DeClassifier: Class-Inheritance Inference Engine for Optimized C++ Binaries

Rukayat Ayomide Erinfolami
Binghamton University
rerinfo1@binghamton.edu

Aravind Prakash
Binghamton University
aprakash@binghamton.edu

ABSTRACT

Recovering class inheritance from C++ binaries has several security benefits including in solving problems such as decompilation and program hardening. Thanks to the optimization guidelines prescribed by the C++ standard, commercial C++ binaries tend to be optimized. While state-of-the-art class inheritance inference solutions are effective in dealing with unoptimized code, their efficacy is impeded by optimization. Particularly, constructor inlining—or worse exclusion—due to optimization render class inheritance recovery challenging. Further, while modern solutions such as MARX can successfully group classes within an inheritance sub-tree, they fail to establish directionality of inheritance, which is crucial for security-related applications (e.g. decompilation). We implemented a prototype of `DeClassifier` using Binary Analysis Platform (BAP) and evaluated `DeClassifier` against 16 binaries compiled using gcc under multiple optimization settings. We show that (1) `DeClassifier` can recover 94.5% and 71.4% true positive directed edges in the class hierarchy tree (CHT) under O0 and O2 optimizations respectively, (2) a combination of constructor-destructor (ctor-dtor) analysis provides a substantial improvement in inheritance inference than constructor-only (ctor-only) analysis.

CCS CONCEPTS

• **Binary Analysis** → **Class hierarchy recovery**; • **Software Security** → CFI;

KEYWORDS

Class hierarchy recovery, software reverse engineering

ACM Reference Format:
Rukayat Ayomide Erinfolami and Aravind Prakash. 2019. `DeClassifier`: Class-Inheritance Inference Engine for Optimized C++ Binaries. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3321705.3329833

1 INTRODUCTION

Recovery of class inheritance of a C++ program is useful in many ways, and often necessary. While extracting class hierarchy from source code is straightforward (e.g., class-hierarchy analysis [2, 10, 11, 15, 18, 25, 28]), recovering class hierarchy from a binary is hard [22, 24], but useful. For example, any attempt at C++ program decompilation must infer at least a partial class hierarchy from a binary [7, 8]. Similarly, defenses that enforce strict control-flow integrity (CFI) policies on C++ binaries rely on class hierarchy analysis (e.g., Marx [19], VCI [6]).

Although RunTime Type Information (RTTI), a per-class type-revealing data structure may be present in certain C++ programs, it is often absent in commercial-off-the-shelf (COTS) binaries. On the one hand, RTTI structure contains information about the parents of a given polymorphic class, and it can be used to reliably reconstruct the class hierarchy of a program. But on the other hand, the use of RTTI is discouraged in commercial code due to the high runtime overhead imposed by operators (i.e., `dynamic_cast` and `typeinfo`) that use RTTI [15]. In fact, most COTS software are closed source and do not contain RTTI in the binary. Without RTTI, inferring class hierarchy (high level semantics in general) from COTS C++ software poses multiple challenges. These challenges are often rooted in optimization that is common in C++ code.

First, most solutions (e.g., VCI [6], SmartDec [7, 8], HexRays Decompiler [24]) heavily rely on constructor analysis due to the well-defined inheritance-revealing control flow during construction of a C++ object. However, constructor analysis suffers from low precision and is insufficient. This is because COTS C++ binaries are often optimized and tend to have many inlined functions including inlined constructors. In fact, per ISO C++ 7.1.2/3—"*A function defined within a class definition is an inlined function*". Second, aggressive compiler optimization could result in the exclusion of key functions (e.g., constructors) and/or entire classes from the binary, which makes inference hard. For example, when a most derived class is not instantiated, the compiler may conveniently exclude such class definitions from the binary. In fact, we consistently found a significant reduction in the number of constructors in the binary with higher levels of optimizations (see Table 9 in Appendix C). Finally, *is-a* and *has-a* relationships between classes pose a disambiguation challenge. It is hard to discern inheritance relationship (i.e., A inherits from B or A *is-a* B) from composed relationship (i.e., A *has-a* object of B)–especially in the case of optimized code.

These challenges are evidenced in most relevant recent works VCI [6] and Marx [19]. These efforts employ class hierarchy analysis on C++ binaries without relying on RTTI. On the one hand, VCI's precision and accuracy are dependent on constructor identification, which in turn is heavily impeded by inlined or missing constructors. On the other hand, Marx acknowledges the difficulty imposed by optimization and inlining, and limits its scope to identifying class membership to inheritance trees without actually recovering

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329833

a directed inheritance tree. Given the common occurrence of optimization in COTS C++ binaries, there is a pressing need to design class inference tools that can handle optimized code.

In this paper, we present DeClassifier, a robust class inheritance inference engine for C++ binaries. DeClassifier employs static analysis and is built on top of BAP [3]. Unlike prior efforts, DeClassifier can recover class hierarchy from optimized C++ binaries. As a key distinction, our inference engine is based on code features that *cannot* be optimized away (i.e., eliminated) during compile time. As such, these features form robust inference points. DeClassifier incorporates multiple novel analysis techniques in order to handle optimized code including inlined and missing constructors. This makes DeClassifier apt for COTS binaries. First, we take advantage of the fact that in COTS C++ software, destructors tend to be virtual in order to avoid memory leaks. Because calls to virtual functions can not be statically resolved, compilers can not inline virtual functions during compile time, and retain them in the binary without inlining their code at the callsites. Therefore, in comparison with constructors that are non-virtual, destructors in a binary tend to be more in number. As such, DeClassifier employs a combination of *constructor-destructor* analysis to achieve optimal recovery. Second, because virtual functions must be retained in the binary, we employ intra-procedural object-layout analysis on virtual functions to construct an object model for each class and use it to recover inheritance relationship from functions with inlined constructors or inlined destructors. Thereby also eliminating false positives in the inheritance relationships. Finally, we identify precise points of completion of main object initialization in order to distinguish between composed and inherited objects.

Full CHT recovery is hard: In general, recovering C++ semantics from optimized binaries is hard. Although DeClassifier employs multiple novel techniques to handle optimized code, C++ compilers eliminate entire classes from the binary—if the classes are deemed to be unnecessary (e.g., through dead code elimination) during compilation. In such cases, DeClassifier misses classes that have no remnants in the binary. Even so, to the best of our knowledge, DeClassifier is the only *practical* solution that can effectively infer *directed* class hierarchy tree from optimized code.

Our contributions can be summarized as follows:

- We present DeClassifier, an inference engine for recovering class hierarchy information from optimized C++ code.
- We employ multiple novel analysis techniques including *constructor-destructor* analysis, intra-procedural object layout analysis, and precise identification of object completion. These techniques allow DeClassifier to handle optimized code including inlined or missing constructors, distinguish between constructors and non-virtual destructors, and decipher between composed and inherited objects. Although destructor analysis has been previously suggested as a technique to recover class hierarchy [8], we show that: (1) destructor analysis is more effective and reliable than constructor analysis, and (2) optimized code can be successfully reversed using destructor analysis.
- We evaluate DeClassifier on 16 binaries. On average we are able to correctly recover substantial parts of the inheritance graph. Specifically, 94.5% (recall) of edges under O0 and 71.4% (recall) of edges under O2.

2 TECHNICAL BACKGROUND

2.1 Implementation of Polymorphism and Inheritance in C++

In order to implement polymorphism, C++ compilers utilize a per-class supplementary data structure called a "VTable". The structure of the VTable is dictated by the Itanium [1] and MSVC [21] C++ ABIs. A VTable is allocated for each polymorphic class (i.e., a class which: (1) defines virtual functions, or (2) inherits from other class(es) that defines virtual functions, or (3) inherits a class virtually). Each VTable contains an array of function pointers representing the polymorphic functions that the object of a given type can invoke at runtime. In order to prevent corruption, the VTables are allocated in the read-only sections of a binary. Each object of a polymorphic class contains an implicit pointer to the VTable (vptr) as the first member variable. Within the constructor of a polymorphic class, the vptr is initialized to point to the VTable for the type of the object being constructed. Among other fields, each VTable contains 3 mandatory fields—*OffsetToTop*, a pointer to *TypeInfo* (also called RTTI), and an array of virtual function pointers or vfptrs. Although RTTI is a mandatory field, a NULL value is used to signify its exclusion during compilation. In the past, mandatory fields have been used as a signature for identification of VTables in the binary [20].

