Reversing C++

As recent as a couple of years ago, reverse engineers can get by with just knowledge of C and assembly to reverse most applications. Now, due to the increasing use of C++ in malware as well as most moderns applications being written in C++, understanding the disassembly of C++ object oriented code is a must. This paper will attempt to fill that gap by discussing methods of manually identifying C++ concepts in the disassembly, how to automate the analysis, and tools we developed to enhance the disassembly based on the analysis done.

Paul Vincent Sabanal

Researcher, IBM Internet Security Systems X-Force R&D

Mark Vincent Yason

Researcher, IBM Internet Security Systems X-Force R&D

Table of Contents

I. Introduction and Motivation

As reverse engineers, it is important that we are able to understand C++ concepts as they are represented in disassemblies and of course, have a big picture idea on what are the major pieces (classes) of the C++ target and how these pieces relate together (class relationships). In order to achieve this understanding, the reverse engineer must able to (1) Identify the classes (2) Identify relationships between classes (3) Identify the class members. This paper attempts to provide the reader information on how to achieve these three goals. First, this paper discusses the manual approach on analyzing C++ targets in order to retrieve class information. Next, it discusses ways on how to automate these manual approaches.

Understanding C++ constructs in a disassembly is indeed a good skill to have, but what are our motivations behind learning this skill and writing this paper? The following are what motivated us in producing this paper:

1) **Increasing use of C++ code in malcode**
   Having experience as malcode analysts, there are cases in which the malcode we are trying to understand is written in C++. Loading the malcode in IDA and performing static analysis of virtual function calls is sometimes difficult because being an indirect call, it is not easy to determine where these calls will go. Some example of notorious malcodes that are written in C++ are Agobot, some variants of Mytob, we are also seeing some new malcodes developed in C++ from our honeypot.
2) **Most modern applications use C++**
   For large and complex applications and systems, C++ is one of the languages of choice. This means that for binary auditing, reversers expects that there are targets that are written in C++. Information about how C++ concepts are translated into binary and being able to extract high level information such as class relationships is beneficial.
3) **General lack of publicly available information regarding the subject of C++ reversing**
   We believe that being able to document the subject of C++ reversing and sharing it to fellow reverse engineers is a good thing. It is indeed not easy to gather information about this subject and there is only a handful of information that specifically focuses on it.

II. Manual Approach

This section introduces the manual approach of analyzing C++ binaries; it specifically focuses on identifying/extracting C++ classes and their corresponding members (variables, functions, constructors/destructors) and relationships. Note

A. Identifying C++ Binaries and Constructs

As a natural way to start, the reverser must first determine if a specific target is indeed a compiled C++ binary and is using C++ constructs. Below are some pertinent indications that the binary being analyzed is a C++ binary and is using C++ constructs.

1) **Heavy use of `ecx` (*this* ptr)**. One of the first things that a reverser may see is the heavy use of `ecx` (which is used as the *this* pointer). One place the reverser may see it is that it is being assigned a value just before a function is about to be called:

```
.text:004019E4 mov     ecx, esi
.text:004019E6         push    0BBh
.text:004019EB   call    sub_401120   ; Class member function
```

   Another place is if a function is using `ecx` without first initializing it, which suggests that this is a possible class member function:

```
.text:004010D0 sub_4010D0      proc near
.text:004010D0                 push    esi
.text:004010D1                 mov     esi, ecx
.text:004010DD                 mov     dword ptr [esi], offset off_40C0D0
.text:00401101                 mov     dword ptr [esi+4], 0BBh
.text:00401108                 call    sub_401EB0
.text:0040110D                 add     esp, 18h
.text:00401110                 pop     esi
.text:00401111                 retn
.text:00401111 sub_4010D0      endp
```

4) **Calling Convention**. Related to (1), Class member functions are called with the usual function parameters in the stack and with `ecx` pointing to the class's object (i.e. *this* pointer.). Here is an example of a class instantiation, in which the allocated class object (`eax`) will eventually be passed to `ecx` and then invocation of the constructor follows.

