
eschwartz@cert.org

Cory F. Cohen
Carnegie Mellon University
Software Engineering Institute
cfc@cert.org

Michael Duggan
Carnegie Mellon University
Software Engineering Institute
mwd@cert.org

Jeffrey Gennari
Carnegie Mellon University
Software Engineering Institute
jsg@cert.org

Jeffrey S. Havrilla
Carnegie Mellon University
Software Engineering Institute
jsh@cert.org

Charles Hines
Carnegie Mellon University
Software Engineering Institute
hines@cert.org

ABSTRACT

High-level C++ source code abstractions such as classes and methods greatly assist human analysts and automated algorithms alike when analyzing C++ programs. Unfortunately, these abstractions are lost when compiling C++ source code, which impedes the understanding of C++ executables. In this paper, we propose a system, OOAnalyzer, that uses an innovative new design to statically recover detailed C++ abstractions from executables in a scalable manner.

OOAnalyzer's design is motivated by the observation that many human analysts reason about C++ programs by recognizing simple patterns in binary code and then combining these findings using logical inference, domain knowledge, and intuition. We codify this approach by combining a lightweight symbolic analysis with a flexible Prolog-based reasoning system. Unlike most existing work, OOAnalyzer is able to recover both polymorphic and non-polymorphic C++ classes. We show in our evaluation that OOAnalyzer assigns over 78% of methods to the correct class on our test corpus, which includes both malware and real-world software such as Firefox and MySQL. These recovered abstractions can help analysts understand the behavior of C++ malware and cleanware, and can also improve the precision of program analyses on C++ executables.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; *Malware and its mitigation*;

KEYWORDS

software reverse engineering; binary analysis; malware analysis

ACM Reference Format:
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 *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243793

1 INTRODUCTION

Spurred by advances in computer hardware, modern software continues to rapidly grow in complexity, and shows no sign of slowing. To cope with this increasing complexity, software engineers have turned to *object oriented* (OO) programming languages, such as C++, which provide a natural framework of high-level abstractions for constructing large and complex applications. The OO programming paradigm focuses on sophisticated, user-created data structures known as classes that bind related data (members) and operations (methods) together. This organization of related data and operations largely enables developers to manage C++ *source code* more effectively and build more complex software.

Unfortunately, like its predecessor, C++ allows programmers to perform dangerous operations in the spirit of enabling speed and flexibility over security. It is thus no surprise that vulnerabilities in C++ software are a common occurrence, as developers race to develop larger, more complex programs in a potentially insecure language. More surprisingly, malware authors are increasingly writing their malicious code in C++ (*e.g.,* Duqu, Stuxnet, and Flamer) to leverage its engineering benefits as well.

Further compounding these problems is the fact that the high-level abstractions of C++ objects are lost during the compilation process, which makes analyzing C++ *executables* difficult for human analysts and automated algorithms alike. For example, an algorithm searching for *use-after-free* vulnerabilities requires knowledge of object constructors [7], and an analyst attempting to understand a malware sample's behavior would greatly benefit from knowing which methods are on related classes [9]. Researchers have also demonstrated that many exploit protections are more effective with C++ abstractions, and that the level of protection and efficiency improves with the accuracy of the C++ abstractions. For example, researchers in executable-level control-flow integrity (CFI) protection systems [1, 35] have recently shown that the overall level of protection against exploits can be significantly improved by incorporating knowledge of C++ abstractions [8, 19, 21, 34]. Although there are existing systems that can recover C++ abstractions from executables, most of them rely on virtual function tables (vftables) as their

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243793

primary source of information, and as a result only consider polymorphic classes (*i.e.,* classes with virtual methods) [6–10, 15, 19, 33].

In this paper, we address this limitation by developing a new system, OOAnalyzer, that can accurately recover *detailed* C++ abstractions about *all* classes and methods, including the list of classes, the methods on each class, the relationships (*e.g.,* inheritance) between classes, and a list of special methods such as constructors and virtual methods. OOAnalyzer avoids the limitations of prior work by leveraging a sophisticated reasoning system that incorporates information from a variety of sources, including some that yield information about all types of classes (*i.e.,* not just polymorphic classes). For example, OOAnalyzer can observe actions on object pointers, such as method invocations, to learn the relationships between methods and classes, and this information pertains to any method that is invoked in the target program.

OOAnalyzer's design is motivated by the observation that many human analysts reason about C++ programs in an incremental fashion [23, 27]. In particular, they often make simple, low-level findings by spotting patterns in binary code, and then combine these findings using logical inference, domain knowledge, and intuition. OOAnalyzer employs a lightweight static symbolic binary analysis and a Prolog-based inference system to codify the human analyst approach, allowing it to efficiently search for code patterns that are indicative of higher-level OO program properties. More importantly, OOAnalyzer's inference system also allows it to *reason hypothetically* through ambiguous scenarios. When OOAnalyzer is stuck and cannot make progress, it can temporarily promote an uncertain property about the program to higher certainty, enabling OOAnalyzer to reason about the new scenario as if it was true. If that scenario leads to a contradiction, OOAnalyzer uses Prolog's ability to backtrack to search for an alternate reasoning path. This ability is critical for reasoning about OO programs, which often contain ambiguous properties that need to be resolved before reasoning can progress effectively.

OOAnalyzer's inference system allows it to scale to large, real-world programs such as Firefox and MySQL. Because its reasoning component can cope with incomplete, contradictory and ambiguous facts, we designed OOAnalyzer to use a simple but scalable static symbolic analysis to generate the initial facts that serve as the basis for higher level reasoning. OOAnalyzer also gains scalability by reasoning about OO properties in the domain and language of high-level OO abstractions, rather than reasoning purely on detailed, low-level executable semantics.

We also propose a new *edit distance* metric for evaluating the quality of recovered C++ abstractions. Most existing systems recover classes by discovering vftables, which makes evaluation trivial because each vftable can be mapped to its corresponding source code class and compared. Because OOAnalyzer can recover non-polymorphic classes, which do not have a corresponding natural identifier such as vftables, there is not always a clear correspondence between the classes that OOAnalyzer recovers and those in the source code. Edit distance allows us to evaluate the quality of our results without this correspondence. Using our new metric, we show in our evaluation that, on average, OOAnalyzer places over 78% of methods on the correct class, and can distinguish constructors with an average recall and precision of 0.88 and 0.88.

In summary, the contributions of our paper are:

(1) We design and implement OOAnalyzer, a system for recovering detailed C++ abstractions from executables in a scalable manner. OOAnalyzer recovers information about *all* classes and methods, including non-polymorphic classes.
(2) We propose using *edit distance* as a metric for evaluating the quality of C++ abstractions returned by systems such as OOAnalyzer. We show that debug symbols can be used to generate the ground truth for this comparison.
(3) We evaluate OOAnalyzer on malware samples and well-known cleanware programs including Firefox and MySQL. We show that OOAnalyzer is able to accurately recover most C++ classes and their methods (78% of methods on average), and can identify special methods such as constructors, destructors, vftables, and virtual methods (average F-scores of 0.87, 0.41, 0.97, and 0.88).

2 BACKGROUND

We assume that readers are familiar with the basic concepts of C++ such as classes, methods, and members. In this section, we review the more advanced features of C++ that are pertinent to the design of OOAnalyzer, and briefly discuss how Microsoft Visual C++ implements these features. For more information, we refer the reader to other sources [11].

2.1 Virtual Functions

Sometimes a programmer may wish to invoke a method on an object without knowing the object's exact type, in which case we say the method and class are both *polymorphic*. For example, a configuration file may select the class that implements an object. In C++, polymorphic methods are known as *virtual functions*. When a virtual function is invoked, its implementation is selected at runtime based on the object's type (instead of the type of the pointer to the object).

Virtual functions are implemented by including an implicit class member that points to the *virtual function table* (vftable) for the object. The virtual function table contains an entry for each virtual function that can be called on objects of that type. Visual C++ computes these virtual function tables at compile time, and a constructor or destructor may use code like the following to *install* a vftable into the current object:

```
mov eax, objptr
mov [eax], vftableptr
```

Many related works rely on virtual function tables as their primary source of information, and as a result can only recover information about polymorphic classes or virtual functions [6–10, 15, 19, 33].

2.2 Class Relationships

A program's classes can relate to one another in a variety of ways. The two most common relationships are inheritance and composition. When class A inherits from class B, most members and methods on class B will be automatically pulled into the definition of class A. Class A is usually called the *derived* class, and class B is called the *base* class. Inheritance is often used in practice to minimize code duplication by factoring shared code into base classes that are inherited by derived classes with more specific behaviors. The other type of relationship is *composition*. Class A *is composed*

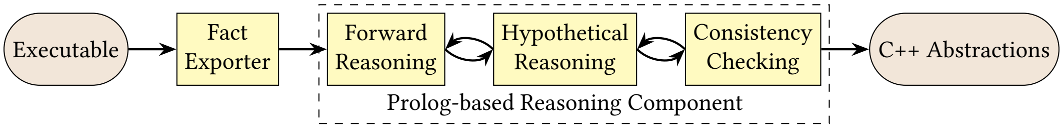

Figure 1: The system-level design of OOAnalyzer. OOAnalyzer exports low-level facts for an executable using a lightweight symbolic analysis. Prolog-based reasoning then deduces new facts, makes hypothetical assertions (*e.g.,* guesses) to address ambiguous properties, and validates the consistency of the C++ abstractions model. The final model is provided to the user when it is consistent and no proposed guesses remain.

of class B when class A stores an object of class B as a data member. Class A can access class B's public functionality through the object's methods and members.

2.3 Method Implementation and `thiscall`

C++ methods are implemented as special functions that reserve a parameter for the object on which the method is being invoked [11]. Most C++ methods are implemented using the `thiscall` calling convention, which passes the object pointer in the `ecx` register. This is important because such methods can be detected by observing that a function appears to access an object through the `ecx` register.

2.4 Runtime Type Identification

RTTI is optional metadata that is used to implement C++ type introspection features such as `dynamic_cast` and `typeid`. RTTI data structures include information such as each class's name and base classes. Only polymorphic classes have RTTI records and malware authors sometimes disable RTTI during compilation if the program does not utilize the introspection features. Some related work relies on RTTI as its primary source of information [33].

3 DESIGN

OOAnalyzer's design is largely motivated by the observation that human analysts often reverse engineer complex programs *incrementally* by combining logic, domain knowledge, and intuition [23, 27]. Specifically, analysts often start by developing a rough mental model of the program by "skimming" for common patterns that indicate or suggest specific C++ abstractions. Some patterns are strong enough that the analyst may immediately conclude a new fact about the program. In other cases, the pattern only suggests that the program *may* have a particular property, in which case the analyst often makes an *educated guess* that the program has the property. After making an educated guess, if the analyst later encounters conflicting evidence, she revises her guess. As the analyst observes and concludes more facts about the program that do not contradict each other, she becomes more confident in her knowledge about the program.