In the case of multiple inheritance, wherein a class derives from more than one polymorphic base class, the first base class in the declaration order is referred to as a "primary base class", and the remaining bases are called "secondary base classes". The VTable for derived class comprises of a group of 2 or more VTables—a primary and one or more secondary VTables depending on the number of secondary bases. The derived class and its primary base class share the primary VTable, and each secondary base class is allocated a secondary VTable. Further, the derived object and the primary base sub-object share the same base address, and each secondary base sub-object is found at a positive offset from the derived object base address. The VTable group comprising of the primary and secondary VTables is collectively called a "complete-object VTable".

The offsetToTop field indicates the displacement that must be added to the sub-object within a derived object to reach the base of the derived object. If the RTTI value is not null, the RTTI pointers for all the VTables within a complete-object VTable point to the same RTTI structure. For more information on each of the fields and other optional fields in the VTable, we refer readers to the ABI [1].

2.2 Construction and Destruction

The constructor and destructor are called when an object of a class is created and destroyed respectively. Each class may contain one or more constructors and a single destructor (except for the case of virtual inheritance, or when there is a deleting destructor). If no constructor is defined by the developer, a default constructor will be provided by the compiler. During construction, constructors

of base classes are first invoked, starting from the primary base. The address of the sub-object being constructed is passed as an argument to the base class constructor. Then the vptr(s) of the object being constructed is assigned into appropriate offset(s). Finally, member variables including composed members are constructed. Similarly, during object destruction, the vptr of the object gets assigned. Then member variables including composed member objects are destroyed. Finally, destructors of immediate bases are invoked, ending with the primary base. Again, the address of the sub-object being destroyed is passed as an argument.

As a part of the construction and destruction of member variables, constructors and destructors may themselves invoke virtual functions. Therefore assignment of vptr occurs before the respective initialization/finalization of member variables. Note that the constructors and destructors of each base class write their own set of vptrs in appropriate locations in the object, which get overwritten by subsequent classes in the inheritance chain.

2.3 Virtual Destructors

Unlike constructors, destructors in C++ can be–and often are–declared as virtual. It may sometimes be necessary to delete a derived class object that is referenced through a base class pointer. The C++ standard states that deleting an object of derived class through a pointer to its base class that has non-virtual destructor leads to undefined behavior (see paragraph 3 in ISO/IEC 14882-2014). Therefore, it is common practice to mark destructors as virtual, which forces runtime resolution of the virtual call to the correct derived class destructor. These destructors must therefore be retained in the binary. As shown in Table 9 (Appendix C), binaries usually contain more destructors than constructors.

3 SOLUTION OVERVIEW

3.1 Motivating Example

Consider a synthetic program in Figure 1 with classes A, B, C and D, the disassembly of D's destructor, the inheritance and composition relationships, and the VTable layout for all four classes. C and B are primary and secondary base classes of D, while A is composed in D. Each class declares a constructor, a virtual destructor and a virtual function. The running example has five important characteristics:

(1) It is reflective of real-world applications where destructors are virtual and the program contains multiple inheritance as well as composition.
(2) Constructors have been eliminated through optimization.
(3) Destructors are retained in spite of optimization.
(4) The destructors of D's base classes (C and B) were inlined in D's destructor.

There are existing solutions which attempt to recover class hierarchy from binaries, however, they fail in one or more ways while dealing with real-world applications. Table 1 shows the techniques used by these solutions, their limitations and CHT recovered for the running example. We considered the following:

(a) HandleInlining: Can correctly identify relationships when constructors or destructors are inlined either in derived class constructors or destructors or in other functions.
(b) InhVsComp: Can differentiate inheritance from composition.
(c) DtorAnalysis: Uses destructor analysis.
(d) OLA: Uses Object Layout in inheritance inference.
(e) CHTRecovery: Shows the output generated for the running example.

We ran Marx on over 12 binaries and found that it can handle (a) and (b) above, but lacks (c). Katz et al[13] employ a predictive modeling based approach and can handle (a) and (b), but does not consider (c). From the description in the literature for VCI and SmartDec, we see that they cannot handle (a), but can handle (b) only when there is no optimization. VCI does not consider (c), SmartDec does, but their approach is incapable of handling optimized code. ObjDigger does not handle any of the capabilities we considered. Lego handles (a), but does not handle (b) and does not consider (c). None of the existing solutions consider (d).

In Table 1, we evaluated tools that are either opensource or ones for which we were able to get an artifact from the authors. For the remaining tools, we estimate the hierarchy based on the techniques used in the paper. Lego can assign direction of inheritance only when the destructor is explicitly called. All the other solutions will either recover no hierarchy (VCI and SmartDec) or ignore direction of inheritance (Marx and Katz et al). The limitations of existing solutions highlight the facts that (1) ctor-only approach is insufficient (2) delineating has-a and is-a relationship between derived and base objects are necessary for faithful inheritance construction and (3) assigning direction of inheritance is challenging but necessary.

Table 1: Class hierarchy extraction from binaries. Entry *p* implies recovery only when constructors not inlined, "{}" means no hierarchy was recovered, and "N/A" means that the tool does not recover hierarchy.

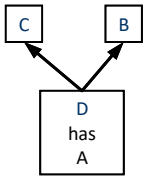
Solution	Key Technique	Handle-Inlining	InhVs-Comp	Dtor-Analysis	OLA	CHT-Recovery
VCI[6]	Static ctor analysis	✗	*p*	✗	✗	{}
Marx[19]	Static overwrite analysis	✓	✓	✗	✗	$\{D_{pry}, C\}$, $\{D_{sec}, B\}$
Katz et al.[13]	Object tracelet, predictive modeling	✓	✓	✗	✗	$\{D_{pry}, C\}$, $\{D_{sec}, B\}$
SmartDec[7]	Static ctor/dtor analysis	✗	*p*	*p*	✗	{}
ObjDigger[12]	Symbolic execution, data flow analysis	✗	✗	✗	✗	N/A
Lego[27]	Dynamic analysis	✓	✗	✗	✗	$D- > C$, $D- > B$
DeClassifier	Static ctor-dtor, overwrite analysis, OLA	✓	✓	✓	✓	$D- > C$, $D- > B$

3.2 Key Challenges

C1: Missing Constructors One of the results of compiler optimization is complete removal of constructors. As shown in Figure 1, even though the same simple operations ("printf") performed by

```
#include <iostream>
class A{
 public:
  int a;
  A(){printf("A\n"); }
  virtual ~A(){printf("~A\n"); }
  virtual void af(){...}
};
class B{
 public:
  int b;
  B(){ printf("B\n"); }
  virtual ~B(){printf("~B\n");}
  virtual void bf(){...}
};
```

```
class C{
 public:
  int c;
  C(){printf("C\n"); }
  virtual ~C(){printf("~C\n");}
  virtual void cf(){...}
};
class D: public C, public B{
 public:
  int d;
  A *a;
  D(){printf("D\n"); }
  virtual ~D(){printf("~D\n");
  a = new A();}
  virtual void df(){...}
};
```



Field	VTable(A)	VTable(B)	VTable(C)	VTable(D)
Offset to Top	0	0	0	0
RTTI	0	0	0	0
Vfptr[0]	&A::~A()	&B::~B()	&C::~C()	&D::~D()
Vfptr[1]	&A::~A()	&B::~B()	&C::~C()	&D::~D()
Vfptr[2]	&A::af()	&B::bf()	&C::cf()	&C::cf()
Vfptr[3]				&D::df()
Offset to Top				-16
RTTI				0
Vfptr[0]				thunk to D::~D()
Vfptr[1]				thunk to D::~D()
Vfptr[2]				&B::bf()

```
; D::~D(void)
...
1 mov rbx, rdi
2 mov qword ptr [rdi], offset off_201CC8 ;pry vptr D
3 mov qword ptr [rdi+10h], offset off_201CF8 ;sec vptr D
...
4 mov qword ptr [rbx+20h], offset off_201C50 ;vptr A
5 mov qword ptr [rbx+10h], offset off_201C78 ;vptr B
6 mov qword ptr [rbx], offset off_201CA0 ;vptr C
...
```

Figure 1: Running example comprising classes A, B, C and D, and corresponding VTable layout when compiled using g++ under -O2 optimization. The binary has no constructor. Note: 'pry' means primary, and 'sec' means secondary.