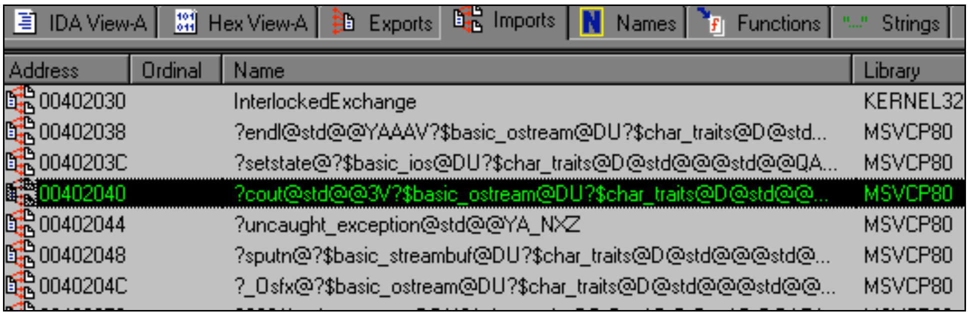
```
.text:00401994          push    0Ch
.text:00401996          call    ??2@YAPAXI@Z    ; operator new(uint)
.text:004019AB          mov     ecx, eax
:::
.text:004019AD          call    ClassA_ctor
```

Additionally, reversers will notice indirect function calls which are more likely virtual functions; it is of course, difficult to follow where these calls go without first knowing the actual class or running the code under a debugger. Consider the following virtual function call example:

```
.text:00401996 call     ??2@YAPAXI@Z    ; operator new(uint)
:::
.text:004019B2 mov      esi, eax
:::
.text:004019AD call     ClassA_ctor
:::
.text:004019FF mov      eax, [esi]     ;EAX = virtual function table
.text:00401A01          add     esp, 8
.text:00401A04          mov     ecx, esi
.text:00401A06          push    0CCh
.text:00401A0B          call    dword ptr [eax]
```

In this case, the reverser must first know where the virtual function table (vftable) of `ClassA` is located, and then determine the actual address of the function based from the list of functions listed in the vftable.

5) **STL Code and Imported DLLs**. Another way to determine if a sample is a C++ binary is if the target is using STL code, which can be determined via Imported functions or library signature identification such as IDA's FLIRT:



Address	Ordinal	Name	Library
00402030		InterlockedExchange	KERNEL32
00402038		?endl@std@@YAAAV?$basic_ostream@DU?$char_traits@D@std...	MSVCP80
0040203C		?setstate@?$basic_ios@DU?$char_traits@D@std@@@std@@QA...	MSVCP80
00402040		?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@...	MSVCP80
00402044		?uncaught_exception@std@@YA_NXZ	MSVCP80
00402048		?sputn@?$basic_streambuf@DU?$char_traits@D@std@@@std@...	MSVCP80
0040204C		?_Osfx@?$basic_ostream@DU?$char_traits@D@std@@@std@@...	MSVCP80

And the calls to STL code:

```
.text:00401201     mov     ecx, eax
.text:00401203     call
ds:?sputc@?$basic_streambuf@DU?$char_traits@D@std@@@std@@QAEHD@Z ;
std::basic_streambuf<char,std::char_traits<char>>::sputc(char)
```

Class Instance Layout

Before going any further, the reverser should also be familiar with how classes are laid out in memory. Let's start with a very simple class.

```
class Ex1
{
         int var1;
         int var2;
         char var3;
public:
         int get_var1();
};
```

The layout for this class will look like this:

```
class Ex1       size(12):
        +---
 0      | var1
 4      | var2
 8      | var3
        | <alignment member> (size=3)
        +---
```

Padding was added to the last member variable because it must align on a 4-byte boundary. In Visual C++, member variables are placed in the memory in the same order as they are declared.