3.1 Design Goals and Motivations

3.1.1 Support for Non-polymorphic Classes. Most existing work on recovering C++ abstractions relies on the presence of virtual function tables, and as a result can only recover information about polymorphic classes [6–10, 15, 19, 33]. We designed OOAnalyzer so that it can reason about *all* classes and methods. As a consequence, OOAnalyzer cannot use vftables as class identifiers since not all classes will have them. Instead, it represents classes as the set of methods defined on the class. Another important consequence is that even though vftables provide valuable evidence about method-to-class assignments for virtual functions [10], OOAnalyzer cannot rely solely on this source of evidence. Instead, OOAnalyzer primarily assigns methods to classes by observing method calls on object pointers as they flow throughout the program. When a group of methods is invoked on the same object pointer, those methods must be defined on that object's class or one of its base classes.

3.1.2 Logic Programming to Resolve Ambiguity. Some C++ properties are ambiguous at the executable level, which makes educated guessing an important part of recovering C++ abstractions. Ambiguous properties occur because programs with distinct C++ abstractions can have equivalent run-time (*i.e.,* executable) semantics and thus can result in identical executables. To allow OOAnalyzer to make and recover from educated guesses, OOAnalyzer features Prolog prominently in its design. Prolog is used both as a mechanism for succinctly encoding the rules that comprise OOAnalyzer's reasoning process and as a strategy to search for a consistent model of the program. OOAnalyzer also takes full advantage of Prolog's backtracking capabilities, which allows it to cope with faulty assumptions and guesses. Whenever an inconsistency in reasoning is detected, Prolog allows OOAnalyzer to backtrack or "rewind" any reasoning performed since the last guess that was made. We show in Section 6.4 that without the ability to make and recover from educated guesses, OOAnalyzer's average error rate balloons significantly (from 21.8% to 81%).

3.2 Design Overview

OOAnalyzer formalizes and automates the incremental reasoning approach by combining a lightweight static symbolic binary analysis with a flexible Prolog-based reasoning framework. As can be seen in Fig. 1, OOAnalyzer takes an executable program as input, and first extracts low-level facts that form the basis of reasoning. It then deduces new facts that are implied by the current facts using forward reasoning rules until it can reach no new conclusions. It then identifies an ambiguous property for which a direct deduction is not possible, and hypothetically asserts, or *guesses*, a fact about that property. After asserting the fact, it deduces the consequences of the guess by returning to forward reasoning. When it can reach no new conclusions, it finally validates the consistency of the C++ abstractions model. If the model is inconsistent, OOAnalyzer systematically revisits the guesses that it made through hypothetical reasoning, starting with the most recent one. When the current model is internally consistent and no proposed guesses remain, OOAnalyzer outputs the discovered model for the user.

3.2.1 Executable Fact Exporter. The executable fact exporter is responsible for performing the "traditional" binary analysis steps of disassembling and lifting assembly instructions to a semantic representation, partitioning the instructions into separate functions, and conducting semantic analysis. There are many ways to perform semantic analysis, and in the interest of scalability, OOAnalyzer uses a lightweight symbolic analysis. OOAnalyzer also makes a number of simplifying assumptions that are characteristic of executables emitted by a reasonable compiler [11].

The facts generated by the fact exporter are called *initial* facts. They generally describe low-level program behaviors, such as calling a method on an object pointer, and these behaviors form the foundation on which all other conclusions in the system are based. These facts are approximations, and most have one sided errors. As a result, most of OOAnalyzer's rules assume that *initial* facts are "low confidence." Informally, this means that they need to be validated or corroborated by other facts before they should be utilized, since they could be wrong![1] All *initial* facts are static, meaning that they are not modified during the later reasoning stages of OOAnalyzer.

3.2.2 Forward Reasoning. OOAnalyzer reasons about the program by matching a built-in set of rules over facts in the fact base. Each reasoning rule is an inference rule that has one or more preconditions and a conclusion. If all of the preconditions are satisfied by the fact base, then the conclusion is added to the fact base. Initially, the fact base consists of only the *initial* facts that are emitted by the executable fact exporter. As reasoning proceeds, more facts are added by forward reasoning and hypothetical reasoning.

The facts emitted by forward and hypothetical reasoning are called *entity* facts. Unlike *initial* facts, which typically describe a property of executable semantics, *entity* facts describe an aspect of the C++ abstractions that our system is attempting to recover, and intermediate conclusions about those properties. *Entity* facts are dynamically asserted and retracted as the model of the program evolves during the reasoning process.

3.2.3 Hypothetical Reasoning. Sometimes OOAnalyzer is unable to reach new forward reasoning conclusions before important properties about the program are resolved. To continue making progress in these scenarios, OOAnalyzer identifies an ambiguous property and makes an educated guess about it, which we call *hypothetical reasoning*. OOAnalyzer has hypothetical reasoning rules that function similarly to forward reasoning rules, but instead describe the ambiguous situations in which OOAnalyzer should make its guesses. The analysis of the program is complete only when all ambiguous properties have been resolved. Since hypothetical rules only provide educated guesses for ambiguous properties, it is possible for an incorrect guess to introduce inconsistencies in the model of the program. As a result, the model must pass consistency checks before the resulting entity fact is accepted.

3.2.4 Consistency Checking. When OOAnalyzer detects an inconsistency in the current fact base it *backtracks* and systematically revisits the earlier guesses that have been made, starting with the most recent one. Consistency checks are implemented by a special set of rules that detect contradictions instead of asserting new facts.

Conceptually, consistency rules could be implemented as constraints that block forward reasoning and hypothetical rules from making inconsistent conclusions. But this design would not allow OOAnalyzer to backtrack and correct the root cause of the problem (*i.e.,* a bad guess), which may have occurred much earlier. By separating our consistency rules and forcing OOAnalyzer to backtrack when they are violated, it allows OOAnalyzer to utilize the conclusions of forward reasoning but revert them when they lead to an inconsistent state.

4 REASONING SYSTEM

4.1 Symbolic Analysis

OOAnalyzer's fact exporter employs a lightweight symbolic analysis that is provided as one of the features of the Pharos binary analysis framework (Section 5). Pharos's symbolic analysis attempts to represent the final values of registers and memory at the end of a function as symbolic expressions in terms of the function's symbolic inputs. For example, if a function increments `eax` and the initial symbolic value of `eax` is represented as `eax_init`, the output state for `eax` would be `eax_init` + 1. Each function's symbolic summary is computed using a lightweight, intra-procedural, path- and flow-sensitive data-flow algorithm. (*Inter*-procedural reasoning occurs later in the Prolog part of the system; see Section 4.2 for details.) OOAnalyzer also uses auxiliary analyses in Pharos that track the propagation of object pointers and identify calling conventions (Section 4.2).

OOAnalyzer's symbolic analysis is designed to be lightweight and scalable, and as a result, differs from conventional binary symbolic analysis [4, 5, 25, 26] in many ways. First, OOAnalyzer does not use SMT constraint solvers to reason about whether a particular program execution is feasible. Instead, OOAnalyzer assumes all execution paths are feasible. OOAnalyzer reasons about each path separately (*i.e.,* it is path-sensitive), but prevents exponential path explosion [25] by only unrolling loops for five iterations and setting thresholds on the maximum size of symbolic expressions. The memory model decides if two symbolic memory addresses alias by seeing if the symbolic memory address expressions are equal after applying simplification and normalization rules.

Despite its simplicity, OOAnalyzer's symbolic analysis performs well for two reasons. First, most initial facts describe compiler written code that manipulates entities such as object pointers and virtual function tables, and such code seldom employs complicated loops, branches, or memory dereferences that are a bane to more general static binary analysis. Second, even when the symbolic analysis does make a noticeable mistake, the later components of OOAnalyzer can usually detect and recover from it.

4.2 Initial Facts

As we explained in Section 3.2.1, *initial facts* are emitted by the fact exporter and generally describe low-level program behaviors such as computing an offset into the current method's object or calling a method using an object pointer. Table 1 provides a brief summary of selected *initial* facts and examples of assembly code patterns that would produce them.

[1] "Wrong fact" is obviously an oxymoron, but we use *fact* to denote a piece of evidence, rather than something that is indisputable. In other words, we use fact synonymously with *belief.*

Predicate Name	Description	Assembly Code Example

ObjPtrAllocation(I, F, P, S)	Instruction I in function F allocates S bytes of memory for the object pointed to by P.	`push 28h`
		`call operator new`

ObjPtrInvoke(I, F, P, M)	Instruction I in function F calls method M on the object pointed to by P.	`mov ecx, objptr`
		`call M`

ObjPtrOffset(P_1, O, P_2)	Object pointer P_2 points to P_1 + O. This usually indicates the presence of object composition.	`mov ecx, objptr`
		`add ecx, 10h`

MemberAccess(I, M, O, S)	Instruction I in method M accesses S bytes of memory at offset O from the object pointer. This generally indicates the size and offset of a member.	`mov ecx, objptr`
		`mov ebx, [ecx+0ch]`

ThisCallMethod(M, P)	Method M receives the object pointed to by P in the `ecx` register, which indicates the method expects to be called with `thiscall`.	Not applicable.
NoCallsBefore(M)	No methods are called on an object pointer before method M, which is often indicative of constructors.	Not applicable.

ReturnsSelf(M)	Method M returns the object pointer that was passed as a parameter. This code pattern is required for constructors.	`mov eax, objptr`
		`retn`

UninitializedReads(M)	Method M reads members before writing to them, which is not typical of constructors.	Not applicable.
PossibleVFTableEntry(VFT, O, M)	It is possible that method M is at offset O in vftable VFT.	Not applicable.

Table 1: A list of selected *initial* fact predicates produced by OOAnalyzer's fact exporter. Initial facts form the basis upon which OOAnalyzer's reasoning system operates.

One of the most important categories of initial facts describes the creation, manipulation, and usage of object pointers. These facts enable OOAnalyzer to reason about relationships between classes and methods without relying on RTTI or vftables, unlike most prior work. Using its symbolic analysis, OOAnalyzer assigns a unique token to each object pointer that appears to be passed to a function using the `thiscall` calling convention, and then records when such pointers are allocated (ObjPtrAllocation), invoked on a method (ObjPtrInvoke), or created at an offset from an existing object pointer (ObjPtrOffset). The last fact often reveals an object instance being stored as a class member (*i.e.,* composition). Finally, to enable *inter*-procedural reasoning about object pointers, the ThisCallMethod fact links methods to the symbolic object pointers they are invoked on.

