Recovering C++ Objects From Binaries
Using Inter-Procedural Data-Flow Analysis

Wesley Jin
CMU
wesleyj@andrew.cmu.edu

Cory Cohen
CERT
cfc@cert.org

Jeffrey Gennari
CERT
jsg@cert.org

Charles Hines
CERT
hines@cert.org

Sagar Chaki
SEI
chaki@sei.cmu.edu

Arie Gurfinkel
SEI
arie@sei.cmu.edu

Jeffrey Havrilla
CERT
jsh@cert.org

Priya Narasimhan
CMU
priya@cs.cmu.edu

Abstract
Object-oriented programming complicates the already difficult task of reverse engineering software, and is being used increasingly by malware authors. Unlike traditional procedural-style code, reverse engineers must understand the complex interactions between object-oriented methods and the shared data structures with which they operate on, a tedious manual process.

In this paper, we present a static approach that uses symbolic execution and inter-procedural data flow analysis to discover object instances, data members, and methods of a common class. The key idea behind our work is to track the propagation and usage of a unique object instance reference, called a this pointer. Our goal is to help malware reverse engineers to understand how classes are laid out and to identify their methods. We have implemented our approach in a tool called OB DIGGER, which produced encouraging results when validated on real-world malware samples.

1. Introduction
As malware grows in sophistication, analysts and reverse engineers are increasingly encountering samples written in code following the object-oriented (OO) programming model. For those tasked with analyzing these programs, recovering class information is an essential but painstaking process. Analysts are often forced to resort to slow and manual analysis of a large number of methods and data structures.

Programs that follow a traditional, procedural-based programming model are typically arranged around functions with well-defined boundaries, inputs, and outputs. They use structures that support limited operations with simple relationships (e.g., C-style structs). The clear relationships between procedures make it relatively easy to recover control and data flow even after compilation.

Conversely, the OO programming model is organized around data structures (i.e. C++ objects) with complex relationships and interactions. While C++ data structures are easily recognizable in source code, compilation hides them behind sets of methods with no obvious organization or relevance to one another. Therefore, to reverse engineer an OO program, analysts must: (1) determine related methods that belong to the same class; and (2) understand how they interact.

To facilitate the recovery of object structures and relationships, we have developed an approach that leverages the use of the "this pointer" (hereafter ThisPtr), a reference assigned to each unique (up to allocation instruction address) object instance. Specifically, we use symbolic execution [11] and static inter-procedural data-flow analysis [9] to track individual ThisPtr propagation and usage between and within functions.

Although previous authors, most notably Fokin et. al [7][6] and Sabanal et. al [18], have used ThisPtr tracking for OO reverse engineering purposes, our work is distinct for two reasons: 1) We document heuristics for identifying object-oriented methods and structures expressed as data-flow patterns, which can be detected in an automated way. Although patterns may vary from compiler to compiler or from one object to another, the key idea is that each variant may be captured as a unique pattern. 2) Our approach relies only on static analysis and symbolic execution. Thus, it has the ability to recover object-oriented artifacts that may not be created during execution.

We have implemented our approach on top of the ROSE analysis framework [15][17]. ROSE provides an infrastructure for disassembly, symbolic execution, control flow analysis, and data flow analysis. Our tool, OB DIGGER, aggregates data from object instances created throughout a binary, compiled with Microsoft's Visual Studio C++ (MSVC). It records potential constructors, data members and methods. Tests against open-source programs, compiled with MSVC, and real-world malware samples indicate that we are able
to recover this information reasonably accurately. Although our current implementation is specific to MSVC, our approach could be extended to others as well.

In summary, the contributions of this paper are:

• We present a purely static approach that uses symbolic execution and inter-procedural data-flow analysis to track object references in binaries, produced by MSVC.

• We provide an implementation capable of analyzing a binary and producing a list of (1) potential constructors or builders; (2) methods; (3) data members.

• We present techniques for detecting inheritance relationships and embedded objects.

• We demonstrate, through experimentation on both open-source programs and closed source malware, that the techniques described in this paper can be practically applied for reverse engineering OO code.

The remainder of this paper is organized as follows. In Sec. 2, we provide a brief overview of C++ object internals. In Sec. 3, we formalize our goals and constraints. In Sec. 4, we provide definitions for data structures crucial to our approach. In Sec. 5, we describe our approach. In Sec. 6, we describe experiments conducted using our implementation. In Sec. 6.1.2, we describe limitations in our current work and our plans to address them in the future. In Sec. 7, we review related work. In Sec. 8, we conclude.

2. Implementing Object Oriented C++ Features

In this section, we provide a basic overview of object-oriented C++ concepts. For a more detailed discussion, we direct interested readers to Gray [8]. Although this paper focuses on code following MSVC’s __thiscall convention, objects produced by other compilers follow similar patterns. Our discussion focuses on the example presented in Fig. 1.

When objects are created, the compiler allocates space in memory for each class instance. The amount of space allocated is based on the number and size of data members and possibly padding for alignment. Fig. 2 illustrates the layout of this memory region for Sub, Add and Add1, as generated by MSVC. Every instantiated object is referenced by a pointer to its start in memory; this reference is commonly referred to the ThisPtr.

The ThisPtr is maintained for the lifetime of the object. It is passed amongst methods, and is used for data member accesses and to make virtual function calls. For example, suppose that tPtr is a ThisPtr to an instance of Add1. The memory dereference [tPtr+12] points at variable one (see Fig. 2), and [tPtr+8] points at the embedded Sub object inherited from Add.