What if the class contains virtual functions?

```
class Ex2
{
         int var1;
public:
         virtual int get_sum(int x, int y);
         virtual void reset_values();
};
```

Here's the class layout:

```
class Ex2       size(8):
        +---
 0      | {vfptr}
 4      | var1
        +---
```

Note that a pointer to the virtual functions table is added at the beginning of the layout. This table contains the address of virtual functions in the order they are declared. The virtual functions table for class Ex2 will look like this.

```
Ex2::$vftable@:
0       | &Ex2::get_sum
4       | &Ex2::reset_values
```

Now what if a class inherits from another class? Here's what happens when a class inherits from a single class i.e. single inheritance:

```
class Ex3: public Ex2
{
         int var1;
public:
         void get_values();
};
```

And the layout:

```
class Ex3       size(12):
        +---
        | +--- (base class Ex2)
 0      | | {vfptr}
 4      | | var1
        | +---
 8      | var1
        +---
```

As you can see, the layout of the derived class is simply appended to the layout of the base class. In the case of multiple inheritance, here's what happens:

```
class Ex4
{
         int var1;
         int var2;
public:
         virtual void func1();
         virtual void func2();
};
```

```
class Ex5: public Ex2, Ex4
{
         int var1;
public:
         void func1();
         virtual void v_ex5();

};


class Ex5       size(24):
        +---
        | +--- (base class Ex2)
 0      | | {vfptr}
 4      | | var1
        | +---
        | +--- (base class Ex4)
 8      | | {vfptr}
12      | | var1
16      | | var2
        | +---
20      | var1
        +---

Ex5::$vftable@Ex2@:
 0      | &Ex2::get_sum
 1      | &Ex2::reset_values
 2      | &Ex5::v_ex5

Ex5::$vftable@Ex4@:
        | -8
 0      | &Ex5::func1
 1      | &Ex4::func2
```

As you can see, a copy if each base class's instance data will be embedded in the derived class's instance, and each base class that contains virtual functions will have their own vftable. Take note that the fist base class shares a vftable with the current object. The current object's virtual functions will be appended at the end of the first base class's virtual functions list

B. Identifying Classes

After identifying C++ binaries, discussing some important C++ constructs and how a class instance is represented in memory, this part now present ways on identifying C++ classes used in the target. The methods discussed below only tries to determine what are the classes (i.e. the target has `ClassA`, `ClassB`, `ClassC`, etc). The next sections of this paper will discuss how to infer relationships between these classes and determine their members

1) Identifying Constructors/Destructors

To identify classes in the binary, we need to examine how objects of these classes are created. How their creation are implemented in the binary level can provide us with hints on identifying them in the disassembly.

1) **Global Object.** Global objects, as the name implies, are objects declared as global variables. Memory spaces for these objects are allocated at compile-time and are placed in the data segment of the binary. The constructor is implicitly called before main(), during C++ startup, and the destructor is called at program exit.

   To identify a possible global object, look for a function called with a pointer to a global variable as the *this* pointer. To locate the constructor and destructor, we have to examine cross-references to this global variable. Look for locations where this variable is passed as the *this* pointer to a function call. If this call lies between the path from program entry point and main(), it is probably the constructor.

2) **Local Object.** Local objects are objects that are declared as local variables. The scope of these objects are from the point of declaration until the block exit e.g. end of function, closing braces. Space the size of the object is allocated in the stack. The constructor is called at the point of object declaration, while the destructor is called at the end of the scope.

   A constructor for a local object can be identified if a function is called with a *this* pointer that points to an uninitialized stack variable. The destructor is the last function called with this *this* pointer in the same block where the constructor was called i.e. the block where the object was declared.