Listing 1: D's constructor and an initializing function

```
D::D(void) ...
1 mov rbx, rdi
2 mov qword ptr [rdi], offset off_201CA0 ;vptr C
3 mov qword ptr [rdi+10h], offset off_201C78 ;vptr B ...
4 mov qword ptr [rbx], offset off_201CC8 ;pry vptr D
5 mov qword ptr [rbx+10h], offset off_201CF8 ;sec vptr D
6 mov qword ptr [rbx+20h], offset off_201C50 ;vptr A ...

foo() ...
1 call D::D(void) ...
```

the constructors are also performed by the destructors, only the destructors are retained in the binary. This is because a virtual function is guaranteed to be in the binary as long as the VTable it belongs to is present. In fact, we found that significant number of constructors are optimized out during compilation, and their definitions are excluded from the binary (see Table 9 Appendix C).

C2: Constructor Inlining Compilers inline functions by replacing the callsite with the body of the called function. Virtual function callsites cannot be inlined since the exact function to call is only known at runtime depending on the object type. However, because constructors cannot be virtual, their calls are statically resolved and inlined when possible. In fact, we found this to be very common and prescribed by the C++ standard (see ISO C++ 7.1.2/3). Any function defined within a class definition will be inlined as a default behavior. This is a challenge since state-of-the-art class hierarchy recovery tools like VCI [6] depend primarily on the identification of constructors. To illustrate the problem, let us have a constructor for D in the running example (D's constructor will be retained if it performs complex operations) and an initializing function (Listing 1).

Constructor inlining gives rise to two problems:

- *Missed base class constructor calls*: Consider Listing 1 where the constructors of the primary and secondary bases of D are inlined on lines 2 and 3 respectively. In order not to include composed classes in a given class hierarchy, VCI looks at the first primary vptr initialization to the base address of an object which appears on line 2 and concludes that the constructor belongs to the class with vptr 0x201CA0 (i.e., C instead of D), subsequent constructor calls are ignored. As such it fails to identify any relationship between D and C or D and B. Overwrite analysis adopted by Marx will be able to group the primary vptr of D with the primary vptr of C as well as the secondary vptr of D with the primary vptr of B, however, it cannot differentiate the derived class from the base class.
- *False constructor identification*: In higher levels of optimization, the compiler could inline entire constructor D() in the instantiating function `foo`. Therefore, although not a constructor, `foo` would contain vptr initialization. In order to accommodate inlining, VCI identifies constructors by only looking for vptr initialization, not requiring that the write is at zero offset from object address (Section 4.2 in [6]). If `foo` calls other functions which also contain inlined constructors, a false relationship is inferred among the vptrs these non-constructor functions initialize.

C3: Inheritance vs Composition Failure to correctly differentiate the base class constructors (or destructors) from those of member classes will result in false inheritance inference between a class and its member classes. VCI partially handles this by considering only constructor calls that happen before initialization of the primary vptrs, however, this works only for constructors and not

destructors, since the derived class vptrs get written first. A general approach is required to differentiate composed and inherited objects for constructors or destructors.

3.3 Scope and Assumptions

Our primary goal is to leverage multiple binary-level features to reconstruct polymorphic non-virtual class inheritance tree from COTS binaries—even in an optimized setting. We target COTS C++ binaries which might have been compiled with high levels of optimization and with no debugging information, symbol information, RTTI, etc. Also, we assume that the source code of such binaries is not available. The core techniques of `DeClassifier` are based on the ABI and are compiler agnostic. We currently support any C++ compiler that adheres to the Itanium ABI (e.g., gcc, llvm), however, DeClassifier can be easily modified to support MSVC ABI.

3.4 Our approach

As a preliminary step, we recover all the VTables in the binary and group them into complete-object VTables. We utilize an already known scanning-based algorithm [20] to extract VTables. These VTables include primary and secondary VTables, which are then grouped to form complete-object VTables. In a nutshell, we start with a primary VTable (i.e., offsetToTop = 0), and group it with succeeding secondary VTables (i.e., offsetToTop ≠ 0) until we reach the next primary VTable. Each unique group is a complete-object VTable and provides a one-to-one mapping between the complete-object VTables and polymorphic classes in the binary. In the remainder of this section, we outline the analyses we undertake to infer inheritance relationships among the complete-object VTables, i.e., polymorphic classes.

Key insight: Code features that require runtime decisions (e.g., virtual function dispatch) can not be optimized away by the compiler, and therefore provide a robust source for inheritance inference. Our inheritance inference approach is based on identifying key inference points that *cannot* be optimized away during compilation (an exception being the removal of entire classes). Particularly, we leverage the virtual functions including virtual destructors to infer inheritance semantics. This way, we ensure meaningful inference even under strong compiler optimization.

Combining constructors with destructors We combine constructor-destructor analysis to achieve optimal recovery. State-of-the-art binary-level class inheritance extraction tools have primarily focused on constructor analysis. However, the number of constructors present in the binary decreases as the level of optimization increases, thus leading to inaccurate inference. Like constructors, destructors also provide insight into a particular class inheritance. Typically, the number of destructors in the binary tend to be more than the number of constructors (Table 9 in Appendix C).

In order to prevent memory leak, destructors are declared virtual which ensures that specific objects are destructed as expected.

Constructors cannot be virtual and can be removed or inlined. However, destructors are commonly virtual and are explicit functions in the binary, with or without inlining.

This eliminates the possibility of a virtual destructor not being present in the binary. **C1** is thereby addressed since we can augment destructors with available constructors.

Identifying valid constructors and destructors Inlining of constructors within other 'host' functions results in false inference of the host function as a constructor. This is one of the main reason for falses in analyzing optimized code. Therefore, if we can correctly eliminate such functions, we can safely analyze constructors and destructors, and correctly identify explicit calls to (or inlined) base constructors and destructors and also correctly eliminate calls to composed class constructors and destructors. This handles challenge **C3**. Only constructors (and destructors) write vptr to the implicit `this` pointer. We employ static analysis to detect whether the `this` pointer is initialized with a vptr. If so, it is classified as a constructor, else it is a function that contains inlined constructor.

We also perform additional analysis, explained in Section 4.2 to differentiate constructors from destructors. This helps us to address "false constructor identification" under **C2**. In order to address "missed base class constructor calls" for constructors that inline their base constructors, we ensure that the correct primary vptr associated to a constructor is identified. We do the same for destructors with inlined base destructors.

Object Layout Analysis (OLA) There are cases where destructors are not virtual, in that case, they could also be optimized out just like constructors. This creates the possibility of some classes having neither constructor nor destructor present in the binary. In cases where we can not find an explicit constructor or destructor for a class, we employ intra-procedural static analysis to model the object layout. Specifically, we start from the explicit virtual functions in a class' VTable (these functions cannot be optimized out). Next, we identify type-revealing instructions within the functions. Starting from these instructions, we obtain a backward slice and back-propagate type information to obtain a mapping:

$$this + offset \rightarrow type$$

By extending the analysis across all the classes, and checking different class objects for type congruence (i.e., member types at a given offset across all polymorphic classes must be the same), we can eliminate inconsistent inferences.

Identifying Completion of main Object Initialization (COI) Completion of initialization of a class' main object (a.k.a main object initialization) is the point during construction where base class vptrs and class' vptrs have all been completely written and overwritten in the object. Construction of composed objects always takes place *after* the construction of main object has completed. Irrespective of compiler optimization, we can conclude that vptr initialization that occur before this point belong to base classes while those that occur after it belong to composed objects. We found overwrite analysis [19] to be effective in identifying COI. This saves us from relying on explicit base class constructor calls and also helps us to correctly eliminate composed objects from our inheritance. In a case where a standalone constructor does not exist for a class, we depend only on the destructor. For classes with neither constructors nor destructors, we rely on the inference from OLA.