Only one instance of a function is created by a compiler. However, many instances of a C++ class may exist. Therefore, the ThisPtr allows for operations on the data members within a particular object instance. In MSVC, the ThisPtr is typically passed as a hidden parameter to OO functions via the EXC register in accordance with the __thiscall calling convention (more on this later). For example, the sum() method, depicted in Fig. 3, implements the __thiscall calling convention. At 0x401120, the ThisPtr is retrieved from EXC. At 0x40112A, the immediate value 1 is moved into the memory location corresponding to ThisPtr plus 12. This memory location corresponds to the offset of the Add1 class member variable one. Therefore, the instruction at 0x40112A corresponds to the high-level assignment one = 1.

Inheritance manifests in memory layouts by embedding an object of each superclass inside the object of the subclass (see Fig. 2).

---

1 This output is generated using the -d1reportAllClassLayout flag.
class Sub size(4):
|---
| 0 | c
|---
class Add size(12):
|---
| 0 | x
| 4 | y
| 8 | Sub b
|---
class Add1 size(16):
|---
| 0 | x
| 4 | y
| 8 | Sub b
|---

Figure 2. Class layouts for Sub, Add, and Add1.

0x401120: mov [esp+4], ecx
0x401125: mov eax, [esp+4]
0x40112A: mov dword ptr [eax+12], 1
0x401131: mov eax, [esp+4]
0x401136: mov eax, [eax]
0x401138: mov ecx, [esp+4]
0x40113D: add eax, [ecx+12]
0x401140: ret

Figure 3. Assembly Code for sum() in Add1.

0x402300: mov eax, [ecx]
0x402305: mov eax, [eax+4]
0x40230A: call eax

Figure 4. Virtual function call example.

3. Problem Statement

Given a binary executable compiled from C++ source code without debugging information, recover the following:

- Unique constructors and builder methods for classes instantiated
- Methods associated with object instances
- Location and size of data members used in these methods.

Goals and Assumptions. The goal of this work is to expedite the recovery of object-oriented structures in compiled executables. We aim to aid program understanding such that reverse engineers and malware analysts are able to quickly identify when objects and their data members are being used.

However, we do not seek to recover the original source code of classes for two reasons. First, recovering source might not always be possible, because compilation is not an injective mapping. Different sources can be compiled to produce the same binary, so identifying the original source code is impossible in general. Second, from the malware analysts’ point of view, it is more important to understand the details of the compiled code (e.g., method relationships and class layouts) than high-level abstractions.

4. Definitions

In this section, we review basic definitions and abstractions from data-flow analysis, used in the next section. For additional information, we direct the reader to work by Kiss, Jász, and Gyimóthy [12].

A computer system can be defined as $C = (P, M, R)$, where $P$ is a program; $M$ and $R$ are memory locations and registers that are available for use by $P$. Each program is composed of a set of functions, $F$, which can be further divided into sequences of instructions (i.e., $\forall f \in F, f = <i_0, i_1, i_2, ...>$). Let $I$ be the set of instructions, and $V$ be the set of values they manipulate.

Instructions read from and write to parts of $M$ and $R$. Let $Use : I \mapsto 2^{(V \times (M \cup R))}$ be a mapping such that $Use(i)$ is the set of all pairs $\langle v, a \rangle$, where $v$ is the value read by $i$ and $a$ is either a memory address in $M$ or a register in $R$ that stores $v$:

$$Use(i) = \{ \langle v, a \rangle \mid a \in M \cup R, i \overset{v}{\rightarrow} a \}$$

Simply stated, $Use(i)$ is a data structure that maps instructions to values read in particular registers and memory locations. Similarly, let $Def : I \mapsto 2^{(V \times (M \cup R))}$ be a mapping between an instruction and the locations it writes to:

$$Def(i) = \{ \langle v, a \rangle \mid a \in M \cup R, i \overset{v}{\leftarrow} a \}$$

An instruction is said to depend on another instruction, if it reads/uses a value that has been set by the other. We define the function $DepOn$ to be a mapping between an instruction $i$ and the set of triplets $\langle v, a, j \rangle$, where $v$ is the value written to the register or memory location $a$ by $j$.

$$DepOn(i) = \{ \langle v, a, j \rangle \mid \langle v, a \rangle \in Use(i) \land Def(j) \}$$

Simply stated, $DepOn$ adds to the data provided by $Use$ by providing information about the instruction responsible for defining the value that was read. Alternatively, the first instruction is said to be a dependent of the second. $DepOf$ is the inverse of $DepOn$:

$$DepOf(j) = \{ \langle v, a, i \rangle \mid \langle v, a \rangle \in Use(i) \land Def(j) \}$$

Finally, we define the notion of data flow order. Consider a function $X$ that calls three others: $A$, $B$, and $C$, such that $B$ calls $D$ just before returning. Furthermore, suppose that $X$ passes each method the value $v$. Written in flow order, $[A, B, D, C]$, implies that 1) $A, B, D$ and $C$ contain instructions that all use $v$ (i.e., instructions in $A, B, C$ and $D$ share a reaching definition of $v$), 2) Instructions in $A$ dominate the instructions in $B, D$ and $C$. Instructions in $B$ dominate those in $C$ and $D$, and so forth. (i.e., given a control-flow graph containing $X, A, B, C$ and $D$, and taking the first instruction in $X$ as the starting node, every path to the start from the instructions in $B, C$ and $D$ must go through the instructions in $A$. Every path from the instructions in $C$ and $D$ must go through those in $A$ and $B$, and so forth.)