Here's an example:

```
.text:00401060 sub_401060      proc near
.text:00401060
.text:00401060 var_C           = dword ptr -0Ch
.text:00401060 var_8           = dword ptr -8
.text:00401060 var_4           = dword ptr -4
.text:00401060
…(some code)…
.text:004010A4         add     esp, 8
.text:004010A7         cmp     [ebp+var_4], 5
.text:004010AB         jle     short loc_4010CE
.text:004010AB
.text:004010AB { ← block begin
.text:004010AD         lea     ecx, [ebp+var_8] ; var_8 is uninitialized
.text:004010B0         call    sub_401000 ; constructor
.text:004010B5         mov     edx, [ebp+var_8]
.text:004010B8         push    edx
.text:004010B9         push    offset str->WithinIfX
.text:004010BE         call    sub_4010E4
.text:004010C3         add     esp, 8
.text:004010C6         lea     ecx, [ebp+var_8]
.text:004010C9         call    sub_401020 ; destructor
.text:004010CE } ← block end
.text:004010CE
.text:004010CE loc_4010CE:              ; CODE XREF: sub_401060+4B□j
.text:004010CE         mov     [ebp+var_C], 0
.text:004010D5         lea     ecx, [ebp+var_4]
.text:004010D8         call    sub_401020.
```

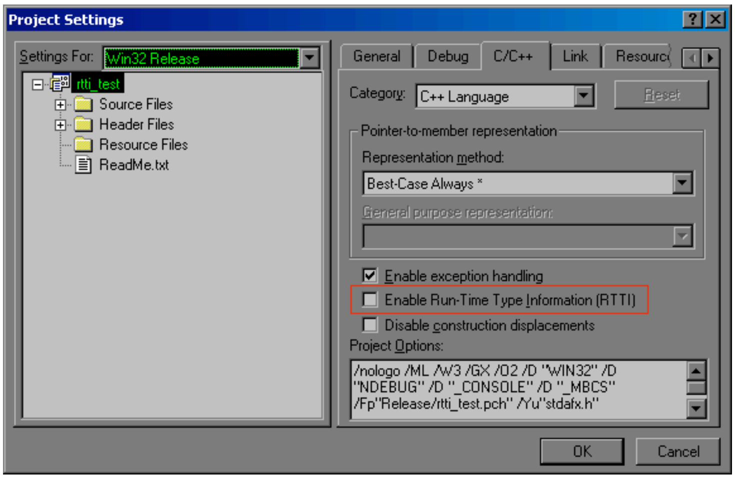
3) **Dynamically Allocated Object**. These objects are dynamically created via the new operator. The new operator is actually converted into a call to the new() function, followed by a call to the constructor, The new() function takes the size of the object as parameter, allocates memory of this size in the heap, then returns the address of this buffer. The returned address is then passed to the constructor as the this pointer. The destructor has to be invoked explicitly via the delete operator. The delete operator is converted into a call to the destructor, followed by a call to free to deallocate the memory allocated in the heap.

Here's an example:

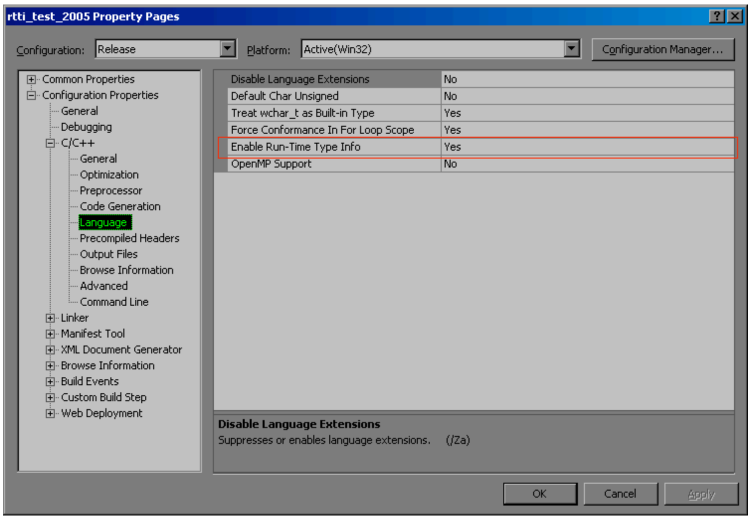
```
.text:0040103D _main          proc near
.text:0040103D argc            = dword ptr  8
.text:0040103D argv            = dword ptr  0Ch
.text:0040103D envp            = dword ptr  10h
.text:0040103D
.text:0040103D        push    esi
.text:0040103E        push    4               ; size_t
.text:00401040        call    ??2@YAPAXI@Z    ; operator new(uint)
.text:00401045        test    eax, eax ;eax = address of allocated
memory
.text:00401047        pop     ecx
.text:00401048        jz      short loc_401055
.text:0040104A        mov     ecx, eax
.text:0040104C        call    sub_401000 ; call to constructor
.text:00401051        mov     esi, eax
.text:00401053        jmp     short loc_401057
.text:00401055 loc_401055:                   ; CODE XREF: _main+B□j
.text:00401055                 xor     esi, esi
.text:00401057 loc_401057:                   ; CODE XREF: _main+16□j
.text:00401057        push    45h
.text:00401059        mov     ecx, esi
.text:0040105B        call    sub_401027
.text:00401060        test    esi, esi
.text:00401062        jz      short loc_401072
.text:00401064        mov     ecx, esi
.text:00401066        call    sub_40101B ; call to destructor
.text:0040106B        push    esi             ; void *
.text:0040106C        call    j__free ; call to free thunk function
.text:00401071        pop     ecx
.text:00401072 loc_401072:                   ; CODE XREF: _main+25□j
.text:00401072        xor     eax, eax
.text:00401074        pop     esi
.text:00401075        retn
.text:00401075 _main           endp
```