Another important group of initial facts are those used to activate *hypothetical* reasoning rules, which in turn produce high-confidence *entity* facts. The group of initial facts that are used to identify constructors is a good example. If a method is always the first to be called on an object (NoCallsBefore), returns the object pointer that is passed to it (ReturnsSelf), and does not read from any data members in the object (UninitializedReads), then it is likely to be a constructor, and hypothetical reasoning will use these facts to assert a Constructor fact and hypothetically reason about the consequences. Most entity facts have a corresponding possible initial fact that triggers hypothetical reasoning about that entity, which is discussed further in the next section. For space reasons, we do not include all initial facts in Table 1, but they can be found in the OOAnalyzer source repository [18].

4.3 Entity Facts

As mentioned in Section 3.2.2, *entity* facts describe properties of the abstract entities such as methods, virtual function tables, and classes that comprise the C++ abstractions which OOAnalyzer recovers. Entity facts can be roughly organized by the type of entity they describe, which includes (1) methods; (2) virtual function tables and virtual base tables; (3) class relationships; (4) sizes; and (5) classes. Table 2 displays the list of selected *entity* facts with this ordering.

Most entity facts have at least one corresponding *initial* fact that triggers reasoning about that entity. For example, the fact exporter identifies possible vftables in memory by scanning for adjacent entries that could plausibly be code addresses and emits these as low-confidence *initial* PossibleVFTableEntry facts. If reasoning rules corroborate their existence in the current model, OOAnalyzer dynamically asserts *entity* facts such as VFTableEntry to confirm the existence and contents of the table. This two-tier reasoning is used for many of the entity facts.

Class relationships are described by several facts. The DerivedClass fact reflects that a class inherits from another class, while the ComposedObject fact indicates composition. Because inheritance and composition often look similar at the executable level, OOAnalyzer also uses an intermediate fact, ObjectInObject, which is true when DerivedClass or ComposedObject is true, but not both (*i.e.,* DerivedClass⊕ComposedObject). Finally, the fact HasNoBase explicitly expresses that a class does *not* inherit from another class. Some rules are able to prove the existence of a base class without actually identifying the specific class, which is expressed as ¬HasNoBase.

Size facts bound the potential sizes of classes and vftables. Constraints on the sizes of classes (ClassSize) are obtained from allocations and member composition, and can be used to disprove certain inheritance relationships based on the observation that a smaller class cannot be derived from a larger class. The size of vftables (VFTableSize) can also be bounded. For example, a vftable cannot be so large that it overlaps with another known vftable, and a derived class's vftable cannot be smaller than its base class's vftable.

OOAnalyzer represents classes as sets of methods to allow it to reason about non-polymorphic classes, which do not have a

Predicate Name	Description
Method(M)	Method M is an OO method on a class or struct. It is passed an object pointer.
Constructor(M)	Method M is an object constructor. It initializes objects, but does not allocate memory for the object.
Destructor(M)	Method M is an object destructor. It deinitializes objects, but does not free their memory.
DeletingDestructor(M)	Method M is a deleting destructor. It calls a "real" destructor before deallocating the object's memory.
VirtualMethodCall(I, F, P, VFT, O)	Instruction I in function F virtually invokes the method at offset O of the vftable VFT on pointer P.
VFTable(VFT)	VFT is a virtual function table. (There are similar rules for virtual base tables.)
VFTableInstall(I, M, O, VFT)	Instruction I in method M installs vftable VFT at offset O of the current object.
VFTableEntry(VFT, O, M)	Offset O in vftable VFT contains a pointer to method M.
DerivedClass(Cl$_d$, Cl$_b$, O)	Class Cl$_d$ inherits from class Cl$_b$. The members of Cl$_b$ are stored at offset O of Cl$_d$.
ComposedObject(Cl$_o$, Cl$_i$, O)	Class Cl$_o$ is composed of an object of class Cl$_i$ at offset O.
ObjectInObject(Cl$_o$, Cl$_i$, O)	Either DerivedClass(Cl$_o$, Cl$_i$, O) or ComposedObject(Cl$_o$, Cl$_i$, O) is true, but not both.
HasNoBase(Cl)	Class Cl is known not to inherit from any base classes.
ClassSize(Cl)	This function returns the size in bytes of instantiated Cl objects.
VFTableSize(VFT)	This function returns the size in bytes of vftable VFT.
Cl$_a$ = Cl$_b$	The sets of methods, Cl$_a$ and Cl$_b$, both represent methods from the same class. This predicate indicates the sets of methods should be combined into a single class.
Cl$_a$ ≤ Cl$_b$	The sets of methods, Cl$_a$ and Cl$_b$, either both represent methods from the same class, *or*, the methods in Cl$_b$ are (possibly indirectly) inherited from Cl$_a$.
M ∈ Cl	The method M is defined directly on class Cl. (It is a member of the set of methods defining the class Cl.)
ClassCallsMethod(Cl, M)	An instance of class Cl calls method M, indicating M is on class Cl or one of its ancestors.

Table 2: A list of selected *entity* fact predicates that are produced by the forward reasoning and hypothetical reasoning capabilities of OOAnalyzer.

natural identifier for the class such as a vftable address. Initially, each method is considered its own singleton class, but is eventually merged with other classes using class merging facts such as Cl$_a$ = Cl$_b$, which indicates that two previously distinct classes are really the same class. When Cl$_a$ is merged with Cl$_b$, any existing facts about Cl$_b$ are updated to reference Cl$_a$ instead. The ClassCallsMethod fact provides evidence that M is called on an object of class Cl, which indicates that M must be directly defined on Cl or one of its ancestors. This in turn helps hypothetical reasoning compute the candidate classes which M could be assigned to. Because the fact is based on the data flow of object pointers rather than vftables, it provides another example of how OOAnalyzer is able to assign methods to non-polymorphic classes.

4.4 Reasoning Rules

As the primary mechanism for encoding the domain knowledge about C++ programming and compilers, OOAnalyzer's forward reasoning component is one of the most important pieces of OOAnalyzer's design. In this section, we present OOAnalyzer's reasoning rules as inference rules. Inference rules have the following form:

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

where P_i represents the ith premise of the rule, and C represents the conclusion. If all premises in the rule are present in the current fact base, OOAnalyzer adds the conclusion to the fact base as well. Fig. 2 shows a selection of rules which we also discuss below. Unfortunately, we are unable to reproduce all the reasoning rules in this paper due to space limitations, but they can be found in the OOAnalyzer source repository [18].

One salient feature of OOAnalyzer is that it can reason about non-polymorphic classes, which do not have any associated vftable. For example, Merge-6 shows the formal encoding of a rule which OOAnalyzer uses to determine that a method called by a base class and its derived class cannot be defined on the derived class. This is one of many rules in OOAnalyzer that does not rely on vftables at all, and instead is based on observing actions on object pointers.

Although OOAnalyzer does not *depend* on the information in vftables, it can leverage their information to learn about polymorphic classes, similar to existing work. For example, Relate-3 shows a rule that detects inheritance between classes by observing a constructor replace a vftable that another constructor installed in an object. Relate-6 is a slightly more complicated encoding of the observation that when a method is present in two vftables, the two classes must be related in some way. Because of the ambiguous nature of this observation, which does not identify the exact relationship (*e.g.*, a sibling or parent-child relationship), the conclusion is the negation of HasNoBase(Cl$_b$),[2] or that Cl$_b$ inherits from an

[2] An astute reader may recognize that Prolog does not allow rules to contain negation in their conclusion. However, for most facts in OOAnalyzer, it is important to explicitly represent whether that fact is definitely true, definitely false, or unknown in the model. Thus, most facts in OOAnalyzer are actually represented by both a positive and negative predicate for the fact (*e.g.*, HasNoBase-True and HasNoBase-False). Such predicates are kept consistent by OOAnalyzer's consistency checking rules (Section 4.6) and boilerplate code.

Merge-6

$$\frac{\begin{array}{c}\text{Constructor}(\text{M}_d) \qquad \text{M}_d \in \text{Cl}_d \\ \text{Constructor}(\text{M}_b) \qquad \text{M}_b \in \text{Cl}_b \\ \text{ClassCallsMethod}(\text{Cl}_d, \text{M}) \\ \text{ClassCallsMethod}(\text{Cl}_b, \text{M}) \qquad \text{M}_d \neq \text{M}_b \\ \text{M} \in \text{Cl}_m \qquad \text{Cl}_d \neq \text{Cl}_b \qquad \text{DerivedClass}(\text{Cl}_d, \text{Cl}_b, \_)\end{array}}{\text{Cl}_m \neq \text{Cl}_d}$$

Relate-3

$$\frac{\begin{array}{c}\text{Constructor}(\text{M}_d) \qquad \text{VFTableInstall}(\_, \text{M}_d, \_, \text{VFT}_d) \\ \text{Constructor}(\text{M}_b) \qquad \text{VFTableInstall}(\_, \text{M}_b, 0, \text{VFT}_b) \\ \text{ThisCallMethod}(\text{M}_d, \text{P}_d) \\ \text{ObjPtrOffset}(\text{P}_d, \text{O}, \text{P}_b) \qquad \text{ObjPtrInvoke}(\_, \text{M}_d, \text{P}_b, \text{M}_b) \\ \text{M}_d \neq \text{M}_b \qquad \text{VFT}_d \neq \text{VFT}_b \qquad \text{M}_d \in \text{Cl}_d \qquad \text{M}_b \in \text{Cl}_b\end{array}}{\text{DerivedClass}(\text{Cl}_d, \text{Cl}_b, \text{O})}$$

Relate-6

$$\frac{\begin{array}{c}\text{VFTableEntry}(\text{VFT}_a, \_, \text{M}) \qquad \text{VFTableInstall}(\_, \text{M}_a, 0, \text{VFT}_a) \\ \text{VFTableEntry}(\text{VFT}_b, \_, \text{M}) \qquad \text{VFTableInstall}(\_, \text{M}_b, \_, \text{VFT}_b) \\ \text{VFT}_a \neq \text{VFT}_b \qquad \text{Constructor}(\text{M}_a) \qquad \text{Constructor}(\text{M}_b) \\ \text{M}_a \in \text{Cl}_a \qquad \text{M}_b \in \text{Cl}_b \qquad \text{M} \in \text{Cl}_a \qquad \text{Cl}_a \neq \text{Cl}_b\end{array}}{\neg\text{HasNoBase}(\text{Cl}_b)}$$

Merge-17

$$\frac{\begin{array}{c}\text{VFTableInstall}(\_, \text{M}_d, 0, \text{VFT}_d) \qquad \text{M}_d \in \text{Cl}_d \\ \text{VFTableInstall}(\_, \text{M}_b, 0, \text{VFT}_b) \qquad \text{M}_b \in \text{Cl}_b \\ \text{DerivedClass}(\text{Cl}_d, \text{Cl}_b, \_) \\ \text{VFTableSize}(\text{VFT}_b) \leq \text{Size} \qquad \text{VFTableEntry}(\text{VFT}_d, \text{O}, \text{M}) \\ \text{M} \in \text{Cl} \qquad \text{Cl} \neq \text{Cl}_d \qquad \text{O} > \text{Size}\end{array}}{\text{Cl}_d = \text{Cl}}$$

Figure 2: Selected forward reasoning rules

unspecified base class. Although this conclusion by itself is vague, if there is no direct evidence that determines the inheritance relationship, it will eventually trigger hypothetical reasoning to find a relationship that is consistent with all observed facts.