5. Approach

Our approach consists of a preliminary stage and iterative analysis passes. We begin by disassembling binaries using the ROSE framework, and use data and control flow analysis to build the Use, Def, DepOn and DepOf maps. These data structures are then used to identify object-oriented structures. ROSE provides the x86 instruction semantics and symbolic emulation infrastructure required for this analysis.

The key idea behind identifying OO structures is to track the propagation of unique (up to allocation sites) ThisPtr instances within and between functions. We begin by identifying the set of functions $(F_M)$ that possibly follow the __thiscall convention. Next, using heuristics about known heap allocation functions such as the new() operator and stack allocation patterns, we identify
5.1 Data and Control Flow Analysis

To construct the four maps described above, we implemented the well-known work-list algorithm \[3, 10, 12\] for data-flow analysis. Our algorithm is shown in Procedure buildDependencies(). It maintains a list of symbolic expressions (called states) that capture the contents of registers and memory after each basic block\(^2\) is executed. For each basic block B, and for each instruction i of B in flow order, the algorithm: (i) symbolically executes i; (ii) updates the register and memory contents of B’s state with the result r; and (iii) adds i to the list of “modifiers” of r. This list records the addresses of all instructions that have contributed to the value up to this point. For example, processing the instruction add \[eax\] , 5, located at address Ox00405630, updates the memory contents at the address pointed to by EAX with 5, and adds Ox00405630 to the value’s list of modifiers. Therefore, when a different instruction reading this same memory location is processed later (i.e., cmp [EAX], 0), a dependency relationship with the add is established by reading the list of modifiers.

The state of each basic block, before any instructions are executed, is composed of the ‘merged’ states of each of the block’s predecessors. In more detail, if control-flow can reach a basic block from multiple locations, the contents of registers and memory at block entry may have different symbolic values and modifiers, depending on the specific path taken. Thus, the merged state combines the information from each possible entry path by performing a union across all possible entry states. Explicitly, if the contents of a register or memory location is the same in two different entry states, the symbolic value for that location in the merged state is the same. If they are different, the merged state reflects that the value is unknown, and the resulting list of modifiers is the combination of the lists from each entry state.

The state of each basic block, after all instructions are executed, is compared with the its previous state in states. If any of the registers or memory contents have changed, states is updated and all the block’s successors (those that the block can flow into) are marked for processing. The algorithm terminates when the state of all blocks stop changing.

5.2 Identifying __fastcall Functions

Most methods follow the __fastcall calling convention. When identifying data members and inheritance, we restrict ourselves to such functions, and thus our first step is to identify them. Note that the steps outlined here are not precise enough to distinguish between __fastcall and some instances of __fastcall3 (i.e., a more complete algorithm would also need to verify that EDX is not being used to pass parameters.) However, it is a cheap way to eliminate many functions that cannot be methods from further analysis, thereby improving the overall efficiency of our approach.

A key trait of __fastcall in MSVC is ThisPtr is passed as a parameter in the ECX register\(^3\). Exploiting this feature, we find __fastcall methods as follows:

\(^2\)A sequence of instructions with one entry and one exit.

\(^3\)http://msdn.microsoft.com/library/ex8tkfbw(v=vs.80).aspx

Procedure buildDependencies()

| Input: Func: A binary function composed of assembly instructions. |
| Input: EntryState: Symbolic state of system, storing register and memory contents, upon function entry |
| Result: Uses, Defs, DepsOn and DepsOf are populated for each instruction |

```plaintext
foreach block ∈ getBasicBlocks(Func) do
    states[block] ← initSymbolicState();
    queue[block] ← true;
    changed ← true;
    while changed do
        foreach block ∈ getBasicBlocks(Func) do
            if queue[block] then
                if isFirstBlock(block) then
                    curstate ← EntryState;
                    else
                        foreach pred ∈ getPredecessorBlocks(block) do
                            curstate ← mergeStates(curstate, states[pred]);
                        end
                        foreach instr ∈ getInstructions(block) do
                            curstate ← symbolicExec(instr, curstate);
                            if not regsAndMemEqual(curstate, states[block]) then
                                changed ← true;
                            end
                            foreach successor ∈ getSuccessorBlocks(block) do
                                if changed then
                                    queue[successor] ← true;
                                end
                                states[block] ← curstate;
                                queue[block] ← false;
                        end
                        if changed then
                            changed ← true;
                        else
                            changed ← false;
                        end
                    end
                end
            end
        end
    changed ← false;
end
```

We examine each method within a binary, \( f ∈ F \), and look for those that contain instructions that use ECX, whose value has been defined externally to the function. We examine DepOn and look for an instruction, \( i \), that maps to the tuple \((\ast, ECX, j)\), where \( j \) is an instruction that belongs to a different function than \( i \), and \( \ast \) matches an arbitrary value.) Therefore, the set of methods following __fastcall is:

\[ F_M \leftarrow \{ f \mid \exists i \in f, \exists j \notin f, (\ast, ECX, j) \in \text{DepOn}(i) \} \]

Our algorithm for identifying __fastcall methods is shown in Procedure findThisCall(). It generates a set containing pairs, each containing the __fastcall method, and the first instruction within that method to read ECX. In the rest of this paper, by a __fastcall method, we mean a method identified by findThisCall().
5.3 Identifying Object Instances and Methods

Once potential \_thiscall methods have been identified, the next step is to group them into object instances by finding those that share a common ThisPtr. Recall, the ThisPtr is a reference to an object instance. Object-oriented methods are passed these pointers, so that they know which object instance they are operating on, and they use the pointer to obtain member values and identify virtual methods. Therefore, we first identify a potential ThisPtr, which points to the stack or the heap. Next, from the data structures constructed earlier in buildDependencies(), we look for those object-oriented methods that have been passed this particular pointer in EAX.