2) Polymorphic Class Identification via RTTI

Another way to identify classes, specifically *polymorphic classes* (classes with member virtual functions) is via Run-time Type Information (RTTI). RTTI is a mechanism in which the type of an object can be determined at runtime. This mechanism is the one being utilized by the `typeid` and `dynamic_cast` operator. Both these operators need information about the classes passed to them, such as class name and class hierarchy. In fact, the compiler will display a warning if these operators are used without enabling RTTI. By default, RTTI is disabled on MSVC 6.0.



On MSVC 2005, RTTI is enabled by default.



As a side note, there is a compiler switch that enables the MSVC compiler to generate class layout, the switch is `-d1reportAllClassLayout` this switch generates a `.layout` file which contains a wealth of information regarding the layout of a class including offsets of the base classes within the derived class, virtual function table (vftable), virtual base class table (vbtables, *which is further described below*), and member variables, etc.

To make RTTI possible, the compiler stores several data structures in the compiled code, these data structures contains information about classes (specifically, polymorphic classes) in the code. These data structures are as follows:

RTTICompleteObjectLocator

This structure contains pointers to two structures that identify (1) the actual class information and (2) the class hierarchy:

Offset	**Type**	**Name**	**Description**
0x00	DW	signature	Always 0?
0x04	DW	offset	Offset of vftable within the class
0x08	DW	cdOffset	
0x0C	DW	pTypeDescriptor	Class Information
0x10	DW	pClassHierarchyDescriptor	Class Hierarchy Information

Below is an example how the `RTTICompleteObjectLocator` pointer is laid out. The pointer to this data structure is just below the vftable of the class:

```
.rdata:00404128                dd offset ClassA_RTTICompleteObjectLocator
.rdata:0040412C ClassA_vftable dd offset sub_401000 ; DATA XREF:...
.rdata:00404130                dd offset sub_401050
.rdata:00404134                dd offset sub_4010C0

.rdata:00404138                dd offset ClassB_RTTICompleteObjectLocator
.rdata:0040413C ClassB_vftable dd offset sub_4012B0 ; DATA XREF:...
.rdata:00404140                dd offset sub_401300
.rdata:00404144                dd offset sub_4010C0
```

And this is an example of the actual `RTTICompleteObjectLocator` structure:

```
.rdata:004045A4 ClassB_RTTICompleteObjectLocator
                 dd 0   ; signature
.rdata:004045A8  dd 0   ; offset
.rdata:004045AC  dd 0   ; cdOffset
.rdata:004045B0  dd offset ClassB_TypeDescriptor
.rdata:004045B4  dd offset ClassB_RTTIClassHierarchyDescriptor ;
```

TypeDescriptor

This structure is pointed to by the 4th DWORD field in `RTTICompleteObjectLocator`, it contains the class name, which if obtained will give the reverser a general idea what this class is supposed to do.