In addition, completion of main object initialization makes it possible to correctly differentiate constructors and destructors from other functions which contain vptr initialization as a result of inlining. This subsequently helps to avoid false positive inheritance inference thereby solving challenge **C2**.

By adopting these approaches, `DeClassifier` is able to address the limitations of existing solutions by improving recovery even with inlining, correctly differentiating inheritance from composition and assigning direction of inheritance as shown in Table 1

Complexity of the problem: `DeClassifier` attempts to solve a hard problem, directional class hierarchy recovery from optimized binaries. For this reason, we employ standard tools like BAP to lift binary to an intermediate representation. BAP is currently unable to analyze large binaries. In the current form of `DeClassifier` it inherits this limitation of BAP, however, if BAP is replaced with a more effective static analysis tool, we expect significant improvement in recovery. Also, `DeClassifier` assigns direction of inheritance when neither constructor nor destructor is available using OLA, however, if there is not enough information in the binary for OLA, no inheritance will be assigned for the classes involved.

4 DECLASSIFIER

We developed `DeClassifier`, a class inheritance inference engine that employs static analysis to reconstruct class hierarchy from optimized C++ binaries. We describe the phases present in `DeClassifier` in the following subsections.

4.1 VTable Extraction and Grouping

Complete object VTables consist of all the VTables that belong to a class. They provide an unlabeled unique representation for each polymorphic class in a binary. Like other solutions [6, 19], we treat complete object VTables as analogous to polymorphic classes and they form nodes in the CHT generated by `DeClassifier`.

Much work has been done on extracting VTables from the binary [9, 20, 29]. Vptrs are scattered throughout the text region of the binary as immediate values. Typically, they get written into locations in an object by constructors and destructors during object initialization. So we scan the text section to recover all immediate values which point to read-only sections of memory since VTables are stored in the read-only section to prevent VTable injection attacks. The well-defined nature of VTables, particularly the existence of mandatory fields [1] provides us with a signature to filter out recovered immediate values which point to read-only section but are not VTables, for instance, jump tables.

The recovered list of VTables contains all primary and secondary VTables where one or more of them make the complete object VTables for a single polymorphic class. Therefore, from the current list of VTables, we need to construct another list which comprises of only complete object VTables. To achieve this, we merge primary VTables with their corresponding set of secondary VTables, with each item being represented by the primary VTable address.

All VTables belonging to a class are laid out contiguously starting from the primary VTable which has an offsetToTop of 0. All the secondary VTables have a non-zero offsetToTop. Given a set of VTables, we first sort them in increasing order of addresses. Then, we merge a primary VTable with all secondary VTables immediately following it. This process gives us a set of complete object VTables.

4.2 Correct identification of Constructors and Destructors

Constructor and destructor calls within actual constructors and destructors are those that guarantee inheritance. Functions containing inlined vptr initialization can contain multiple such initializations for different unrelated classes, therefore using the information within them will result in an imprecise class hierarchy.

Constructors and destructors initialize the vptrs of the classes they belong to, among other operations they perform. The primary vptr of the class must be eventually written into the first entry of the object, before or after vptr of base classes are written, depending on whether a destructor or constructor is being considered. To do this, the object address gets passed, usually as the first argument. Therefore, we scan functions for primary vptr write to zero offset from the object address. We lift the binary to BAP IR and construct use-def chains for each IR variable. Next, we recursively propagate the defines into uses until all IR instructions are a combination of defines corresponding to function inputs. At this point, if the IR instruction corresponding to vptr initialization writes to the memory location pointed to by the first argument (implicit `this` pointer), we infer the function to be a constructor or a destructor. The instruction that writes vptr is the point of COI.

Our analysis will correctly distinguish constructors and destructors from functions that inline a constructor or destructor since the object address must be adjusted in order to write the primary vptr in the case of the latter. This gives us the complete set of constructors and destructors. However, we are still left with the task of correctly differentiating between constructors and destructors so that we do not wrongly infer the derived class as the base or vice versa or include composed classes in the inheritance. As discussed in Section 2.2, the ordering of initialization of base and derived class vptrs are in the reverse order for constructors and destructors. We infer a function to be a destructor if one of the following is true:

(1) Destructors are mostly virtual, so they have entries in the VTables. We check if the function address exists in a VTable.
(2) Due to the use of destructors, for destructing objects, they call the delete operator. We also check if the function being verified calls the delete operator. A constructor will not call the delete operator.
(3) In cases where explicit calls are made to base class constructor, we check if the calls are made before vptrs are initialized.

All identified constructors and destructors are associated with the primary vptr they belong to. A constructor is associated with the last primary vptr written to the base address an object. This is because if base class constructors are inlined, their vptrs are written first. A destructor is associated with the first primary vptr written to the base address an object. Once we have the set of constructors and destructors, we perform constructor-destructor analysis.

4.3 Constructor-Destructor Analysis

The constructor of a derived class calls the constructors of its base classes (or inlines the base classes' vptr initialization) before initializing its own vptr(s). Since we have already identified valid

constructors in the previous step, we extract all calls to valid constructors that take place before the last write of primary vptr to zero offset of the object being constructed. For inlined base class constructors, we extract all complete object VTables initialized also before the last write of primary vptr. Composed classes get initialized only after the complete object VTable of the current class has been initialized, either through explicit constructor call or inlined vptr initialization. Therefore, we are able to correctly exclude composed classes from our class hierarchy.

In a destructor, the derived class' complete object VTable is first initialized, followed by calls to composed class' destructors (or composed class' vptr initialization) and finally, calls to base classes' destructors. For a destructor, the last primary vptr write to zero offset of the object does not demarcate between base and composed objects destruction. However, the number of secondary vptrs initialized gives us insight into where calls to destructors of base classes begin. The number of VTables (primary and secondary) a class has is equal to the total number of direct base classes it has. To correctly eliminate composed class destructors, we map each vptr initialized to each destructor call starting from the last call. Finally, we ignore other calls which do not have a corresponding vptr initialization.

For all base class constructor/destructor calls identified, we locate their associated complete object VTable and map them as the bases. In the case of inlined base class constructor/destructor, we directly map the inlined complete object VTable as the base. For the sake of space, we have included the algorithms used to analyze constructors and destructors in Appendix A

4.4 Object Layout Analysis

We perform object layout analysis on virtual member functions of a class. Particularly, we are interested in member functions that operate on the `this` pointer. Calls to these functions are explicit and cannot be inlined, as such, they are available in the binary.

Specifically, we perform coarse type inferencing and label the object with its member types. First, we convert the binary to BAP IR to perform static analysis. Next, we identify type-revealing instructions in the function (`jmp *ebx, mov rdi, rax; call printf`, etc.) and their corresponding IRs. We employ intra-procedural static analysis to identify the offsets within `this` pointer that the types map to. This approach is similar to the type inferencing performed by past efforts such as REWARDS [17]. As an end result, we obtain a type map for offsets within `this` pointer. In order for an inheritance relationship between two classes to be correct, types of member variables in the two classes at specific offsets must be congruent (compatible) to each other.

Next, we use overwrite analysis. Since there is no way to infer if an inlined vptr initialization belongs to a constructor or destructor, the order of overwrite cannot be used to infer direction of inheritance. Therefore, we use the result of OLA to decide direction of inheritance for relationships identified through overwrite analysis. We analyze specific attributes of an object as well as its complete object VTable. We consider type congruence and VTable size.

Minimum Object Size Analysis Analyzing the size of an object can be done either dynamically or statically. The dynamic analysis approach has two major challenges, 1) coverage and 2) how to compute the size of the stack and global objects. Objects are created in three major locations at runtime, heap, stack and global region of the memory. To create objects on the heap, the new operator must be called which can be hooked to get the size passed to malloc (that will be upper bound for the object size). However, size of stack objects pose a challenge in the sense that there is no difference between the stack pointer movement when memory is allocated for a local variable (e.g an integer variable) and for object creation.

In this work, we analyze object size statically to obtain a lower bound object size. With this approach, coverage is not a challenge and neither is the location of an object a challenge. Just like constructors and destructors, the first argument passed to member functions of a class is the object address. To access a member variable, a literal value is added to the object address (i.e. `this` pointer) to reach that variable. The maximum offset that can be added to the `this` pointer will always be less than the size of the object itself, which gives a lower bound for the object size. Identifying non virtual functions of a class is challenging, however, pointers to all virtual functions of a class reside in the complete object VTable for that class. For every complete object VTable identified, we analyze each virtual function it contains to obtain the maximum offset accessed from the `this` pointer. We associate this value to the complete object VTable as the lower bound for its object size.