Identifying ThisPtr creation follows a similar pattern for both the stack and the heap. Heap space is obtained using functions such as MSVC’s \_new() operator. Stack space is allocated upon function invocation in the function prologue. The lea instruction is often used subsequently to load references to portions of this space. In the remainder of this section, we describe how we track a heap-addressed ThisPtr to object-oriented methods. Tracking a stack-addressed ThisPtr is very similar, except the process begins at a lea instruction.

We are able to identify calls to \_new(), either by parsing the binary’s import section or from fingerprints/shashes of known implementation. Once a call to \_new() has been identified, we iterate through each \_thiscall method, and attempt to identify those that contain an instruction that uses this \_new()’s returned value.

To identify methods belonging to an object created on the heap, we do the following for each function that calls \_new():

1. We retrieve the ThisPtr by identifying the first instruction, \( j \), that reads EAX after a call to \_new(). The symbolic value of the ThisPtr is found from Use(\( j \)), corresponding to the pair \( \langle \text{ThisPtrFromNew}, \text{EAX} \rangle \).

2. We then iterate through each \_thiscall method, called in the same function that calls \_new(), looking for those that contain an instruction, \( i \), that reads EAX with a matching value.

Simply stated, we look for \_thiscall methods that are passed values of EAX that match the symbolic value in EAX, immediately following a call to \_new(). Or more formally:

\[
\{ f \in F_M \mid \exists i \in F_M, \langle \text{ThisPtrFromNew}, \text{EAX} \rangle \in \text{Use}(i) \land i \in f \}
\]

where \( f \) is a \_thiscall method, \( i \) is an instruction in \( f \), and \text{ThisPtrFromNew} is defined above. Also see Procedure findObjectMethodsFromNew().

Fig. 5 illustrates these concepts. The call to \_new() at 0x401008 allocates space on the heap. The ThisPtr, referred to this region, is returned in EAX and the instruction at 0x40100D saves it into a temporary variable. Next, this ThisPtr is transferred to the EAX register prior to the call to the constructor at 0x40121 and to the method call at 0x40133. Since constructor and method share a ThisPtr, they are methods belonging to the same class.

Procedure findReturnValueOfNew()

{\textbf{Input:} NewCaller: a function that calls \_new() \textbf{Input:} NewAddresses: a set of addresses of \_new() functions \textbf{Input:} Uses: the Uses map built by buildDependencies() \textbf{Result:} ThisPtr: the symbolic value returned by a \_new() call

1. \textbf{return} false;
2. \textbf{foreach} instr \in \text{getInstructions}(\text{NewCaller}) \textbf{do}
3. \hspace{1em}if found = false then
4. \hspace{2em}if \text{getInstructionType}(\text{instr}) = x86\_call then
5. \hspace{3em}if \text{getCallDest}(\text{instr}) \in \text{NewAddresses} then
6. \hspace{4em}\textbf{[found} \leftarrow \text{true}; \text{ // found the call to new}
7. \hspace{1em}\textbf{else}
8. \hspace{2em}\textbf{foreach} \langle \text{ThisPtr, alloc} \rangle \in \text{Uses[\text{instr}]} \textbf{do}
9. \hspace{3em}if alloc = EAX then
10. \hspace{4em}\textbf{return} ThisPtr; \text{ // return the symbolic value}
11. \textbf{return} failure; \text{ // not usually reached}

In a similar fashion, we identify objects created on the stack by identifying lea instructions, \( l \), that reference locally allocated stack space. The value of ThisPtr is found from the pair \( \langle \text{ThisPtr, REG} \rangle \in \text{Def}(l) \), where REG is the first parameter of the lea instruction. The pointer is tracked to \_thiscall methods in the same way as on the heap.

Identifying which methods are likely constructors is complicated by several factors. Constructors are required to return a ThisPtr, which distinguishes them from many, but not all conventional methods. If the class uses virtual functions, initialization
of the virtual function table pointers can be used to reliably identify constructors, but virtual functions are not present in all classes. Another common heuristic is that constructors are always called first after space is allocated for the object. This heuristic fails when compiler optimization has resulted in the constructor being inlined following the allocation. We chose to identify constructors as the first method called following allocation of the object if it returned the same ThisPtr that was passed as a parameter. This algorithm erroneously identifies some functions as constructors; for example, builder/factory methods can closely resemble constructors at the binary level. However, because these types of methods may be indistinguishable from constructors in the binary we have not counted this as an error. This heuristic also misses some legitimate constructors, for example in methods that construct other types of objects.

5.4 Data Members

Once related __thiscall methods have been associated with unique object instances, we process each one to retrieve data members. Recall that the ThisPtr points into the memory region allocated for an object. Therefore, by finding memory dereferences that use ThisPtr and extracting their offset into this area and size, we identify the location and width of the data member in the class layout.