Another rule, Merge-17, demonstrates how bounding the size of classes and vftables can help assign methods to classes [10]. When a method appears in the vftable of a derived class, that method could be defined in either the derived or base class. Merge-17 shows how OOAnalyzer can decisively place the method on the derived class by bounding the size of the base class's vftable, and noting that the method's offset in the vftable is too large for the method to be on the base class.

4.5 Hypothetical Reasoning Rules

Hypothetical reasoning rules are identical to forward reasoning rules in structure, but are interpreted differently. First, hypothetical rules are only used when forward reasoning rules are unable to produce any new conclusions. Second, most hypothetical reasoning rules occur in pairs that share the same premises but contain opposite conclusions. As with standard rules, if all of the premises match the current fact base, the conclusion of the rule is added to the fact base, and reasoning proceeds. If Prolog backtracks to this point because of a contradiction, OOAnalyzer will revoke this fact and instead assert the opposite conclusion. If this too results in a contradiction, the inconsistency must have come from an earlier hypothesis, and OOAnalyzer backtracks even further.

Guess6-T

$$\frac{\begin{array}{c}\text{ClassCallsMethod}(\text{Cl}_d, \text{M}) \qquad \neg\text{ClassCallsMethod}(\text{Cl}_b, \text{M}) \\ \text{M} \in \text{Cl} \qquad \text{DerivedClass}(\text{Cl}_d, \text{Cl}_b, \_)\end{array}}{\text{Cl}_d = \text{Cl}}$$

Guess1-T

$$\frac{\text{ObjectInObject}(\text{Cl}_d, \text{Cl}_b, \text{O})}{\text{DerivedClass}(\text{Cl}_d, \text{Cl}_b, \text{O})}$$

Guess4-T

$$\frac{\begin{array}{c}\text{Constructor}(\text{M}) \qquad \text{M} \in \text{Cl} \qquad \text{VFTableInstall}(\_, \text{M}, 0, \text{VFT}_a) \\ \neg\exists\text{VFT}_b.\ \text{VFTableInstall}(\_, \text{M}, \_, \text{VFT}_b) \land \text{VFT}_a \neq \text{VFT}_b\end{array}}{\text{HasNoBase}(\text{Cl})}$$

Guess3-T

$$\frac{\begin{array}{c}\text{Method}(\text{M}) \qquad \text{ReturnsSelf}(\text{M}) \\ \text{NoCallsBefore}(\text{M}) \qquad \neg\text{PossibleVFTableEntry}(\_, \_, \text{M}) \\ \text{VFTableInstall}(\_, \text{M}, \_, \_) \qquad \neg\text{UninitializedReads}(\text{M})\end{array}}{\text{Constructor}(\text{M})}$$

Figure 3: Selected hypothetical reasoning rules

We ordered the hypothetical rules in OOAnalyzer so that the most likely guesses are made first in order to minimize backtracking. Some of the highest priority hypothetical rules in OOAnalyzer were originally forward reasoning rules (because we believed them to be always true), but we later identified rare exceptions that were difficult to characterize, and so we converted them to hypothetical rules. We ordered other rules based on a combination of experimentation and our beliefs about how prevalent the phenomena are. For example, we prioritize rules related to single inheritance before multiple inheritance, because in our experience multiple inheritance is less common. A few guessing rules are also ordered *by design* to consist of progressively more relaxed constraints. The idea behind these rules is to choose entities that have the most evidence associated with them first. This is especially important for entities that consistency checks struggle to reject. Two examples of this are discussed below in rules Guess4-T and Guess3-T.

One of the most important hypothetical reasoning rules, Guess6-T, is shown in Fig. 3. This rule handles one of the most common ambiguities in method assignment, which occurs when a method is called on a derived class but not a base class. It is possible that the method is defined on the base class but is never invoked there; or, the method may be defined directly on the derived class. As Guess6-T shows, OOAnalyzer initially guesses that the method is on the derived class. If that results in a contradiction, Guess6-F (not shown) instead guesses that the method is *not* on the derived class.[3] Guess6-T is also another example of a rule that does not depend on vftables, since ClassCallsMethod is based on data-flow of object pointers.

[3] Note that the method is not necessarily on the base class since it could be on one of the base class's ancestors.

Another common ambiguity when reasoning is whether a relationship between two classes represents an inheritance relationship or a composition relationship (Section 2.2). As Guess1-T shows, in the absence of other information, OOAnalyzer will initially guess that an unspecified relationship is an inheritance relationship. If this is proven untrue by consistency checking rules, then OOAnalyzer will instead guess that there is a class composition relationship (*i.e.,* a class object as a class member).

Rule Guess4-T illustrates the importance of the *priority* in which hypothetical reasoning rules are applied. Guess4-T hypothetically reasons that a class has no base class based on the very weak precondition that the class's constructor installs a single vftable. While the HasNoBase fact is critically important for reasoning about method to class assignment, the best evidence that a class does not have a base class is often a lack of evidence for inheritance. This rule exemplifies such reasoning, because any evidence that inheritance might exist would immediately preclude this general rule. This rule also exemplifies progressively more general guessing rules. OOAnalyzer guesses that classes with one vftable installation are candidates for HasNoBase (*e.g.,* Guess4-T) before evaluating an even weaker rule that guesses all remaining classes without proven base classes actually have no base class at all (not shown). Since the only forward reasoning rules for HasNoBase are based on RTTI data structures, and we do not assume these are always available, Guess4-T has an unexpectedly important role in OOAnalyzer for such a weak rule.

Another important use of hypothetical reasoning is to detect special methods such as constructors. For example, Guess3-T is one of the highest priority rules used to guess that a method is a constructor because it requires relatively strong heuristics. The method must not be present in any vftables, may not read any uninitialized memory, and must install a vftable in its object. We have found these heuristics usually indicate that a method is a constructor. A series of lower priority guessing rules then relaxes these constraints.

4.6 Consistency Checking Rules

Consistency rules ensure that all facts in the fact base are internally consistent with each other. Fig. 4 lists selected consistency rules. For space reasons, we only list a few consistency rules, but the complete set can be found in the OOAnalyzer source repository [18]. Most of the interesting consistency rules ensure a variety of C++-specific invariants are true. For example, Consistency-VFTables ensures that a VFTable cannot be assigned to two unrelated classes. Another example is Consistency-MultipleRealDestructors, which checks that each class has at most one real destructor. When consistency checks fail, they conclude false, which forces Prolog to backtrack and revisit guesses made during hypothetical reasoning. OOAnalyzer completes and presents the results to the user after all proposed guesses have been made and the model passes the consistency checks.

5 IMPLEMENTATION

OOAnalyzer's executable fact exporter is a tool built inside the Pharos binary analysis framework [20]. Pharos is developed by Carnegie Mellon's Software Engineering Institute and builds upon the binary analysis components of the ROSE compiler infrastructure [22] from Lawrence Livermore National Lab. At a high level, ROSE handles Portable Executable (PE) file format parsing, instruction disassembly and the partitioning of those instructions into functions. ROSE also provides instruction semantics and a static analysis framework. Pharos builds on these capabilities by adding a lightweight symbolic analysis that summarizes the output of each function in terms of its symbolic inputs. OOAnalyzer utilizes this symbolic analysis to generate initial facts. OOAnalyzer consists of approximately 2,313 lines of C++ code inside the Pharos framework. Pharos itself has 51,222 lines of C++. Most of OOAnalyzer's C++ code implements the executable fact exporter, but there is also a user front-end and an interface with the Prolog engine.

Consistency-VFTables

$$\frac{\begin{array}{c}\mathsf{VFTableInstall}(\_, \mathsf{M}_a, \mathsf{O}, \mathsf{VFT}) \\ \mathsf{VFTableInstall}(\_, \mathsf{M}_b, \mathsf{O}, \mathsf{VFT}) \\ \mathsf{M}_a \neq \mathsf{M}_b \qquad \mathsf{M}_a \in \mathsf{Cl}_a \qquad \mathsf{M}_b \in \mathsf{Cl}_b \qquad \mathsf{Cl}_a \neq \mathsf{Cl}_b \\ \mathsf{HasNoBase}(\mathsf{Cl}_a) \qquad \mathsf{HasNoBase}(\mathsf{Cl}_b)\end{array}}{\mathsf{false}}$$

Consistency-VirtualConstructor

$$\frac{\mathsf{Constructor}(\mathsf{M}) \qquad \mathsf{VFTableEntry}(\mathsf{VFT}, \_, \mathsf{M})}{\mathsf{false}}$$

Consistency-DoubleDuty

$$\frac{\mathsf{Constructor}(\mathsf{M}) \qquad \mathsf{Destructor}(\mathsf{M}) \vee \mathsf{DeletingDestructor}(\mathsf{M})}{\mathsf{false}}$$

Consistency-MultipleRealDestructors

$$\frac{\mathsf{Destructor}(\mathsf{M}_a) \qquad \mathsf{Destructor}(\mathsf{M}_b) \qquad \mathsf{M}_a \neq \mathsf{M}_b}{\mathsf{false}}$$

Figure 4: Selected consistency rules

OOAnalyzer's Prolog implementation consists of approximately 4,996 lines of Prolog rules. OOAnalyzer employs XSB Prolog [29] because it is mature, open source, can be embedded into C/C++ programs, and has robust tabling support. From the perspective of a Prolog programmer, tabling is essentially a mechanism for caching the execution of Prolog rules. We quickly found that tabling support is a practical requirement due to the large number of facts that can be emitted for programs, and the nature of OOAnalyzer's hypothetical reasoning strategy, which results in repetitive queries being issued under slightly different contexts. Tabling allows these repetitive queries to be made much more efficiently. The current OOAnalyzer implementation can analyze 32-bit Windows PE executables, and we are working to add support for 64-bit executables. We chose to focus on Windows because it is the platform most commonly targeted by malware and other closed source C++ programs.