VTable OffsetToTop Considering the case of multiple inheritance, the derived object consists of sub-objects of its base classes. OffsetToTop refers to the offset of a given base-in-derived object from the top of the derived object. The offsetToTop value for each base class is stored in the offsetToTop field of the VTable corresponding to that base class. As already mentioned, one of the operations performed within a constructor is calling the constructor of base classes. Before this call is made, the constructor adds the offsetToTop value for the given base class to the `this` pointer in order to reach the base class sub-object. Hence, we compare the offsetToTop value with the offset at the constructor call site for equality to conclude on inheritance between the two classes.

VTable Size We compute the VTable size of a class as the total number of virtual functions pointer, pure virtual functions and zero destructor entries (this exists only in the VTables of abstract classes) that it contain. We do not consider the complete object VTable in this case because relationships can be identified between a secondary VTable and a primary VTable. Therefore we ensure that only the associated VTable sizes are considered.

VTable sizes are strictly non-decreasing down a particular inheritance chain from base class to derived class. Hence, the sizes of two VTables found to be related provide indication of direction of inheritance.

4.5 Performing overwrite analysis

We analyze each function identified to contain inlined constructor or destructor, examining every VTable writes (both primary and secondary) that they perform. VTable pointers written to the same memory locations are grouped together as being related. In Marx, if overwrite analysis identifies two vptrs A and B to be related and also finds B and C to be related, the three vptrs A, B, and C are grouped to be in the same set even though A and C might not be

related. In this work, once we identify a relationship between A and B, we immediately use the result from OLA to decide the direction of inheritance and then continue building up the class hierarchy with subsequent relationships.

We locate vptr overwrites in two ways: either the object address passed to a known constructor or destructor is the location where a primary or secondary vptr is written, or multiple vptrs are written in the same memory location. These overwrites are considered at the function level. For the first case, we locate the primary vptr associated with the constructor or destruction that is being called.

With multiple inheritance, a secondary vptr will overwrite a primary VTable or vice verse. Therefore, we also locate the corresponding primary VTable of every secondary VTable in any identified group of related vptrs. We are able to locate corresponding primary VTable using VTable grouping.

4.6 Inaccuracies in static analysis

We rely on BAP to perform OLA and overwrite analysis, but it has been reported to have some limitations [14]; 1. inability to lift certain instructions and 2. bugs in how some instructions are translated. Even though the bugs do not affect our analysis based on the kind of instructions needed for our analysis, we also found some inaccuracies in its disassembly. The sequence of instructions generated by BAP, may violate the sequence in which they can be executed. For instance, it treats all conditional jump instructions as though they will be executed. The current form of `DeClassifier` inherits these limitations of BAP.

4.7 CHT Generation

In this phase, we build the complete Class Hierarchy Tree for a binary by combining the relationships identified during constructor-destructor analysis phase with those identified by overwrite analysis. Constructor-destructor analysis directly assigns direction to any relationship it recovers using the order of calls or vptr initialization. For relationships recovered through overwrite analysis, we use attributes obtained from OLA to assign direction of inheritance. We apply the following rules to infer inheritance:

- If the constructor of class A calls constructor B before completion of object construction, A inherits from B. Converse holds true for destructors.
- Class A inherits from Class B only if size of object A $\geq$ size of object B.
- Class A inherits from Class B only if size of VTable of class A $\geq$ size of VTable of class B.
- Class A inherits from Class B only if type of each member in B given by $this_B$ + offset (from OLA) is compatible with the type of corresponding member $this_A$ + offset in A.
- Class C is a secondary base of class A only if the `offsetToTop` value in secondary VTable of A is equal to the displacement that must be added to an object of A to reach the C sub-object in A (obtained through OLA). For example, in the running example, the C sub-object in D can be obtained by adding 16 to the base address of D, which is simply the `offsetToTop` value (-16) in secondary VTable of D.

5 EVALUATION

Our evaluation aims to answer the following questions:

- What is the precision and recall of the entire class hierarchy recovered by `DeClassifier`?
- How effective is our direction of inheritance assignment using OLA?
- How does `DeClassifier` outperform ctor-only analysis for recovering class hierarchy?

All binaries were compiled using gcc under O0 and O2 optimization. We do not consider O3 optimization in the main evaluation because in terms of inlining, similar binaries are produced under O2 and O3. However, we have included results for SPEC2006 binaries compiled with O3 optimization in the Appendix (see Appendix D). All analysis experiments were performed on Ubuntu 16.04 LTS running on Intel Core i7 3.60GHz with 32GB RAM. We did not evaluate Node and Mongodb because BAP was unable to analyze them. Mongodb is too large for it and there was a runtime error while analyzing Node. We reported this error to BAP team and they acknowledged it is a bug in BAP which would be worked on. All SPEC2006 benchmark programs with polymorphic classes were considered except for Astar and Namd. Astar has just one polymorphic class, there is no edge for comparison. Namd has 3 polymorphic classes, where two of the classes inherit from the third. However, when compiled under O2 optimization, the VTable of the third gets optimized out.

5.1 Ground Truth

We obtained the ground truth for standalone programs by compiling them with the `-fdump-class-hierarchy` option on GCC. This generates a .class file for each .cpp file with at least one polymorphic class. The .class file contains VTable layouts and the inheritance information. The 7 WX Widget programs in our test set are together in a single package, there is no way we could distinguish the classes that belong to each of the programs using the output of `-fdump-class-hierarchy`. Therefore, we compiled the package with the `-frtti` option and then analyzed the RTTI structures in each of the binaries to obtain the ground truth inheritance.

5.2 Precision and Recall

In order to measure the performance of `DeClassifier`, we evaluated precision P and Recall R of the class hierarchy recovered from each of the 16 binaries considered. Precision answers the question of what fraction of the class hierarchy recovered is correct and what fraction is wrong, while recall answers the question of what fraction of the ground truth class hierarchy has been recovered and what fraction is not recovered (see Appendix B for formulas).

Table 2 and Table 3 show the breakdown of classes based on the number of classes they inherit from under O0 and O2 optimization levels respectively. Table 4 and Table 5 show the precision and recall of ctor-only analysis, ctor-dtor analysis as well as ctor-dtor+OLA under O0 and O2 optimization levels respectively.

Recall for O0 binaries was computed using all found edges in the ground truth. However, it is done differently for O2 binaries. Due to optimization, some classes (VTables) get removed by the compiler and the fact that our tool does not recover such classes does not make it less effective since they are in fact not available in the binary. To ensure these classes do not influence the recall recorded,

Table 2: CHT recovery results for binaries compiled with gcc -O0. Column with "inh = 0" contains # of classes that do not inherit from any class, "inh = 1" contains # of classes that inherit from exactly 1 immediate base class, and "inh >1" contains # of classes that inherit from more than 1 immediate base classes.

Programs	Ground Truth				Analysis						
	#Classes	inh=0	inh=1	inh>1	#Classes	Ctor-only			Ctor-Dtor		
						inh = 0	inh = 1	inh >1	inh = 0	inh = 1	inher >1
libebml	27	5	22	0	26	14	12	0	7	19	0
libflac	18	8	10	0	18	12	6	0	8	10	0
libzmq	76	17	47	12	76	39	29	8	17	47	12
libwx_baseu	285	24	258	3	287	157	128	2	49	235	3
libwx_baseu_net	44	27	15	2	44	35	8	1	27	15	2
libwx_gtk2u_adv	266	150	114	2	266	199	67	0	146	117	3
libwx_gtk2u_aui	62	51	11	0	62	58	4	0	52	9	1
libwx_gtk2_core	683	220	445	18	683	408	263	12	242	419	22
libwx_gtk2u_html	138	66	70	2	138	104	34	0	65	71	2
libwx_gtk2u_xrc	122	110	12	0	122	116	6	0	111	11	0
Doxygen	974	208	670	96	944	462	417	65	222	629	93
Xalanc	975	317	643	15	968	472	486	10	308	646	14
DealII	884	34	846	4	689	431	256	2	47	639	3
Omnetpp	112	10	102	0	109	50	59	0	11	98	0
Soplex	29	8	20	1	29	18	11	0	8	20	1
Povray	32	11	21	0	29	16	13	0	12	17	0

Table 3: CHT recovery results for binaries compiled with gcc -O2. Column with "inh = 0" contains # of classes that do not inherit from any class, "inh = 1" contains # of classes that inherit from exactly 1 immediate base class, and "inh >1" contains # of classes that inherit from more than 1 immediate base classes.