Specifically, we identify the first instruction, j, in the function to read ECX. We retrieve the value of the ThisPtr from the pair ⟨thisPtr, ECX⟩ ∈ Use(j). We then iterate through all of the other instructions that dereference memory, i, looking for a pair ⟨*, thisPtr + offset⟩ ∈ Use(i). The algorithm is given more formally in Procedure findDataMembers(), which produces a mapping, MemberMap, between a __thiscall method and a set of data members discovered at a particular offset, represented by the pair ⟨offset, size⟩.

Fig. 6 illustrates the use of ThisPtr for accessing a data member. The ThisPtr is moved from ECX to EAX at 0x401104 and 0x401107. The data variable located at memory address ThisPtr plus 0xC is transferred to EAX at 0x40110A. Therefore, we determine that there is a data member at offset twelve in this class’ layout. Since the size of the dereference is 32-bit, we can assume that a variable, of at least that size, exists at that particular offset.

5.5 Virtual Function Tables

Objects that have virtual function tables initialize the memory at ThisPtr (zero offset) with the address of the table. This memory is used to identify the virtual function table addresses (i.e., ⟨vtableAddr, thisPtr⟩). We then identify calls made to entries within this table by examining the dependents of the mov instruction. In more detail, we find the set of instructions, Q:

\[ Q = \{ q | (\text{vtableAddr}, \text{thisPtr}, q) \in \text{DepOf}(i) \} \]

where Q contains the set of instructions that read the ThisPtr from the address initialized by the mov instruction. Using symbolic execution, we follow the flow of the pointer from this instruction to a call instruction. We record the branch target and offset of the call destination from ThisPtr as an entry at the given offset within the virtual table.

5.6 Inheritance and Embedded Objects

Although our current implementation does not fully support inheritance detection, we describe our current progress in this area.
Inheritance relationships can be determined by analyzing constructors. When a class inherits from a parent, the constructors of the subclass call the parent’s constructors. Specifically, the subclasses pass their ThisPtr to the parents’ constructors. In the case of single inheritance, the subclass constructor passes the ThisPtr directly (the memory address is exactly equal to ThisPtr with no offset). In the case of multiple inheritance, the subclass passes the pointer plus the offset at which the parent is located in the class layout.

Unfortunately, this behavior is also observed when an object contains embedded objects. Therefore, in order to distinguish between embedded objects and inheritance, we need additional discriminators. One reliable method would be to check if the subclass overwrites the virtual table address of its parent in its constructors. As mentioned earlier, classes in general, and the parent class in this case, are not required to contain virtual functions.

In summary, to identify inheritance relationships, we could: (1) retrieve all cross-references (calls out of the function) from constructors to other constructors; (2) compare the values of ECX at the beginning of each function; a constructor that calls other constructors that share a common ECX value (possibly plus some constant) indicates either an inheritance relationship or an embedded object; (3) check to see if the caller overwrites, the address passed to the called constructors. Recall, the pointer to the virtual table is typically located at offset zero within a class layout. Therefore, if a constructor writes to a memory address, that corresponds to a ThisPtr passed to another constructor, with a pointer to a new virtual table, we can label the other constructor as a parent. If we cannot find such an overwrite, it is possible that the constructor is instantiating an embedded object within the class. See Procedure lookForInheritance().

```
0x401104: mov [ebp-4], ecx
0x401107: mov ecx, [ebp-4]
0x40110A: call sub_4010C0
0x40110F: mov ecx, [ebp-4]
0x401111: add ecx, 10h
0x401112: mov ecx, [ebp-4]
0x401115: call sub_401080
0x40111A: mov eax, [ebp-4]
0x40111D: mov dword ptr [eax], 0x40816C
0x401123: mov ecx, [ebp-4]
0x401126: mov dword ptr [ecx+10h], 0x40817C
```

Figure 7. Example constructor with multiple inheritance.

In summary, there’s an open problem related to reliably distinguishing between embedded objects and multiple inheritance in the absence of virtual functions in the parent. Some of our remaining deficiencies stem from this difficulty, and we plan to continue investigating this problem in future work.

5.7 Object Instance Aggregation and Reporting

Our implementation aggregates data from object instances created throughout a binary. This information is grouped by unique constructor, and in some cases builder methods that return object instances that are largely indistinguishable from constructors. The list of all seen data members and methods associated with an object instance, produced by some constructor, are merged with that of another object instance, produced by the same constructor. Information from constructors known to belong to the same class, for example because they share a common virtual table, are also merged.

In this way we provide results to the analyst which are more useful than individual object instances and yet are not truly class definitions either. With more rigorous detection of inheritance and object embedding relationships these merged object instances should converge on complete class definitions although we do not claim that result in this work.

Fig. 7 shows part of a constructor that calls two other methods at 0x4010C0 and 0x401080. It passes its ThisPtr without any offset to the first call at 0x401107. It passes the second call at 0x40110A at 0x40112F and 0x401162, we observe that these same memory locations are overwritten with constants corresponding to two new virtual function table addresses. Therefore, we know that this constructor inherits from two other constructors, at 0x4010C0 and 0x401080, whose layouts are embedded at offset zero and sixteen of the class (see Fig. 2 for an example of single inheritance and layout embedding).

In summary, there’s an open problem related to reliably distinguishing between embedded objects and multiple inheritance in the absence of virtual functions in the parent. Some of our remaining deficiencies stem from this difficulty, and we plan to continue investigating this problem in future work.