6 EVALUATION

In this section, we evaluate OOAnalyzer's ability to identify C++ classes and their constituent methods (Section 6.4), and to classify methods as constructors, destructors, and virtual methods (Section 6.5). Along the way, we discuss how we produce ground truth data (Section 6.2) for our program corpus (Section 6.1), and use that ground truth data to develop a new class membership metric based on edit distance (Section 6.3).

6.1 Program Corpus

We evaluated OOAnalyzer on a program corpus of 27 programs that were compiled to 32-bit Windows PE executables. We chose our corpus to reflect the two most common scenarios in which recovering C++ abstractions is necessary: analyzing cleanware and malware.

Cleanware. The top eighteen rows of Table 3 list the cleanware programs in our corpus. We started by selecting Windows cleanware evaluated in other C++ abstraction recovery work [14], namely: CImg 1.05, Light POP SMTP 608b, optionparser 1.3, PicoHttpD 1.2, and x3c 1.02. We used the same executables from that work [14], which were compiled with Visual Studio 2010 using the Debug configuration (i.e., optimizations disabled). For this paper, we added log4cpp 1.1, muParser 2.2.3 and TinyXML 2.6.1, which we compiled ourselves using Visual Studio 2010 in both Debug and Release configurations to analyze the impact of optimizations. We felt that these programs adequately covered small and medium sized programs since they range in size from 42 to 663 KiB, but did not represent larger cleanware programs. To represent more complex programs, we included Firefox web browser 52.0,[4] and several programs from MySQL database 5.2.0, including mysql.exe, which is larger than 5 MiB, and is a strenuous test of OOAnalyzer's ability to handle large and complex programs. We used the official precompiled 32-bit Windows executables for these programs, which were compiled by Visual Studio 12 and 15 respectively, both with optimizations enabled.

Malware. The nine malware programs in our corpus are shown in the bottom rows of Table 3. Malware is one of the most common reverse engineering targets, but evaluating it is difficult because ground truth is seldom available. We addressed this problem by searching our private malware collection of hundreds of millions of samples for executables that have corresponding Program Database (PDB) symbol files [16]. Such pairs can be identified because the Visual Studio linker embeds the debug GUID, which uniquely identifies a PDB file, in the corresponding executable. We believe that most of these debugging symbols were collected after a malicious actor inadvertently copied them to a target system along with the malware. To ensure that all of the malware files in our corpus are actually malicious, we only considered files on which at least one antivirus product in VirusTotal reported a detection, and then manually verified that each file was malicious. From their PDB files, we were able to determine that all the malware samples were compiled with Visual Studio 9, 10, or 11, and only one sample was compiled with optimizations enabled.

6.2 Ground Truth

We produced ground truth C++ abstractions for each program by parsing the PDB [16] files that are optionally produced by Visual C++ during compilation. We only used these debugging symbols to evaluate our results, and did not provide them to OOAnalyzer. The ground truth for each program includes the list of classes in the program, the methods and members in each class, and the location of each class's virtual function and base tables if applicable.

6.2.1 Scope. Our evaluation considers any method or object whose *implementation* is in the executable to be in scope. In C++ programs, this often includes some library code, because the implementations of methods from templated library classes will be included in the executable *even when employing dynamic linking*. Programs which utilize heavily templated libraries such as the Standard Template Library and Boost libraries [3] can have substantial amounts of library code inside of them. Unfortunately, since these library methods cannot be easily distinguished from application code, their presence increases the difficulty of understanding the given executable. Thus, it is important for tools such as OOAnalyzer, which attempt to ease this burden, to consider them in scope.

6.2.2 Ground Truth Exceptions. Some differences between OOAnalyzer's output and the ground truth are minor differences that are both uninteresting (*i.e.,* an analyst does not care about the distinction) and indistinguishable at the executable level (*i.e.,* the distinction only makes sense in source code). Thus, we adjust the following special cases in the ground truth:

- When a class method does not use its object, it can be indistinguishable from a regular function. In particular, when a function is called in a 32-bit binary, it can be ambiguous whether the compiler placed the object pointer in the `ecx` register explicitly to call a OO method using `thiscall` (Section 2.3), or if the compiler happened to leave the object pointer in `ecx` when it called a regular function. When the OO method does not access `ecx`, it is ambiguous whether `ecx` was actually a parameter. Because of this ambiguity, we treat each method that does not use its object as a regular, non-OO function.
- When a class method is linked into the executable but not actually invoked in the control flow of the program, it is often impossible to determine which class it belongs to. We identify methods with no code or data references using a Hex-Rays IDA Python script [12] and exclude those methods from the ground truth. This scenario occurs more frequently than might be expected because the linker includes entire object modules without removing unused functions unless Whole Program Optimization (Section 7.1) is enabled.

6.3 Edit Distance as a Class Membership Metric

In the next section, we evaluate OOAnalyzer's ability to identify C++ classes and the methods in each class.[5] But first we motivate and introduce a new metric for quantifying the results. Recall from Section 3.1.1 that OOAnalyzer represents each class as a set of methods so that it can recover information about non-polymorphic

[4] Firefox notably consists of an executable and several large DLLs such as `xul.dll`. We only evaluated OOAnalyzer on `firefox.exe`. As we note in Section 7.3, this will include in scope any C++ classes that `firefox.exe` imports from `xul.dll`, but will not include classes internal to `xul.dll`. We believe this is what most people would want when reverse engineering a program as complex as Firefox. The other programs in our corpus do not contain significant amounts of code in DLLs.

[5] In this paper, we consider any object type with a method to be a class. This can include classes, structs, and oddly enough, unions.

Program	Ver.	Opt.	Com-piler	Size (KiB)	Num. Class	Num. Meth-ods	Method Edit Distance										
							w/o RTTI							with RTTI		w/o guess	
							Move	Add	Rem	Split	Join	Total	%	Total	%	Total	%
CImg	1.0.5		VS10	590	29	220	3	15	1	1	1	21	9.5	21	9.5	200	90.9
Firefox	52.0	✓	VS15	505	141	638	40	64	67	40	1	212	33.2	212	33.2	499	78.2
light-pop3-smtp	608b		VS10	132	44	295	15	15	0	12	2	44	14.9	41	13.9	263	89.2
log4cpp Debug	1.1		VS10	264	139	893	100	59	2	66	12	239	26.8	240	26.9	786	88.0
log4cpp Release	1.1	✓	VS10	97	76	378	27	15	2	24	7	75	19.8	75	19.8	244	64.6
muParser Debug	2.2.3		VS10	664	180	1437	213	111	17	104	38	483	33.6	474	33.0	1310	91.2
muParser Release	2.2.3	✓	VS10	302	94	598	59	55	8	34	27	183	30.6	181	30.3	407	68.1
MySQL cfg_editor.exe	5.2.0	✓	VS12	4,386	190	1266	200	63	3	68	57	391	30.9	388	30.6	1005	79.4
MySQL connection.dll	5.2.0		VS12	136	43	167	14	13	0	16	5	48	28.7	48	28.7	143	85.6
MySQL ha_example.dll	5.2.0	✓	VS12	54	21	256	16	8	0	4	4	32	12.5	32	12.5	211	82.4
MySQL libmysql.dll	5.2.0	✓	VS12	4,570	200	1328	197	65	3	75	66	406	30.6	399	30.0	1042	78.5
MySQL mysql.exe	5.2.0	✓	VS12	4,678	202	1395	229	69	4	74	63	439	31.5	433	31.0	1110	79.6
MySQL upgrade.exe	5.2.0	✓	VS12	5,321	333	2071	294	166	17	92	86	655	31.6	655	31.6	1578	76.2
optionparser	1.3		VS10	55	11	56	3	3	0	0	0	6	10.7	6	10.7	56	100.
PicoHttpD	1.2		VS10	386	95	656	54	58	10	24	20	166	25.3	161	24.5	569	86.7
TinyXML Debug	2.6.1		VS10	594	35	415	30	23	1	5	10	69	16.6	68	16.4	384	92.5
TinyXML Release	2.6.1	✓	VS10	222	33	283	19	9	6	10	11	55	19.4	56	19.8	229	80.9
x3c	1.0.2		VS10	42	6	28	1	4	0	0	0	5	17.9	5	17.9	28	100.
Malware 0faaa3d3	—		VS9	276	21	135	4	6	7	2	2	21	15.6	19	14.1	68	50.4
Malware 29be5a33	—		VS9	571	19	130	4	9	1	0	1	15	11.5	15	11.5	110	84.6
Malware 6098cb7c	—	✓	VS9	445	55	339	5	10	5	6	3	29	8.6	29	8.6	174	51.3
Malware 628053dc	—		VS10	1,322	207	1920	121	179	24	27	27	378	19.7	374	19.5	1724	89.8
Malware 67b9be3c	—		VS11	927	400	2072	280	159	89	111	31	670	32.3	670	32.3	1821	87.9
Malware cfa69fff	—		VS10	98	39	184	15	11	3	7	1	37	20.1	33	17.9	111	60.3
Malware d597bee8	—		VS10	68	19	133	4	8	0	3	2	17	12.8	15	11.3	91	68.4
Malware deb6a7a1	—		VS9	1,673	283	2712	264	281	38	19	37	639	23.6	639	23.6	2493	91.9
Malware f101c05e	—		VS9	1,256	169	1601	106	165	22	16	20	329	20.5	329	20.5	1453	90.8
Average					114	800							21.8		21.5		81.0

Table 3: The edit distance between the classes that OOAnalyzer recovered and the ground truth. A low edit distance indicates that the class assignments are close to the ground truth. The edit distance is broken down into the number of move, add, remove, split and join edits to reveal the types of errors that OOAnalyzer made when it was not allowed to use RTTI. These sum to the total edit distance which is also reported as a percentage of methods. The remaining columns show for comparison the results when OOAnalyzer is allowed to utilize RTTI and when OOAnalyzer's hypothetical reasoning component is disabled.

classes, which often do not have a natural identifier such as a virtual function table.

Unfortunately, without class identifiers that are present in both the ground truth and OOAnalyzer's output, it can be difficult to establish a mapping between the two. For example, if the ground truth contains two classes that consist of the method sets $\{M1, M2\}$ and $\{M3, M4\}$, and OOAnalyzer reports a single class consisting of $\{M1, M2, M3, M4\}$, how should that be judged? On the one hand, all four methods in the executable were identified, but on the other hand, OOAnalyzer accidentally merged two classes into one.