Offset	Type	Name	Description
0x00	DW	pVFTable	Always point to type_info's vftable
0x04	DW	spare	?
0x08	SZ	name	Class Name

This is an example of an actual TypeDescriptor:

```
.data:0041A098 ClassA_TypeDescriptor ; DATA XREF: ....
                 dd offset type_info_vftable   ; TypeDescriptor.pVFTable
.data:0041A09C   dd 0                          ; TypeDescriptor.spare
.data:0041A0A0   db '.?AVClassA@@',0           ; TypeDescriptor.name
```

RTTIClassHierarchyDescriptor

This structure contains information about the hierarchy of the class including the number of base classes and an array of `RTTIBaseClassDescriptor` (discussed later) which will eventually point to the `TypeDescriptor` of the base classes.

Offset	Type	Name	Description
0x00	DW	signature	Always 0?
0x04	DW	attributes	Bit 0 – multiple inheritance Bit 1 – virtual inheritance
0x08	DW	numBaseClasses	Number of base classes. Count includes the class itself
0x0C	DW	pBaseClassArray	Array of RTTIBaseClassDescriptor

As an example, below is a class declaration of `ClassG` virtually inheriting from `ClassA` and `ClassE`.

```
class ClassA {…}
class ClassE {…}
class ClassG: public virtual ClassA, public virtual ClassE {…}
```

And below is the actual `RTTIClassHierarchyDescriptor` for ClassG:

```
.rdata:004178C8 ClassG_RTTIClassHierarchyDescriptor ; DATA XREF: ...
.rdata:004178C8  dd 0                    ; signature
.rdata:004178CC  dd 3                    ; attributes
.rdata:004178D0  dd 3                    ; numBaseClasses
.rdata:004178D4  dd offset ClassG_pBaseClassArray ; pBaseClassArray

.rdata:004178D8 ClassG_pBaseClassArray
                 dd offset RTTIBaseClassDescriptor@4178e8
.rdata:004178DC  dd offset RTTIBaseClassDescriptor@417904
.rdata:004178E0  dd offset RTTIBaseClassDescriptor@417920
```

There are 3 base classes (including the count for `ClassG` itself), the `attribute` is 3 (multiple, virtual inheritance), and finally, `pBaseClassArray` points to an array of pointers to `RTTIBaseClassDescriptors.`

RTTIBaseClassDescriptor

This structure contains information about the base class, which includes a pointer to the base class's `TypeDescriptor` and `RTTIClassHierarchyDescriptor` and additionally contains the `PDM` structure contains information on how the base class is laid out inside in the class.

Offset	Type	Name	Description
0x00	DW	pTypeDescriptor	TypeDescriptor of this base class
0x04	DW	numContainedBases	Number of direct bases of this base class
0x08	DW	PMD.mdisp	vftable offset (if PMD.pdisp is -1)
0x0C	DW	PMD.pdisp	vbtable offset (-1: vftable is at displacement PMD.mdisp inside the class)
0x10	DW	PMD.vdisp	Displacement of the base class vftable pointer inside the vbtable
0x14	DW	attributes	?
0x18	DW	pClassDescriptor	RTTIClassHierarchyDescriptor of this base class

A `vbtable (virtual base class table)` is generated for multiple virtual inheritance. Because it is sometimes necessary to *upclass (casting to base classes),* the exact location of the base class needs to be determined. A `vbtable` contains a displacement of each base class' vftable which is effectively the beginning of the base class within the derived class.