Procedure lookForInheritance()

```
Input: Func: function identified as a constructor, member of ThisCalls
Input: Constructors: the set of all identified constructors
Input: ThisCalls: set of functions from findThisCall()
Input: Uses: the Uses map built by buildDependencies()
Input: Dests: the Dests map built by buildDependencies()
Result: Parents: set of parent/inherited constructors called by Func

Parents ← nil;
passed ← nil;
```

```
foreach instr ∈ getInstructions(Func) do
    // Find calls to other constructors
    if getInstructionType(instr) = x86_call then
        target ← getCallDest(instr);
    if target ∈ Constructors then
        // Get ThisPtr passed to each constructor
        foreach ⟨cfx, cxi⟩ ∈ ThisCalls do
            if target = cfx then
                foreach (symval, aloc) ∈ Uses[cxi] do
                    if aloc = ECX then
                        [Parents ← Parents ∪ cfx, symval];
                        passed ← passed ∪ ⟨cfx, symval⟩;
        // Look for mov instruction that overwrites location of a passed ThisPtr
        foreach instr ∈ getInstructions(Func) do
            if getInstructionType(instr) = x86_mov then
                foreach (symval, aloc) ∈ Dests[instr] do
                    if isMemWriteType(aloc) then
                        foreach (pxf, thisptr) ∈ passed do
                            if symval = thisptr then
                                [Parents ← Parents ∪ pxf;
```

```
return Parents;
```

Fig. 8 shows data about merged object instances from one of our experiments. Note that the actual output of our tool generates raw addresses. For illustrative purposes here, we have substituted the raw addresses with symbol information obtained from the compiler generated PDB files. This particular example shows correctly identify methods, members and virtual function information from the class XmlNode. However, it also illustrates a case in which our approach was unable to distinguish between an embedded object and an inheritance relationship. XmlNode inherits from XmlNode.
However, the XmlNode() and SetValue() methods of XmlNode were reported as methods of XmlText.

Constructor: __thi__XmlText::XmlText(bool)
Vtable: 0x7264
Vtable Contents:
Address: 0x7264 Pointer to Function @04035ae
Data Members:
Offset: 16 Size: 4
Offset: 20 Size: 4
Offset: 24 Size: 4
Offset: 28 Size: 4
Offset: 32 Size: 4
Offset: 36 Size: 4
Offset: 40 Size: 4
Offset: 44 Size: 4
Methods:
void __thi__XmlText::XmlText()
void __thi__XmlNode::XmlNode(XmlNode::NodeType)
void __thi__XmlNode::SetValue(char *)

Figure 8. Output of OBJDIGGER (with symbols substituted for addresses).

6. Experiments
To validate our approach, we conducted experiments on open-source packages, downloaded from SourceForge¹, and on real-world malware for which source is unavailable. We propose here a framework for evaluating such algorithms using a mixture of tool output, debugging information, and compiler generated class member layouts.

6.1 Open-source Tests
6.1.1 Methodology
The open-source tests were designed to evaluate the effectiveness of our approach given ground truth. The packages that we used were: The Lean Mean C++ Option Parser version 1.3, Light Pop3/SMTP Library, X3C C++ Plugin Framework version 1.0.2, PicoHttpD Library version 1.2, and Clmg Library version 1.0.5. Each program serves a different purpose, such as XML or math parsing, and includes test programs that exercise different parts of their respective APIs. We ran our tool on a binary from each library.

In these experiments, ground truth came from three sources: 1) a compiler layout produced by MSVC (as shown in Fig. 2) that contains information about the class layout and data members; 2) symbol information from compiler generated PDB files, which allows us to map function addresses to symbolic names (from which we can determine the classes in which they belong), and 3) source code of the test programs and libraries.

The results of our experiments are summarized in Table 1. We collected data in three categories for each test package:

1. # Unique classes found / # of unique classes. Using the symbol information from the PDB, we counted the number of unique classes instantated in the binary code. We excluded classes that were part of the standard compiler library. For the numerator, we counted those classes for which OBJDIGGER identified at least one instance of a constructor and associated methods and members.

2. # of methods found / # of methods in binary. We used the symbol information from the PDB to determine which methods were included in the binary. We counted a method as found by OBJDIGGER if it was associated with at least one instance of a constructor for the correct class. Note that inlined methods, and methods which were not included in binary are not present as symbols in the PDB file. We also excluded from the denominator cases where source code inspection confirmed that the methods were not in the control flow. This sometimes happen the compiler includes functions just because they were part of an object file.

3. # of data members found / # of data members in binary. Using the compiler layout information we compared the class members identified by OBJDIGGER to the members reported by the compiler. In certain circumstances we excluded from the denominator members that were known to have no uses in binary. This sometimes occurs when the compiler excludes the only function which accesses a member because the method was never called.

Our testing methodology, for each package, was as follows:

1. Compile test programs for the package. Generate layout information using the compiler and demangle class names using undname.
2. Run OBJDIGGER on each binary which reports method addresses and object layouts, without names.
3. Extract symbol data from PDB files using IDA Pro⁸ and demangle names using undname⁹. This maps function addresses to method names.
4. Correlate the function addresses from the OBJDIGGER output to the names in the symbol data. As can be seen in Fig. 8, the symbolic names specify the classes that particular methods belong to, and allows us to determine the validity of grouped methods.
5. Compare the discovered data members to those reported by the compiler, using the class name obtained from the symbol for the constructor.
6. Manually inspect the source code of each test program, excluding any methods or members as described in the previous section.

Table 1. Test results for open-source packages.