We propose that class membership should be evaluated by measuring the *edit distance* that is required to transform the classes that OOAnalyzer emits into the classes found in the ground truth. The edit distance is the number of actions used to perform the transformation, where the possible actions are:

1. *moving* a single method to another class;
2. *adding* a single method that OOAnalyzer failed to identify to an arbitrary class;
3. *removing* an extra function that OOAnalyzer mistakenly identified as a method;
4. arbitrarily *splitting* a class into two new classes; and
5. *merging* two separate classes into one.

For instance, in the above example, the class recovered by OOAnalyzer, $\{M1, M2, M3, M4\}$, must be *split* to yield $\{M1, M2\}$ and $\{M3, M4\}$, which yields an edit distance of one.

Edit distance can be interpreted as an upper bound on the number of mistakes that OOAnalyzer made. For example, if OOAnalyzer achieved an edit distance of 6 in a program with 56 methods (10.7%), it must have recovered at least $56 - 6 = 50$ methods (89.3%) correctly, since every incorrect method will require at least one corresponding edit to correct it.

Ideally, we would like to use the *minimal* edit distance as a metric. However, because of the large number of classes and methods in real-world C++ programs, we have found this to be impractical. Instead, we use a greedy algorithm to compute a sequence of edits that transforms the OOAnalyzer output to the ground truth.

Program	Constructor			Destructor			VF Tables			Virtual Methods		
	Recall	Prec	F	Recall	Prec	F	Recall	Prec	F	Recall	Prec	F
CImg	44/51	44/53	0.85	0/22	0/0	0.00	13/13	13/13	1.00	23/30	23/24	0.85
Firefox	40/51	40/54	0.76	1/39	1/1	0.05	18/33	18/18	0.71	85/101	85/98	0.85
light-pop3-smtp	41/52	41/44	0.85	2/27	2/2	0.14	5/5	5/5	1.00	6/7	6/6	0.92
log4cpp Debug	192/209	192/197	0.95	40/118	40/40	0.51	18/18	18/18	1.00	84/101	84/86	0.90
log4cpp Release	135/165	135/170	0.81	24/73	24/36	0.44	18/21	18/18	0.92	84/101	84/86	0.90
muParser Debug	293/325	293/314	0.92	28/156	28/30	0.30	12/12	12/13	0.96	35/47	35/43	0.78
muParser Release	197/252	197/269	0.76	15/91	15/21	0.27	12/14	12/13	0.89	35/47	35/37	0.83
MySQL cfg_editor.exe	260/290	260/311	0.87	107/281	107/111	0.55	69/69	69/69	1.00	321/427	321/325	0.85
MySQL connection.dll	10/10	10/25	0.57	8/36	8/9	0.36	10/13	10/10	0.87	22/39	22/22	0.72
MySQL ha_example.dll	15/19	15/21	0.75	4/19	4/6	0.32	9/9	9/9	1.00	162/170	162/162	0.98
MySQL libmysql.dll	283/310	283/340	0.87	115/297	115/119	0.55	75/75	75/75	1.00	348/453	348/352	0.86
MySQL mysql.exe	282/314	282/341	0.86	115/300	115/121	0.55	75/75	75/75	1.00	341/453	341/345	0.85
MySQL upgrade.exe	459/529	459/570	0.84	198/467	198/221	0.58	150/152	150/150	0.99	484/674	484/490	0.83
optionparser	10/11	10/10	0.95	0/1	0/0	0.00	6/6	6/6	1.00	8/8	8/8	1.00
PicoHttpD	117/142	117/126	0.87	68/109	68/72	0.75	46/46	46/46	1.00	119/159	119/119	0.86
TinyXML Debug	53/60	53/57	0.91	0/39	0/3	0.00	24/24	24/24	1.00	101/119	101/102	0.91
TinyXML Release	49/60	49/53	0.87	27/39	27/36	0.72	24/24	24/24	1.00	101/119	101/103	0.91
x3c	6/7	6/6	0.92	0/5	0/0	0.00	1/1	1/1	1.00	1/1	1/1	1.00
Malware 0faaa3d3	12/12	12/13	0.96	6/12	6/6	0.67	4/4	4/4	1.00	16/19	16/17	0.89
Malware 29be5a33	33/34	33/36	0.94	0/15	0/0	0.00	13/13	13/13	1.00	23/30	23/24	0.85
Malware 6098cb7c	50/52	50/51	0.97	8/15	8/9	0.67	43/43	43/43	1.00	103/106	103/103	0.99
Malware 628053dc	187/228	187/194	0.89	111/171	111/128	0.74	100/100	100/107	0.97	645/663	645/648	0.98
Malware 67b9be3c	464/532	464/490	0.91	169/342	169/188	0.64	123/123	123/123	1.00	139/249	139/217	0.60
Malware cfa69fff	27/29	27/30	0.92	6/24	6/6	0.40	5/5	5/5	1.00	16/20	16/16	0.89
Malware d597bee8	19/20	19/19	0.97	4/11	4/4	0.53	4/4	4/4	1.00	9/12	9/9	0.86
Malware deb6a7a1	262/320	262/275	0.88	159/262	159/182	0.72	130/130	130/137	0.97	842/889	842/871	0.96
Malware f101c05e	163/197	163/169	0.89	105/153	105/122	0.76	93/93	93/100	0.96	472/487	472/475	0.98
Average	0.88	0.88	0.87	0.32	0.88	0.41	0.96	0.99	0.97	0.82	0.96	0.88

Table 4: The recall and precision of various method properties achieved by OOAnalyzer, without utilizing RTTI data. A recall of X/Y indicates that OOAnalyzer detected X instances out of Y total in the ground truth. A precision of X/Y indicates that X of the Y instances that OOAnalyzer reported were actually correct. Green indicates a recall or precision higher than 0.75, whereas red is a value lower than 0.25.

6.4 Class Membership Results

Table 3 shows the edit distance results under three different experiments. For each experiment, we report the number of edits between OOAnalyzer's results and the ground truth, which we call the *absolute edit distance*. Since a program with more methods naturally provides more opportunities for mistakes, we also report the edit distance as a percentage of the number of C++ methods in the program, which we call the *relative edit distance*.

Without RTTI. In the first experiment, OOAnalyzer does not use RTTI data (Section 2.4) even if it is available, which may be appropriate when analyzing malicious or untrusted code. Even in this conservative experiment, OOAnalyzer achieves an average relative edit distance of 21.8%, which indicates that OOAnalyzer is recovering the vast majority of classes correctly. OOAnalyzer is able to recover C++ abstractions equally well for cleanware and malware, with the average relative edit distances being 23.6% and 18.3% respectively. In addition to the edit distances, Table 3 also displays the *types* of edits encountered in this experiment, which provides some insight into the types of mistakes that OOAnalyzer made. The most common edits are moves (40.9%) and adds (29%), which indicate assigning a method to the wrong class, and failing to detect a method. The large number of adds is in part due to an effort to reduce removals (5.8%). The higher number of splits (14.8%) than joins (9.4%) shows that OOAnalyzer is slightly more inclined to merge classes incorrectly than to fail to do so. This is likely caused by shared method implementations which is discussed more in Section 7.1.

Using RTTI. The second experiment evaluates OOAnalyzer's performance when it leverages RTTI data (Section 2.4). Since RTTI provides useful information about the polymorphic methods and classes in the class hierarchy, it is expected that OOAnalyzer would perform better when given access to this information. However, the results of this experiment show that OOAnalyzer only performs marginally better with access to RTTI, with the maximum and average improvements in absolute edit distance being 9 and 1.7. We found these results to be consistent with our intuition that most of OOAnalyzer's edits are related to non-polymorphic classes and methods; polymorphic methods are generally easier to recover.

Without hypothetical reasoning. The final experiment is the same as the *Without RTTI* experiment, except that OOAnalyzer's hypothetical reasoning component is disabled. By comparing this experiment to the first experiment, which uses OOAnalyzer's hypothetical reasoning component, we can measure the contribution of hypothetical reasoning. Without hypothetical reasoning, OOAnalyzer performs significantly worse, yielding 81% as the average relative edit distance (compared to 21.8% with hypothetical reasoning). These results highlight the importance of hypothetical reasoning, and reinforce the challenge of coping with uncertainty while recovering C++ abstractions.

6.5 Method Properties

Table 4 shows how well OOAnalyzer identifies special method properties in the absence of RTTI. Specifically, OOAnalyzer attempts to identify constructors, destructors, virtual methods, and virtual function tables. Each group of columns in the table reports the recall, precision and F-score (*i.e.,* the harmonic mean of precision and recall) for one of these properties. For example, on constructors, `CImg` has a recall of 44/51, which indicates that OOAnalyzer detected 44 of the 51 constructors in `CImg`, and a precision of 44/53, which indicates that OOAnalyzer reported 53 constructors total, of which 44 were correct. As expected, this results in a relatively high F score of 0.85.

With a few exceptions, OOAnalyzer is able to identify constructors with very high accuracy (average F score of 0.87). Unfortunately, destructor detection has proven more difficult, and OOAnalyzer only achieves an average F score of 0.41. We have found that destructors are often trivial implementations that are optimized away, which makes them more difficult to distinguish. Finally, like many other tools in this area, OOAnalyzer is able to identify virtual function tables with high accuracy (average F score of 0.97). As a result, OOAnalyzer can effectively distinguish most virtual methods as well (average F score of 0.88). Unlike most other tools, however, OOAnalyzer also reasons about non-virtual methods and non-polymorphic classes.

6.6 Performance

All experiments were performed using a single core of an Intel Xeon E5-2695 2.4Ghz CPU with 256 GiB of memory. Table 5 lists OOAnalyzer's running time and memory usage (in minutes and mebibytes, respectively) for each benchmark. OOAnalyzer's total running time ranges from 30 seconds to 22.7 hours, with a median and mean of 0.2 and 2.3 hours. OOAnalyzer's maximum memory usage ranges from 43.1 MiB to 3.5 GiB, with a median and mean of 0.7 and 1.0 GiB, respectively. Larger executables clearly tend to require more time and memory, as expected. On most larger executables, Prolog reasoning dominates the runtime of the system, whereas for smaller executables, fact exporting takes the bulk of the time.

7 DISCUSSION AND LIMITATIONS

7.1 Optimizations

Some compiler optimizations can modify executable code in ways that stops OOAnalyzer's rules from working as intended. One of the most problematic classes of optimization is Whole Program Optimization (WPO) [17] (enabled by the GL switch in Microsoft Visual C++), which allows the compiler to perform optimizations across multiple compilation modules at link time. Unfortunately, this switch allows the compiler to violate ABI conventions in functions that are not exported. For example, the compiler may decide to pass object pointers to methods in a register other than `ecx`, even if that method was declared to use the `thiscall` convention (Section 2.3). These optimizations make it more difficult to identify and track the data flow of object pointers.