Programs	#Classes	Ctor-only			Ctor-Dtor			Ctor-Dtor+OLA		
		inh = 0	inh = 1	inh >1	inh = 0	inh = 1	inher >1	inh = 0	inh = 1	inher >1
libebml	26	14	12	0	7	19	0	7	19	0
libflac	18	15	3	0	8	10	0	8	10	0
libzmq	63	35	24	4	23	37	4	22	40	2
libwx_baseu	262	233	29	0	171	91	0	153	108	1
libwx_baseu_net	43	37	6	0	29	14	0	29	14	0
libwx_gtk2u_adv	229	214	15	0	197	18	0	193	35	1
libwx_gtk2u_aui	59	57	2	0	57	2	0	54	5	0
libwx_gtk2_core	621	566	55	0	486	135	0	364	240	17
libwx_gtk2u_html	123	118	5	0	104	19	0	104	19	0
libwx_gtk2u_xrc	93	93	0	0	93	0	0	93	0	0
Doxygen	870	845	23	0	458	400	12	483	446	5
Xalanc	875	615	258	2	396	479	0	410	457	8
DealII	687	544	140	3	123	561	3	96	579	11
Omnetpp	105	69	36	0	36	69	0	8	91	5
Soplex	25	23	2	0	21	3	1	19	6	0
Povray	24	21	3	0	17	7	0	17	7	0

we identified them and removed edges that have them as either derived or base from the ground truth to compare with. Basically, for O2 binaries, we computed recall by comparing with a subset ground truth which is based on the classes found in the binary. Columns labeled "GT" and "Used" under "#Edges" in Table 5 show the number of edges in the overall ground truth and the number of edges after removing edges whose classes are not in the binary.

The average precision and recall of class hierarchy recovered by `DeClassifier` on O0 binaries are 97.4% and 94.5% for libraries and 99.8% and 90.6% for executables respectively. And on O2 binaries, it has an average precision and recall of 85.4% and 58.4% for libraries and 98.4% and 71.4% for executables respectively. `DeClassifier` was unable to recover any hierarchy for libwx_gtk2u_xrc under O2 optimization.

5.3 Effectiveness of Direction Assignment

In this subsection, we discuss the effectiveness of direction of inheritance assignment using OLA. As discussed in section 4.7, after identifying a relationship between two classes using overwrite analysis, we use OLA to assign the direction of inheritance. Table 6 shows the number of directions correctly assigned, the number wrongly assigned (i.e. assigning the derived as the base) and the number not assigned at all. We do not assign direction of inheritance between two classes whenever there is not enough information from OLA about those classes. On the average, directions of inheritance were correctly assigned to 93.6% of relationships identified, 1% were wrongly assigned and 5.4% were not assigned at all.

5.4 Comparison with Ctor-only Analysis

Table 4 and Table 5 show how recall significantly increases from ctor-only analysis, to ctor-dtor+OLA. Precision decreases slightly from ctor-only to ctor-dtor, but increases for ctor-dtor+OLA analysis. Such a decrease in precision is recorded because, for destructor analysis, vptrs are mapped to base class destructors starting from the last call. As a result, if a class has no base class, but has a template class, the template class will be wrongly identified as its base

Table 4: Precision(P) and Recall(R) of CHT generated by `DeClassifier` under O0 Optimization with GCC. Comparison was done with the overall ground truth

Programs	#Classes in GT	#Classes in Binary	Ctor-only		Ctor-Dtor	
			P(%)	R(%)	P(%)	R(%)
libebml	27	26	100	54.5	100	86.4
libflac	18	18	100	60	100	100
libzmq	76	76	95.7	59.2	97.3	96.1
libwx_baseu	285	287	99.2	49	98.3	89.8
libwx_baseu_net	44	44	100	52.6	100	100
libwx_gtk2u_adv	266	266	94.0	50	91.9	95.8
libwx_gtk2u_aui	62	62	100	44.4	90.9	90.9
libwx_gtk2_core	683	683	98.6	58	97.2	93.8
libwx_gtk2u_html	138	138	100	48.6	98.7	100
libwx_gtk2u_xrc	122	122	100	45.5	100	91.7
Average			**98.8**	**52.2**	**97.4**	**94.5**
Doxygen	974	944	99.8	63.4	98.9	93.5
Xalanc	975	968	100	70.4	100	97.5
DealII	874	689	95	28.9	99.8	75.3
Omnetpp	112	109	100	57.8	100	96.1
Soplex	29	29	100	50	100	100
Povray	32	29	100	61.9	100	81.0
Average			**99.13**	**55.4**	**99.8**	**90.6**

class. However, with overwrite analysis, we are able to see that no overwrite actually happens between the two vptrs involved. First, the combination of destructor and constructor significantly increased the recovery compared to constructors alone. Secondly, combining these with OLA helped to recover all other details in the binary that are unavailable from either constructor or destructor analysis due to optimization. For O0 binaries, the average recall increased from 52.2% to 94.5% for libraries and from 55.4% to 90.6% for executables for ctor-only and ctor-dtor analysis respectively. Since no optimization is performed on O0 binaries, overwrite analysis improved neither precision nor recall, for this reason, we did not include a different column for Ctor-Dtor+OLA. For O2 binaries, the tables show that recall increased from 25.1% to 48.6% to 58.4% for libraries and from 20.2% to 56.1% to 71.4% for executables for Ctor only, Ctor-Dtor and Ctor-Dtor+OLA respectively.

6 DISCUSSION

6.1 Falses—Root Cause Analysis

Falses from BAP's IR This was mentioned in Section 4.6. In the disassembly produced by BAP for the snippet in 2, the sequence of instructions after 3 is 5, 6, 4. But there is no actual branch instruction to go back to 4 if the jump is executed. At 6, the content of rbx is stored back in rdi, which is the same location where 0xEDFDA8 is stored at 2. As a result, the rdi value passed to the destructor of InheritedMemberInfoContext::Private is the same as the location where 0xEDFDA8 is written. Therefore our overwrite analysis identifies 0xEDFDA8 and the primary vptr of InheritedMemberInfoContext::Private as being related whereas they are not. If this were dynamic analysis, instruction 4 will not be executed if the jump is executed.

Missing VTables With higher levels of optimization, the compiler removes the entire VTable of any class whose instance is not created. In Table 7, we counted the number of classes which constitute the edges not recovered by `DeClassifier` . Constructor-Destructor analysis and OLA ensure that all information present in the binary is recovered, however, if the information is not present in the binary, it cannot be recovered.

Listing 2: Disassembly from Doxygen

```
1 mov rbx, rdi ...
2 mov qword ptr [rdi], offset off_EDFDA8 ...
3 jnz short loc_5
4 call InheritedMemberInfoContext::Private::~Private() ...
5 ...
6 mov rdi, rbx ...
```

Optimized vptr Initialization We found two cases of optimized vptr initialization that the compiler performs. These initializations violate the typical construction/destruction behavior assumed by prior efforts.

a. Missing intermediate class initialization: The code snippet below shows the destructor of class SList<MemberList>in Doxygen.

```
1 Slist<MemberList>::~Slist() ...
2 mov rbx, rdi
3 mov [rdi], 0xb49d90 ...;Init vptr of Slist<MemberList>
4 mov rdi, rbx ...
5 call QGList::~QGList() ...;Call dtor of most base class
```

From the ground truth, the direct base class of SList<MemberList> is QList<MemberList>, which inherits from QGList. However, the snippet shows that call to the destructor of QList<MemberList> has been optimized and replaced with that of the most base class. In cases like this, we are unable to identify the direct base class. Table 8 shows the number of edges with missing intermediate base class. Note that for our evaluation, we neither consider these edges as false positives nor true positives.

b. Missing all bases: The code snippet below shows the only instance of SPxHarrisRT's construction in Soplex which inherits from SPxRatioTester.

```
1 int _cdecl main(): ...
2 call operator new(unsigned long) ...
3 #Init vptr of soplex::SPxHarrisRT
4 mov [rax], 0x457b50
```

In the binary, only the vptr of SPxHarrisRT gets written into the object address, without initialization of SPxRatioTester's vptr. Other derived classes of SPxRatioTester were initialized similarly. We found such cases are common and lead to inference inaccuracy.