Consider the `ClassG` class declaration previously shown; the compiler will generate the following class structure:

```
class ClassG     size(28):
         +---
 0       | {vfptr}
 4       | {vbptr}
         +---
         +--- (virtual base ClassA)
 8       | {vfptr}
12       | class_a_var01
16       | class_a_var02
         | <alignment member> (size=3)
         +---
         +--- (virtual base ClassE)
20       | {vfptr}
24       | class_e_var01
         +---
```

In this case, the `vbtable` is at offset `4` of the class. The `vbtable,` on the other hand contains the displacement of the each base class inside the derived class:

```
ClassG::$vbtable@:
 0       | -4
 1       | 4 (ClassGd(ClassG+4)ClassA)
 2       | 16 (ClassGd(ClassG+4)ClassE)
```

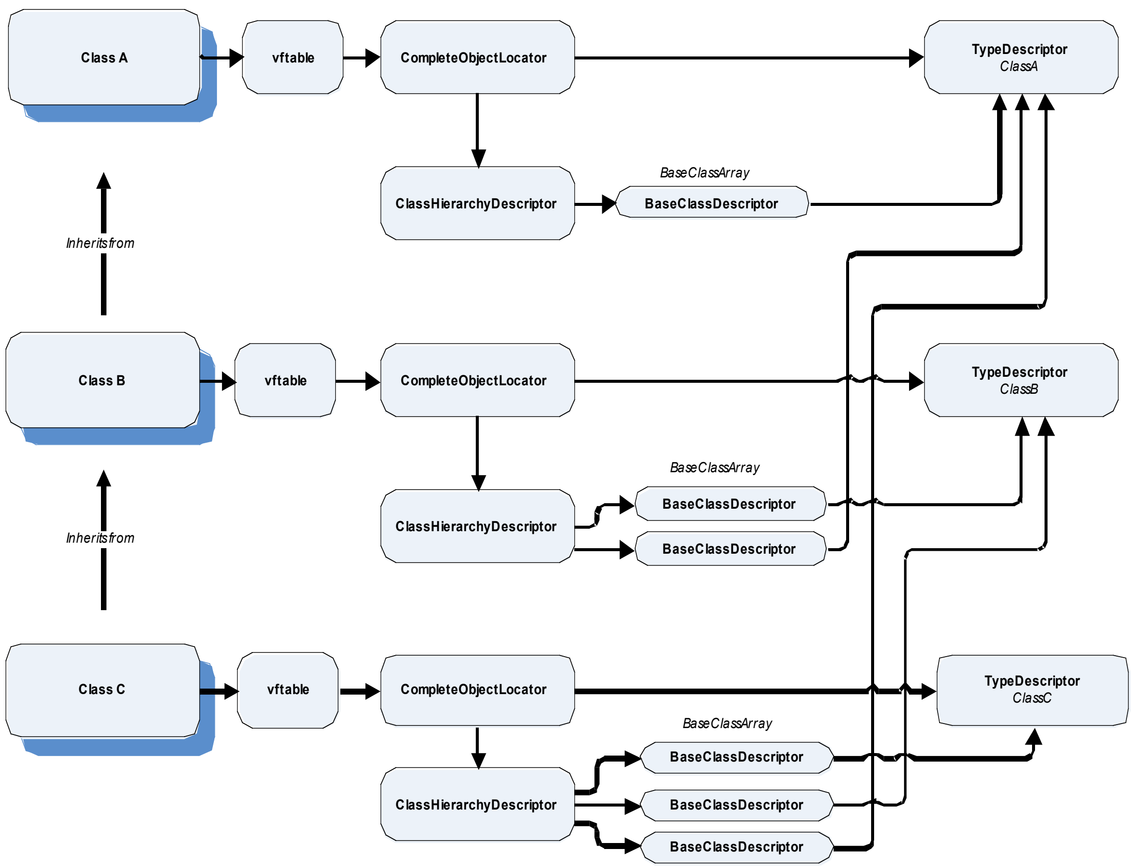
To determine the exact offset of `ClassE` within `ClassG`, the offset of the `vbtable` needs to fetched (4), then the displacement of `ClassE` from the vbtable (16) which if added equals to `20` *`(4 + 16)`*.