Another problematic optimization is when the linker reuses identical function implementations. If the linker detects two symbols that consist of exactly the same executable code, it may only store one copy of the code and point both symbols at the same address. This is problematic for one of the fundamental assumptions in OOAnalyzer, which is that each method in the executable may only belong to one class. This optimization can cause OOAnalyzer to mistakenly conclude that two separate classes need to be merged. For instance, if the methods M_a and M_b have identical implementations it's possible that they will both be assigned to the same address. If M_a is on class Cl_a and M_b is on class Cl_b, then OOAnalyzer would likely make the errant conclusion that Cl_a and Cl_b are actually the same class because it has no way of knowing that M_a and M_b are distinct at the source code level. Such challenges cause difficulty for human reverse engineers as well, and demonstrate the complexity of the problem.

In some situations, optimizing compilers will *inline* a function by replacing a call to that function with a copy of the function's body. Unfortunately, inlining makes recovering C++ abstractions more difficult, since any behavior attributed to a particular function could actually be caused by an inlined function call. One of the most common cases is when constructors inline other constructors (or destructors inline destructors), which happens frequently because of inheritance and composition. When a constructor calls another constructor without inlining, it is easy to detect and usually indicates that the two constructors are on related classes. In the presence of inlining, it may not even be clear that there are two constructors involved. Many of OOAnalyzer's rules have been adjusted to account for common inlining situations. For example, when a constructor calls the constructor of an inherited class and the call is inlined, it is still possible to detect the inheritance relationship because the inlined code will include vftable (or vbtable) installations for both constructors. Such rules are among the most complex in OOAnalyzer, however, and it is impossible to handle all inlined situations perfectly.

7.2 Other platforms

OOAnalyzer is designed to analyze Windows executables produced using the Visual C++ ABI [11]. On many other platforms (e.g., Linux and Unix), C++ compilers target the Itanium C++ ABI instead [13]. Adding support to OOAnalyzer for the Itanium C++ ABI would primarily consist of adjusting the executable fact exporter to be able to detect the different conventions used for operations such as method calls and installing virtual function and base tables. We expect that few changes would be needed to most of the regular reasoning rules, because they reason at a semantic level that should be preserved across ABIs. However, the current hypothetical reasoning rules can be thought of as heuristics that are tuned for code

					Fact Exporting		Reasoning		Both
Program	Ver.	Opt.	Compiler	Size (KiB)	Time (min)	Mem. (MiB)	Time (min)	Mem. (MiB)	Time (min)
CImg	1.0.5		VS10	590	20.3	1,696.9	0.9	30.4	21.2
Firefox	52.0	✓	VS15	505	4.5	354.9	1.2	60.8	5.7
light-pop3-smtp	608b		VS10	132	2.5	326.4	0.2	24.0	2.7
log4cpp Debug	1.1		VS10	264	5.7	757.8	5.1	133.9	10.8
log4cpp Release	1.1	✓	VS10	97	1.7	136.8	0.9	83.2	2.6
muParser Debug	2.2.3		VS10	664	9.2	1,932.1	25.5	245.0	34.7
muParser Release	2.2.3	✓	VS10	302	4.6	422.8	21.1	210.1	25.7
MySQL cfg_editor.exe	5.2.0	✓	VS12	4,386	13.0	839.3	66.5	870.5	79.6
MySQL connection.dll	5.2.0		VS12	136	1.5	168.6	0.1	11.0	1.6
MySQL ha_example.dll	5.2.0	✓	VS12	54	0.3	43.1	0.7	26.9	1.1
MySQL libmysql.dll	5.2.0	✓	VS12	4,570	17.1	1,075.8	81.0	964.6	98.1
MySQL mysql.exe	5.2.0	✓	VS12	4,678	18.9	1,210.1	85.9	1,052.2	104.8
MySQL upgrade.exe	5.2.0	✓	VS12	5,321	20.1	1,363.9	630.2	2,994.0	650.2
optionparser	1.3		VS10	55	1.6	166.1	0.1	8.2	1.7
PicoHttpD	1.2		VS10	386	6.4	736.3	3.0	108.4	9.5
TinyXML Debug	2.6.1		VS10	594	10.3	1,675.5	1.9	60.2	12.1
TinyXML Release	2.6.1	✓	VS10	222	7.5	808.4	0.8	41.7	8.2
x3c	1.0.2		VS10	42	0.5	64.6	0.0	4.7	0.5
Malware 0faaa3d3			VS9	276	1.4	198.0	0.0	9.6	1.4
Malware 29be5a33			VS9	571	8.7	1,210.8	0.4	18.1	9.1
Malware 6098cb7c		✓	VS9	445	15.1	499.1	4.5	34.9	19.6
Malware 628053dc			VS10	1,322	57.4	3,063.6	550.0	1,428.8	607.3
Malware 67b9be3c			VS11	927	12.1	1,570.6	145.9	1,234.7	158.0
Malware cfa69fff			VS10	98	1.8	167.1	0.1	13.5	1.8
Malware d597bee8			VS10	68	1.0	140.1	0.1	9.1	1.1
Malware deb6a7a1			VS9	1,673	73.5	3,558.8	1,287.3	3,377.4	1,360.8
Malware f101c05e			VS9	1,256	61.3	2,785.0	484.2	1,140.8	545.5
Average					14.0	999.0	125.8	525.8	139.8

Table 5: The runtime (in minutes) and memory usage (in mebibytes) that OOAnalyzer consumed on each benchmark. *Fact Exporting* represents the runtime and memory used by the fact exporter, whereas *Reasoning* represents the resources used by the Prolog reasoning components.

produced by the Visual C++ compiler, and might need to be tailored for other compilers and ABIs. We expect this to mostly be a straightforward engineering effort, although there is also an interesting research question in whether these types of rules can be automatically inferred or tuned.

7.3 External classes

Dynamic linking can pose a challenge for C++ abstraction recovery because class definitions can span multiple executable files. A common example of this is when the implementation of a base class is loaded from a dynamically linked library (DLL), while the executable itself contains the implementation of the derived class. This pattern is common in Microsoft Foundation Class (MFC) programs. MFC provides a variety of base classes (implemented in the MFC library) that developers customize by creating a class (implemented in their executable) that inherits from the MFC class. In such programs, obtaining a complete understanding of the program's class relationships requires knowledge from both the executable and the DLL. One approach to this problem is to load the executable in conjunction with all of the required dynamic libraries for analysis. In addition to presenting scalability challenges, this approach can create confusion about whether the recovered C++ abstractions are only valid for the specific versions of the DLLs that were analyzed. This approach is also problematic for malware analysis, since executables are routinely collected without all required libraries.

Instead of loading all executables and DLLs at the same time, OOAnalyzer attempts to form a minimal understanding of external classes by parsing the relocation symbols used for dynamic linking. C++ compilers encode all the information that is needed to be able to call a method from another executable module using an encoding called *mangling*. The mangled name of a method encodes the class that the method belongs to, and attributes such as whether the method is virtual or a constructor. Unlike debug symbols, these relocation symbols are necessary to run the program, and cannot be stripped without breaking the program. To leverage this information, we built a custom demangler that extracts properties for names that are mangled according to the Visual Studio name mangling scheme.[6] This allows OOAnalyzer to reason about these class

[6] The Visual Studio name mangling scheme does not have a canonical source, but researchers have reverse engineered most of it [32].

methods without analyzing their code (which may not be available). Although our ground truth (Section 6.2) does not directly evaluate OOAnalyzer's understanding of external classes, we have found that this information is necessary to inform OOAnalyzer's understanding of internal classes that are *related to* external classes, which is counted.

8 RELATED WORK

8.1 Recovery of C++ Abstractions

The research most similar to ours recovers a broad set of C++ abstractions including grouping methods into classes, detecting relationships among classes, and detecting special methods such as constructors and destructors. Compared to these works, OOAnalyzer is relatively unique in that it statically recovers information about all classes (including non-polymorphic classes). Only two other works [14, 28] attempt to recover information about non-polymorphic classes, and only one does so statically, which we discuss first.

ObjDigger [14] is the predecessor of OOAnalyzer, and its development significantly informed the overall design and approach of OOAnalyzer. ObjDigger attempts to recover many of the same C++ abstractions as OOAnalyzer, and more importantly, is the only other system we know of that is able to *statically* recover non-polymorphic classes. Like OOAnalyzer, ObjDigger does not rely on RTTI data, and instead leverages vftable analysis and object pointer tracking. The most significant difference between ObjDigger and OOAnalyzer (and the primary inspiration for developing OOAnalyzer) is that ObjDigger reasons using procedurally written code. We found that as we tried to evolve ObjDigger to improve its accuracy, eventually it became too complicated to understand how it analyzes very complex scenarios. In OOAnalyzer, we overcame this largely by introducing hypothetical reasoning, which allows OOAnalyzer to reason through complex scenarios using simple rules. We showed in Section 6.4 that OOAnalyzer performed significantly worse when hypothetical reasoning was disabled.

Our evaluation includes the same five executables that ObjDigger was evaluated on (CImg, light-pop3-smtp, optionparser, PicoHttpD, and x3c) but in our prior work we evaluated them using a different metric. Specifically, we scored each program using the percentage of methods that were associated with the "correct class". As we note in Section 6.3, without a clear identifier such as a vftable, this metric is subjective and ill-defined whenever the recovered classes do not bear a clear resemblance to the ground truth. We believe edit distances are a much better metric.

We ran ObjDigger's results through our new edit distance based evaluation system to provide a fair comparison, and the results can be seen in Table 6. OOAnalyzer recovers classes more accurately for all programs, including the five programs that ObjDigger was originally tested on, which are shown in the first five rows. On these five programs, OOAnalyzer achieved an average relative edit distance of 15.7%, compared to 44.6% for ObjDigger. ObjDigger performed very poorly on the programs we did not evaluate in the ObjDigger paper [14]; it only achieved an edit distance of less than 50% on one program. It also failed to produce a result on four of the MySQL programs, either because it crashed or took longer than 24 hours. The majority of ObjDigger's edits are Adds, which indicates that ObjDigger did not detect the method, or was unable to determine which class it is associated with. Finally, although ObjDigger can also identify constructors, in our past work we did not measure its effectiveness at that or on any of the properties in Table 4.

Lego [28] is another system that can recover non-polymorphic classes from C++ executables. Unlike OOAnalyzer (and ObjDigger), Lego recovers this information by processing *dynamic* runtime traces, which allows it to recover class hierarchies from OO languages other than C++. In addition to recovering classes, Lego can also recover inheritance and composition relationships between classes, and identify destructor methods. Lego's primary disadvantage is that, as a dynamic analysis, it relies on having test inputs that trigger the usage of classes and methods. Unfortunately, this makes Lego less applicable when such inputs are unavailable. It also makes it difficult to perform an apples-to-apples comparison with OOAnalyzer, because Lego's performance depends on the quality of the testcases it uses.