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
<th>Methods</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>PicoHttpD 1.2</td>
<td>8/9 (89%)</td>
<td>31/47 (66%)</td>
<td>18/25 (72%)</td>
</tr>
<tr>
<td>x3c 1.0.2</td>
<td>4/5 (80%)</td>
<td>21/24 (88%)</td>
<td>6/8 (75%)</td>
</tr>
<tr>
<td>Clmg 1.0.5</td>
<td>7/7 (100%)</td>
<td>61/83 (73%)</td>
<td>33/42 (79%)</td>
</tr>
<tr>
<td>OptionParser</td>
<td>10/10 (100%)</td>
<td>37/52 (71%)</td>
<td>33/35 (94%)</td>
</tr>
<tr>
<td>Light Pop3/SMTP</td>
<td>8/9 (89%)</td>
<td>29/35 (83%)</td>
<td>16/23 (70%)</td>
</tr>
</tbody>
</table>

6.1.2 Discussion
Table 1 lists the recall or true positive rate for OBJDIGGER. Method and member totals are summed across all classes. While the table does not explicitly list precision values, there were no false positives generated for this test set using the tool, so precision was 100% in all tests.

In each case, we verified that all identified methods and data members were correctly associated with the classes in which they actually belong by looking up their symbolic names.

¹http://www.sourceforge.net
⁸https://www.hex-rays.com/products/ida/
⁹Undname is a MSVC tool for demangling OO method names.
With regards to missed methods, OBJDIGGER was often able to identify many of these missed methods as following _thiscall, but was not able to associate these missed methods with a specific class. It was also able to group many of these missed methods with other found methods that shared the same TThisPtr. Unfortunately, some of those other found methods could be positively identified as a constructor. For example, in the case of PicoHttpD, the single missed class was created as a global variable. A memory address for a location in the .rdata section was passed to the constructor. However, currently, OBJDIGGER only checks for local stack addresses and space allocated by new(). Thus, even though OBJDIGGER correctly identified that this same pointer to the .rdata was passed as a TThisPtr in a couple of other methods (those that we missed), we didn’t report a new class instance being found or any of these associated methods. We chose this conservative approach to avoid over counting unique class instances. It is possible that these methods could have belonged to an object instance from a class that had already been identified, but created in another function.

When a constructor is not found, we are unable to associate any of the found members or methods in that object instance with a specific named class. This leads to a cascading effect where a single missed constructor negatively affects recall. Additionally, missed class methods also mean that any data members that were accessed inside of them were missed as well. This cascading effect is a fundamental challenge in analyzing OO code, since methods and data members are tied together to produce the object abstraction.

With regards to the missed methods and data members in Clmg and Light Pop3, we suspect that these omissions were due to implementations bugs and not limitations with the approach. Specifically, at the time of the experiments, OBJDIGGER had problems tracking objects that were passed as parameters to other methods. The tool also had problems identifying certain methods that were called indirectly, by dereferencing addresses within a class’s virtual table. We are currently working on addressing those issues.

A fundamental limitation of our approach is that we can only detect methods and members called and accessed by the program being analyzed, respectively. Our technique relies on grouping methods together by shared TThisPtr. Thus, if a program creates a class with methods the are never called by any instances of that class (or associated with a unique constructor belonging to that class), OBJDIGGER fails to detect these methods. Similarly, if a data member is never accessed (i.e., OBJDIGGER never observes a memory read or write to a particular location within a class layout), OBJDIGGER fails to detect this particular data member.

6.2 Closed-source Malware Case Study

Object-oriented malware presents many challenges to analysts. Understanding object structures can be critical for recovering functionality. To demonstrate how OBJDIGGER can aid with malware analysis we used it to help analyze a malware sample (file md5 019d3b95b261a582b163c77530aa72f on http://www.virustotal.com).

It is not uncommon for OO malware to encapsulate critical, malicious functionality in C++-style data structures. As a result, reverse engineering OO malware can be challenging because understanding program functionality may first require recovering C++ data structures.

Manualy recovering C++ data structures can be a tedious and error prone task, especially if done piecewise or in conjunction with trying to understand program functionality. OBJDIGGER automatically recovers object structures thereby streamlining analysis efforts. For example, in the sample, OBJDIGGER quickly identifies object instances and potential constructors.

Of the 585 functions within the sample, our tool identified nine object instances and their constructors, methods, data members, and virtual function tables. The analyst can then inspect this reduced set to determine each data structure’s relevance to the program.

```
0x401010 push 0xffffffFh
0x401012 push 41497Bh
0x401017 mov eax, large fs:0
0x40101D push eax
0x40101E mov large fs:0, esp
0x401025 sub esp, 0A8h
0x40102B push esi
0x40102C lea ecx, [esp+4]
0x401030 call sub_403000
0x401035 mov eax, [esp+C0h]
0x40103C mov ecx, [esp+BCh]
0x401043 push eax
0x401044 push ecx
0x401045 lea ecx, [esp+Ch]
0x401049 mov dword ptr [esp+BCh], 0
0x401054 call sub_401470
0x401059 lea ecx, [esp+4]
0x40105D mov esi, eax
0x40105F mov dword ptr [esp+B4h], 0xffffffFh
0x401066 call sub_401F20
0x40106F mov ecx, [esp+ACh]
0x401076 mov eax, esi
0x401078 pop esi
0x401079 mov large fs:0, ecx
0x401080 add esp, 0B4h
0x401086 ret
```

Figure 9. Main function of the malware sample.