7 RELATED WORK

Multiple prior C++ binary-level solutions have recovered semantic information from a binary [5, 6, 9, 19, 20, 26, 29]. However, VCI [6] and Marx [19] are the most recent and relevant tools closest to our work. VCI uses ctor-only analysis to reconstruct the class hierarchy of a program. It handles constructor inlining by relaxing the requirement that the vptr is written into the first argument (implicit `this` pointer) passed to the function being analyzed. This results in incorrect identification of functions as constructors, which subsequently results in false inheritance inference.

Andre et al. [19] presented Marx which reconstructs class inheritance from binary with heuristics. It uses overwrite analysis to group vptrs written in the same memory location into sets since only related vptrs get overwritten in the same memory location. Even though Marx is able to correctly group related classes into

Table 5: Precision(P) and Recall(R) of CHT generated by `DeClassifier` under O2 Optimization with GCC. Comparison was done with those edges in ground truth whose corresponding VTables were found in the binary

Program	#Classes		#Edges		Ctor-only		Ctor-Dtor		Ctor-Dtor+OLA	
	GT	Binary	GT	Used	P(%)	R(%)	P(%)	R(%)	P(%)	R(%)
libebml	27	26	22	22	100.0	54.6	100.0	86.4	100.0	86.4
libflac	18	18	10	10	100.0	30.0	100.0	100.0	100.0	100.0
libzmq	76	64	76	53	100.0	60.4	97.8	75.5	100.0	79.3
libwx_baseu	285	262	264	198	100.0	14.1	100.0	43.9	100.0	47.5
libwx_baseu_net	44	43	19	17	100.0	35.3	92.9	76.5	100.0	82.4
libwx_gtk2u_adv	266	229	118	83	100.0	18.1	88.2	18.1	91.4	38.6
libwx_gtk2u_aui	62	59	11	11	50.0	11.1	50.0	11.1	80.0	44.4
libwx_gtk2_core	683	621	481	293	95.1	13.3	94.7	30.4	93.8	61.4
libwx_gtk2u_html	138	123	74	36	100.0	13.9	88.9	44.4	89.5	47.2
libwx_gtk2u_xrc	122	102	12	-	0.0	0.0	0.0	0.0	0.0	0.0
Average					**84.5**	**25.1**	**81.3**	**48.6**	**85.4**	**58.4**
Doxygen	974	870	866	469	100.0	3.0	68.2	57.6	94.7	80.2
Xalanc	975	875	710	577	100.0	45.4	78.7	65.3	98.3	79.4
DealII	874	687	854	678	98.4	18.6	99.1	80.1	98.4	81.9
Omnetpp	112	105	102	97	100.0	22.7	100.0	58.8	98.7	78.4
Soplex	29	25	22	12	100.0	8.3	66.7	16.7	100.0	50.0
Povray	32	24	21	12	100.0	23.1	100.0	58.3	100.0	58.3
Average					**99.7**	**20.2**	**85.5**	**56.1**	**98.4**	**71.4**

Table 6: Number of direction of inheritance "correctly assigned", "wrongly assigned" and not "assigned" by OLA

Programs	Correctly assigned	Wrongly assigned	Not assigned
libebml	19	0	0
libflac	10	0	0
libzmq	42	0	2
libwx_baseu	94	0	0
libwx_baseu_net	14	0	0
libwx_gtk2u_adv	32	0	8
libwx_gtk2u_aui	4	0	0
libwx_gtk2_core	171	2	20
libwx_gtk2u_html	17	0	1
Doxygen	373	2	49
Xalanc	459	0	17
DealII	555	11	10
Omnetpp	73	4	0
Soplex	6	0	1
Povray	7	2	0

Table 7: Column 2 shows the total number of classes that make up the missing edges for O2 optimization, column 3 shows the number of those classes whose VTables were not found in the binary

Programs	#Polymorphic Classes missing edges (Col A)	#Classes in Col A without VTables in Bin
libzmq	20	13
libwx_baseu	88	27
libwx_baseu_net	6	3
libwx_gtk2u_adv	63	22
libwx_gtk2u_aui	5	5
libwx_gtk2u_core	198	68
libwx_gtk2u_html	13	7
Doxygen	178	100
Xalanc	158	84
Omnetpp	13	5
DealII	257	169
Soplex	3	2
Povray	16	4

sets, it does not reason about the direction of inheritance which significantly limits its application.

Table 8: Number of false positives recovered by ctor-only analysis, ctor-dtor analysis and ctor-dtor + OLA analysis for O2 binaries. "Total edg(es) not found" refers to edges in `DeClassifier`'s output but not in ground truth. "MIB" refers to Missing Immediate Base. "Act(ual) false positive" refers to actual fal(se) positive edges recovered by `DeClassifier`

Programs	Ctor-only			Ctor-Dtor			Ctor-Dtor+OLA		
	Total edg not found	Edg from MIB	Act Fal +ve	Total edg not found	Edg from MIB	Act Fal +ve	Total edg not found	Edg from MIB	Act Fal +ve
libebml	0	0	0	0	0	0	0	0	0
libflac	0	0	0	0	0	0	0	0	0
libzmq	0	0	0	1	0	1	0	0	0
libwx_baseu	1	1	0	4	4	0	15	15	0
libwx_baseu_net	0	0	0	1	0	1	0	0	0
libwx_gtk2u_adv	0	0	0	3	1	2	4	1	3
libwx_gtk2u_aui	1	0	1	1	0	1	1	0	1
libwx_gtk2_core	16	14	2	48	43	5	90	76	14
libwx_gtk2u_html	0	0	0	3	1	2	2	0	2
libwx_gtk2u_xrc	0	0	0	0	0	0	0	0	0
Doxygen	9	9	0	142	16	126	77	53	24
Xalanc	0	0	0	102	0	102	6	1	5
DealII	20	18	2	24	19	5	52	32	20
Omnetpp	14	14	0	12	12	0	24	21	3
Soplex	1	1	0	3	2	1	0	0	0
Povray	0	0	0	0	0	0	0	0	0

OBJDigger, proposed by Jin et al. [12], uses symbolic execution and inter-procedural data flow analysis to recover object instances, data members and methods of the same class. This is achieved by tracking the usage and propagation of the `this` pointer within and between functions. While the authors did not attempt to recover class inheritance, a method to achieve that was described. However, this can only identify primary base class since they assume that a base class will write its vptr only in the zero offset from the object address. A secondary base class will write to a positive non-zero offset from the object address but that was not accounted for.

Fokin et al. [7] presented SmartDec which recovers certain C++ specific language constructs statically. It attempts to recover classes and their inheritance, virtual and non-virtual member functions,

calls to virtual functions, exception raising and handling statements. Its limitation is the inability to differentiate inheritance from composition which results in false relationship inference.

Lego [27], proposed by Srinivasan et al., uses dynamic analysis to monitor objects allocated at runtime, the lifetime of those objects and methods invoked on them. Lego has two main challenges, 1. the precision of the class inheritance recoverable is limited to the portion of binary that gets invoked during executable, 2. it mistakes composition for inheritance when a class has no inheritance and the composed object is the first member of that class.

OOAnalyzer [23] groups methods into classes by combining binary analysis, symbolic analysis and Prolog-based reasoning. The paper explained that class and VTable size can be used to decide inheritance. However, this was not evaluated, therefore, there is no way to confirm the claim that OOAnalyzer can decide inheritance.

Rewards [17] is one of many (e.g., TIE [16], Laika [4]) data structure reverse engineering tools to infer type information from binaries. It uses dynamic analysis to recover syntax and semantics of data structures observed during execution. Rewards only attempts to infer primitive data types of variables and their semantics.