SmartDec [9, 10] is a C/C++ decompiler for executables. SmartDec naturally recovers C++ abstractions, but also has functionality needed for decompilation such as control flow structuring and exception handler analysis, which OOAnalyzer does not. Similar to OOAnalyzer, SmartDec tracks object pointers, performs vftable analysis, and does not rely on RTTI. SmartDec, however, only attempts to recover the methods of polymorphic classes (*i.e.,* classes with virtual functions).

Yoo and Barua [33] describe a system using SecondWrite [2] to statically recover a wide variety of C++ abstractions, including exception handlers. Their system relies on RTTI data, and thus only recovers information about polymorphic classes. Their approach may also be infeasible when analyzing malware, which sometimes has RTTI data disabled to impede analysis.

Katz [15] uses a combination of program analysis and machine learning to map virtual calls to their targets. They train a classifier to estimate each method's likelihood of being dispatched based on learned statistical models of object usage events, including reads, writes, and calls. These object usage events are generated using a lightweight static symbolic analysis similar to OOAnalyzer's (Section 4.1). Whereas OOAnalyzer uses hand-written rules that encode C++ domain knowledge, Katz uses models that are automatically trained for each program. Future research could explore using machine learning to automatically generate C++ reasoning rules for OOAnalyzer.

8.2 Security Protections for C++ Binaries

Early control-flow integrity (CFI) protection systems [1] inferred allowed control-flow transitions from source code. Later, researchers developed CFI systems using binary analysis and rewriting techniques that could be applied directly to executables without requiring access to source code [31, 35]. Such systems did not take into account any knowledge of C++ implementation mechanisms, and enforced relatively coarse-grained policies for C++ executables [21, 24]. While some researchers have proposed techniques that improve precision in language-agnostic ways [30], those focusing on recovering C++ abstractions [8, 19, 21, 34, 35] are more comparable to OOAnalyzer.

Program	In [14]	Opt.	Compiler	Size (KiB)	Num. Classes	Num. Methods	ObjDigger							OOAnalyzer	
							Move	Add	Rem	Split	Join	Edits	%	Edits	%
CImg	✓		VS10	590	29	220	15	101	4	0	1	121	55.0	21	9.5
light-pop3-smtp	✓		VS10	132	44	295	8	113	0	11	2	134	45.4	44	14.9
optionparser	✓		VS10	55	11	56	1	22	0	0	0	23	41.1	6	10.7
PicoHttpD	✓		VS10	386	95	656	37	316	24	15	1	393	59.9	166	25.3
x3c	✓		VS10	42	6	28	0	5	1	0	0	6	21.4	5	17.9
Firefox		✓	VS15	505	141	638	2	459	41	5	0	507	79.5	212	33.2
log4cpp Debug			VS10	264	139	893	6	806	16	1	0	829	92.8	239	26.8
log4cpp Release		✓	VS10	97	76	378	2	258	12	0	0	272	72.0	75	19.8
muParser Debug			VS10	664	180	1437	21	1334	4	2	0	1361	94.7	483	33.6
muParser Release		✓	VS10	302	94	598	8	352	2	7	0	369	61.7	183	30.6
MySQL cfg_editor.exe		✓	VS12	4,386	190	1266	—	—	—	—	—	—	—	391	30.9
MySQL connection.dll			VS12	136	43	167	0	121	1	7	0	129	77.2	48	28.7
MySQL ha_example.dll		✓	VS12	54	21	256	5	230	0	2	0	237	92.6	32	12.5
MySQL libmysql.dll		✓	VS12	4,570	200	1328	—	—	—	—	—	—	—	406	30.6
MySQL mysql.exe		✓	VS12	4,678	202	1395	—	—	—	—	—	—	—	439	31.5
MySQL upgrade.exe		✓	VS12	5,321	333	2071	—	—	—	—	—	—	—	655	31.6
TinyXML Debug			VS10	594	35	415	17	240	3	5	3	268	64.6	69	16.6
TinyXML Release		✓	VS10	222	33	283	11	151	5	7	0	174	61.5	55	19.4
Malware 0faaa3d3			VS9	276	21	135	1	118	1	1	0	121	89.6	21	15.6
Malware 29be5a33			VS9	571	19	130	12	78	1	0	0	91	70.0	15	11.5
Malware 6098cb7c		✓	VS9	445	55	339	0	129	0	2	0	131	38.6	29	8.6
Malware 628053dc			VS10	1,322	207	1920	14	1214	7	9	1	1245	64.8	378	19.7
Malware 67b9be3c			VS11	927	400	2072	78	1091	29	89	12	1299	62.7	670	32.3
Malware cfa69fff			VS10	98	39	184	3	117	1	3	1	125	67.9	37	20.1
Malware d597bee8			VS10	68	19	133	2	62	0	2	2	68	51.1	17	12.8
Malware deb6a7a1			VS9	1,673	283	2712	20	1861	7	11	1	1900	70.1	639	23.6
Malware f101c05e			VS9	1,256	169	1601	12	961	5	8	1	987	61.6	329	20.5
Average					114	800							65.0		21.8

Table 6: The edit distance between the classes that ObjDigger recovered and the ground truth. The first five programs in the table are those evaluated in the original ObjDigger publication [14]. A low edit distance indicates that the class assignments are close to the ground truth. The edit distance for ObjDigger is broken down into the number of move, add, remove, split and join edits to reveal the types of errors that ObjDigger made. These sum to the total edit distance which is also reported as a percentage of methods. For comparison, OOAnalyzer's total edit distances are reported as well. The green background shading indicates the best result.

vfGuard [21] and VTint [34] are examples of CFI systems that incorporate C++ specific protections. Both systems identify and recover information about virtual call sites and vftables. vfGuard attempts to sanitize virtual calls based on this information, whereas VTint relocates identified vftables to a read-only segment of memory, and checks before each virtual call that the referenced vftable is in read-only memory. More recently, other C++-specific CFI systems such as MARX [19] and VCI [8] have begun recovering additional information such as inheritance hierarchies. The inheritance hierarchy information strengthens enforcement policies by disallowing virtual calls to unrelated classes. Although MARX and VCI attempt to recover the inheritance hierarchy, they make no attempt to determine the direction of inheritance relationships as OOAnalyzer does, which could further strengthen the inferred CFI policies.

All four of these systems only recover C++ abstractions that are required to protect virtual calls, thereby only recovering information about polymorphic classes. In contrast, OOAnalyzer attempts to recover all methods on all classes that are implemented in the target binary, including non-polymorphic methods and classes.

8.3 Detection of C++ Vulnerabilities

The RECALL system [6, 7] recovers vftables, constructors and destructors in addition to tracking the dataflow of object pointers. It uses this information to detect *vftable escape* vulnerabilities by observing if the offset into a vftable is too large for the intended type of the object. OOAnalyzer uses similar logic in its forward reasoning rules (Section 3.2.2) to group methods into classes and recover relationships among classes.

9 CONCLUSIONS

We showed that recovering detailed C++ abstractions is possible through the creation of OOAnalyzer. OOAnalyzer uses a lightweight symbolic analysis to efficiently generate an initial set of facts, and analyzes them using a Prolog-based reasoning system.

We evaluated OOAnalyzer and showed that it is both scalable and accurate. It recovered abstractions on programs as complex as Firefox and MySQL, and from C++-based malware executables. It identifies the classes in an executable and the methods of those classes with high accuracy (average 21.8% error rate), and can distinguish special methods such as constructors, destructors, virtual function tables, and virtual methods (average F-scores of 0.87, 0.41, 0.97, and 0.88).

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] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. In *Proceedings of the ACM European Conference on Computer Systems*.

[3] Boost. 1998. Boost C++ Libraries. (1998). Retrieved 14 Aug. 2018 from http://www.boost.org

[4] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

[6] 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*.

[7] 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*.

[8] 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*.

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

[10] 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*.

[11] Jan Gray. 1994. *C++: Under the Hood*. Technical Report. Microsoft. Retrieved August 14, 2018 from http://www.openrce.org/articles/files/jangrayhood.pdf

[12] Hex-Rays. 2017. Hex-Rays IDA Disassembler. (2017). Retrieved 14 Aug. 2018 from https://www.hex-rays.com/products/ida/

[13] Itanium 2017. Itanium C++ ABI. (March 2017). Retrieved 14 Aug. 2018 from https://itanium-cxx-abi.github.io/cxx-abi/

[14] 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*.

[15] 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*.

[16] Microsoft. 2015. Information from Microsoft about the PDB format. (29 Oct. 2015). Retrieved 14 Aug. 2018 from https://github.com/Microsoft/microsoft-pdb

[17] Microsoft. 2016. /GL (Whole Program Optimization). (Nov. 2016). Retrieved 14 Aug. 2018 from https://docs.microsoft.com/en-us/cpp/build/reference/gl-whole-program-optimization

[18] OOAnalyzer 2018. OOAnalyzer prolog rules. (10 May 2018). Retrieved 14 Aug. 2018 from https://github.com/cmu-sei/pharos/tree/master/share/prolog/oorules

[19] 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*.

[20] Pharos 2017. Pharos project page. (2017). Retrieved 14 Aug. 2018 from https://github.com/cmu-sei/pharos

[21] 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*.

[22] ROSE 2018. ROSE compiler infrastructure. (2018). Retrieved 14 Aug. 2018 from http://rosecompiler.org/

[23] Paul Vincent Sabanal and Mark Vincent Yason. 2007. Reversing C++. In *Proceedings of Black Hat USA*.

[24] 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*.

[25] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*.

[26] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the Network and Distributed System Security Symposium*.

[27] Igor Skochinsky. 2006. Reversing Microsoft Visual C++ Part 2: Classes, Methods and RTTI. (2006). Retrieved 14 Aug. 2018 from http://www.openrce.org/articles/full_view/23

[28] 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*.

[29] Terrance Swift and David S. Warren. 2012. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming* 12, 1-2 (2012).

[30] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[31] 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*.

[32] Wikiversity. 2017. Visual C++ name mangling. (2017). Retrieved 14 Aug. 2018 from https://en.wikiversity.org/wiki/Visual_C%2B%2B_name_mangling

[33] Kyungjin Yoo and Rajeev Barua. 2014. Recovery of Object Oriented Features from C++ Binaries. In *Proceedings of the IEEE Asia-Pacific Software Engineering Conference*.

[34] 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*.

[35] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the USENIX Security Symposium*.

ACKNOWLEDGMENTS

Copyright 2018 ACM. All Rights Reserved. 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.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

DM18-0949