Constructor: 403000
Vtable: 41647c
Vtable Contents:
Address: 41647c Pointer to Function @ 403030
Data Members:
Offset: 0x0 Size: 0x4
Offset: 0x8 Size: 0x4
Offset: 0x10 Size: 0x4
...Offset: 0xa0 Size: 0x4
Offset: 0xa4 Size: 0x4
Methods:
401470
401f20

Figure 10. OBJDIGGER output for the malware sample main function.

Fig. 9 shows the disassembly for the main function generated by IDA Pro. A cursory analysis of this function shows that it is a relatively simple routine containing three methods: sub_403000, sub_401470, and sub_401f20. Note that in Fig. 10, OBJDIGGER identified all three of these methods as related to one object, sub_403000 is the constructor and sub_401470 and sub_401f20 are class methods. With this information the analyst immediately has a sense that the malware’s functionality is organized around (at least) one object instantiated in the main function. Because significant parts of the program are encapsulated in this object understanding its internals is likely critical to determining program functionality. For instance, understanding the purpose of this object’s data members takes on greater importance because of the object’s usage in the malware. Notably, OBJDIGGER also provides information on class member offset and size, further simplifying analytical efforts.
In this scenario, the information provided by OBJDIGGER could be recovered manually, but this may take considerable effort. Automatically reasoning through C++ data structures saves time and frees the analyst to focus on questions that are more relevant to malware functionality.

7. Related Work

Sabanal et al. [18] provide a detailed discussion of recovering C++ data structures from binary code. In particular, they are the first to describe heuristics for recognizing C++ objects by monitoring the use of the ThisPtr in binary methods. Our work builds upon their ideas, and captures these heuristics as machine recognizable data-flow patterns. Additionally, our work goes one step further by tracking the propagation of ThisPtr between function to identify common data members and methods of classes. Jens Tröger and Cifuentes [21] pioneered similar use of dataflow analysis techniques applied to binaries to recover virtual function tables; however, their work relied on dynamic execution of code to resolve addresses of object methods.

Lee et al. [13], Balakrishnan and Reps [2], and Ramalingam et al. [16] are focused on variable/data type recovery in executable files. While type recovery is an important and related problem, our primary concern is to recover the class structure of objects.

Srinivasan et al. [20] propose a method that uses dynamic analysis to observe call relationships between methods to infer class hierarchy (similar to what we have done in a static context). However, their ability to recover class structure is limited to portions of a binary that actually run. Furthermore, since they are not tracking memory dereferences that use the ThisPtr, they do not recover data members.

Slowinska et al. [19] and Lin et al. [14] are focused on type discovery of variables using dynamic analysis. Although their work does not deal with object-oriented code directly, their method of tracking the use of memory locations to infer size and type is similar to the way we track memory dereferences, involving the ThisPtr, to infer size and offset of data members.

Adamantiadis [1] provides a detailed explanation of constructors, destructors and virtual function tables at the binary level. They give an example of reverse engineering an object-oriented C++ binary. However, the discussion does not propose an automated technique.

Dewey et al. [4] describe many techniques similar to the ones we use in our work. They specifically state though they are focused on analyzing known non-malicious code for a specific class of vulnerability. Our work is designed to be used explicitly for analyzing malicious software.

Fokin et al. [7] adopt an approach that appears to be very similar to ours, but provides less detail about the data flow analysis. Their earlier work [6] provides interesting insights about the aggregation of related object instances into classes.

8. Conclusion and Future Work

In this paper, we present a purely static approach for recovering object instances, data members and methods from x86 binaries, compiled using MSVC. We produced a tool, OBJDIGGER, which we tested against open-source software and real-world malware. A comparison of the output from the open-source tests against ground truth, generated by the compiler’s debug information, indicates that our technique can achieve its goal effectively. The tests against real-world malware demonstrate that our tool can aid in the malware reverse engineering process.

While our experiments demonstrate that our approach is viable, there is room for improvement. First, OBJDIGGER needs to be extended to recognize and recover objects instantiated on a global scope. We are currently exploring this direction, building on data-flow analysis techniques to reason through mechanics of global object creation and storage.

In certain object arrangements, inheritance and composition are hard to distinguish. Determining whether an embedded object is a parent or a member without using the presence of virtual function tables is an open problem. More work is needed to correctly identify this arrangement. Similarly, constructors and destructors can be difficult to distinguish under certain circumstances. OBJDIGGER needs to be extended to accurately identify destructors to enable better identification and tracking of object scope.

Advanced OO features of C++, such as virtual inheritance, are currently not supported. Virtual inheritance fundamentally changes the layout of objects in memory. The primary mechanism to implement virtual inheritance is the virtual base class table. The virtual base class table maintains offset to multiple parent classes to remove ambiguity possible with multiple inheritance. OBJDIGGER must be extended to correctly recognize and interpret these tables.

Further investigation is needed to fully understand the implications of compiler optimizations such as inlining of constructors, destructors, and other methods.

Finally, further experimentation is needed to determine to what extent OBJDIGGER can analyze non-MSVC generated binaries. Preliminary analysis suggests that compilers such as the GNU C++ Compiler use similar mechanisms to implement OO C++ features, but additional investigation is needed to determine what, if any nuances exist in different compilers. It might also be interesting to investigate what we can discover of OO patterns in other languages, such as Delphi, which analysts see frequently in the malware realm.

On a more practical note, the output of OBJDIGGER can certainly be improved to help the analyst more quickly see relationships between methods and objects (perhaps as a custom plugin to IDA Pro).

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. Carnegie Mellon®, CERT®, CERT Coordination Center® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM-0000440

References