Prakash et. al. [20] proposed a CFI policy for virtual function calls in C++ binaries. The CFI policy is constructed by first recovering VTables and virtual call sites. With this information, virtual call site targets are restricted to virtual functions at offsets equal to that set at the call sites. We adopt its approach of VTable extraction.

8 CONCLUSION

Extracting class inheritance tree from optimized C++ code is hard, yet useful. We present `DeClassifier`, a static-analysis based inference engine that employs multiple novel techniques and infers significant amount of directed class inheritance tree from 16 C++ binaries compiled with gcc O0 and O2 options.

9 ACKNOWLEDGEMENT

We would like to thank anonymous reviewers and our shepherd Sang Kil Cha for their valuable feedback. This research was supported in part by Office of Naval Research Grant #N00014-17-1-2929, National Science Foundation Award #1566532, and DARPA award #81192. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Revision: 1.83. Itanium C++ ABI. http://refspecs.linuxbase.org/cxxabi-1.83.html. (Revision: 1.83).
[2] Dimitar Bounov, Rami Gökhan Kıcı, and Sorin Lerner. 2016. Protecting C++ dynamic dispatch through vtable interleaving. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*.
[3] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*.
[4] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T King. 2008. Digging for Data Structures.. In *Proceedings of 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*.
[5] David Dewey and Jonathon T. Giffin. 2012. Static detection of C++ vtable escape vulnerabilities in binary code.. In *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)*.
[6] 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 (ASIACCS'17)*.
[7] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*.
[8] Alexander Fokin, Katerina Troshina, and Alexander Chernov. 2010. Reconstruction of class hierarchies for decompilation of C++ programs. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*.
[9] Robert Gawlik and Thorsten Holz. 2014. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proceedings of 30th Annual Computer Security Applications Conference (ACSAC'14)*.
[10] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. ShrinkWrap: VTable Protection without Loose Ends. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*.
[11] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of 21st Annual Network and Distributed System Security Symposium (NDSS'14)*.
[12] 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 ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW'14)*.
[13] Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. Estimating Types in Binaries using Predictive Modeling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*.
[14] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing Intermediate Representations for Binary Analysis. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*.
[15] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*.
[16] Jong Hyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*.
[17] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*.
[18] Nathan Burow and Derrick McKee and Scott A. Carr and Mathias Payer. 2018. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*.
[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 24th Annual Network and Distributed System Security Symposium (NDSS'17)*.
[20] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*.
[21] Jang Ray. 1994. C++: Under the Hood. http://www.openrce.org/articles/files/jangrayhood.pdf. (1994).
[22] Paul Vincent Sabanal and Mark Vincent Yason. 2007. Reversing C++. *Blackhat Security Conference* (2007).
[23] 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 *ACM Conference on Computer and Communications Security (CCS'18)*.
[24] Igor Skochinsky. 2011. Practical C++ decompilation. (2011). https://archive.org/details/Recon_2011_Practical_Cpp_decompilation
[25] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of 23rd USENIX Security Symposium (USENIX Security'14)*.
[26] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, 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 IEEE Symposium on Security and Privacy (Oakland'16)*.
[27] Srinivasan Venkatesh and Thomas Reps. 2014. Recovery of Class Hierarchies and CompositionRelationships from Machine Code. In *23rd International Conference on Compiler Construction (CC'14)*.
[28] Chao Zhang, Scott A Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. 2016. VTrust: Regaining Trust on Virtual Calls. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*.
[29] Chao Zhang, Chengyu Song, Zhijie Kevin Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Defending Virtual Function Tables' Integrity. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*.

A ALGORITHMS FOR CONSTRUCTOR-DESTRUCTOR ANALYSIS

Below are the algorithms used for constructor-destructor analysis

Algorithm 1 CtorAnalysis analyzes constructors in C to identify the base classes of a class whose constructor is being analyzed.

```
1:  procedure CtorAnalysis(C)
2:      for each c in C do
3:          ownerPryVT ← getPrimaryVT(c)
4:          coi ← getCOI(c)
5:          for each instr in c do
6:              if isCall(instr)&&addressOf(instr) <= coi then
7:                  target ← getTarget(instr)
8:                  if targetinC then
9:                      basePryVT ← getPrimaryVT(target)
10:                     Base{ownerPryVT}.append(basePryVT)
11:                 end if
12:             end if
13:         end for
14:     end for
        return Base
15: end procedure
```

Algorithm 2 DtorAnalysis analyzes destructors in $\mathcal{D}$ to identify the base classes of a class whose destructor is being analyzed.

```
1:  procedure DtorAnalysis(D)
2:      for each d in D do
3:          ownerPryVT ← getPrimaryVT(d)
4:          coi ← getCOI(d)
5:          noVTs ← getNoOfVTs(d)
6:          for each instr in d do
7:              if isCall(instr)&&addressOf(instr) <= coi then
8:                  target ← getTarget(instr)
9:                  if targetinD then
10:                     allCalls.append(basePryVT)
11:                 end if
12:             end if
13:         end for
14:         for i = lenghtOf(allCalls) − 1 to 0 do
15:             basePryVT ← getPrimaryVT(allCalls[i])
16:             Base_ownerPryVT.append(basePryVT)
17:             noVTs = noVTs − getNoOfVTs(allCalls[i])
18:             if !(noVTs > 0) then
19:                 break
20:             end if
21:         end for
22:     end for
        return Base
23: end procedure
```

B FORMULAS FOR PRECISION AND RECALL

Precision is defined as follows:

$$P = \frac{TP}{TP + FP} \tag{1}$$

Recall is defined as follows:

$$R = \frac{TP}{TP + FN} \tag{2}$$

where *TP*, *FP* and *FN* refer to the number of derived-to-base edges recovered which match edges in the ground truth, number of edges which do not match any in the ground truth and number of edges in the ground truth not recovered.

C EFFECT OF OPTIMIZATION ON FUNCTION INLINING

Table 9 shows the results of the number of constructors and destructors remaining in the binary when compiled under O0 and O2 optimizations.

Table 9: Number of constructors and destructors present in O0 and O2 binaries.

Programs	O0			O2		
	Ctor	Dtor	Fns with inlined ctor/dtor	Ctor	Dtor	Fns with inlined ctor/dtor
Mongodb	3544	2063	40	725	1046	2638
Node	2930	2546	75	290	447	2520
Doxygen	1010	940	16	245	751	744
Soplex	29	25	4	10	12	3
Povray	47	24	9	40	19	20
Namd	4	4	0	0	1	2
Omnetpp	175	108	3	113	98	104
DealII	582	702	48	390	677	497
Xalanc	1391	958	309	954	771	1989
libebml	41	18	26	42	18	25
libflac++	45	18	0	29	18	5
libzmq	82	76	0	58	38	11
libwx_baseu_net	47	44	2	22	35	33
libwx_baseu	371	287	0	158	257	209
libwx_gtk2u_adv	310	259	12	48	155	240
libwx_gtk2u_aui	74	59	0	18	46	55
libwx_gtk2u_core	929	679	3	357	428	857
libwx_gtk2u_html	140	136	0	26	66	91
libwx_gtk2u_xrc	120	116	2	71	12	41

D RESULTS FOR SPEC 2006 BINARIES COMPILED WITH O3 OPTIMIZATION

Table 10 shows the average precision and recall of class hierarchy recovered by `DeClassifier` for SPEC2006 benchmark programs compiled under O3 optimization. We recorded a precision of 91.3% and a recall of 67.02%.

Table 10: Precision and Recall of CHT generated by `DeClassifier` for O3 Optimization with GCC. Comparison was done with those edges in ground truth whose corresponding VTables were found in the binary

Programs	#Classes		#Edges		Ctor-Dtor+OLA				
	GT	Binary	GT	Used	inh = 0	inh = 1	inh >1	precision	recall
Soplex	29	25	22	8	19	5	1	83.3	62.5
Povray	32	26	21	12	18	7	0	85.7	50.0
Omnetpp	112	105	102	97	8	96	1	98.4	63.9
Xalanc	975	876	710	570	410	449	13	93.5	77.9
DealII	874	722	854	652	117	572	12	95.5	80.8
Average								91.3	67.0