The actual `BaseClassDescriptor` of `ClassE` within `ClassG` is as follows:

```
.rdata:00418AFC RTTIBaseClassDescriptor@418afc        ; DATA XREF: ...
                 dd offset oop_re$ClassE$TypeDescriptor
.rdata:00418B00  dd 0                    ; numContainedBases
.rdata:00418B04  dd 0                    ; PMD.mdisp
.rdata:00418B08  dd 4                    ; PMD.pdisp
.rdata:00418B0C  dd 8                    ; PMD.vdisp
.rdata:00418B10  dd 50h                  ; attributes
.rdata:00418B14  dd offset oop_re$ClassE$RTTIClassHierarchyDescriptor ;
pClassDescriptor
```

`PMD.pdisp` is `4` which is the offset of the `vbtable` within `ClassG`, and `PMD.vdisp` is `8` which means that 3rd DWORD within the `vbtable`.

The diagram below shows the how the overall RTTI data structures are connected and laid out.


Class A
vftable
CompleteObjectLocator
TypeDescriptor
ClassA
ClassHierarchyDescriptor
BaseClassArray
BaseClassDescriptor
Inherits from
Class B
vftable
CompleteObjectLocator
TypeDescriptor
ClassB
ClassHierarchyDescriptor
BaseClassArray
BaseClassDescriptor
BaseClassDescriptor
Inherits from
Class C
vftable
CompleteObjectLocator
TypeDescriptor
ClassC
ClassHierarchyDescriptor
BaseClassArray
BaseClassDescriptor
BaseClassDescriptor
BaseClassDescriptor

D. Identifying Class Relationship

1. Class Relationship via Constructor Analysis

Constructors contain code that initializes the object, such as calling up constructors for base classes and setting up vftables. As such, analyzing constructors can give us a pretty good idea about this class's relationship with other classes.

Single Inheritance

```
.text:00401010 sub_401010      proc near
.text:00401010
.text:00401010 var_4           = dword ptr -4
.text:00401010
.text:00401010         push    ebp
.text:00401011         mov     ebp, esp
.text:00401013        push    ecx
.text:00401014         mov     [ebp+var_4], ecx ; get this ptr to current object
.text:00401017         mov     ecx, [ebp+var_4] ;
.text:0040101A         call    sub_401000 ; call class A constructor
.text:0040101F         mov     eax, [ebp+var_4]
.text:00401022         mov     esp, ebp
.text:00401024         pop     ebp
.text:00401025         retn
.text:00401025 sub_401010      endp
```

Let's assume that we have determined that this is function is a constructor using methods mentioned in section II-B. Now, we see that a function is being called using the *this* pointer of the current object. This can be a member function of the current class, or a constructor for the base class.

How do we know which one is it? Actually, there's no way to perfectly distinguish between the two just by looking at the code generated. However, in real world applications, there is a high possibility that constructors will be identified as such prior to this step (see section II-B), so all we have to do is correlate this info to come up with a more accurate identification. In other words, if a function that was pre-determined to be a constructor is called inside another constructor using the current object's *this* pointer, it is probably a constructor for a base class.

Manually identifying this would entail checking other cross-references to this function and see if this function is a constructor called somewhere else in the binary. We will discuss automatic identification methods later in this document.

Multiple Inheritance

```
.text:00401020 sub_401020      proc near
.text:00401020
.text:00401020 var_4           = dword ptr -4
.text:00401020
.text:00401020                 push    ebp
.text:00401021                 mov     ebp, esp
.text:00401023                 push    ecx
.text:00401024                 mov     [ebp+var_4], ecx
.text:00401027                 mov     ecx, [ebp+var_4] ; ptr to base class A
.text:0040102A                 call    sub_401000 ; call class A constructor
.text:0040102A
.text:0040102F                 mov     ecx, [ebp+var_4]
.text:00401032                 add     ecx, 4 ; ptr to base class C
.text:00401035                 call    sub_401010 ; call class C constructor
.text:00401035
.text:0040103A                 mov     eax, [ebp+var_4]
.text:0040103D                 mov     esp, ebp
.text:0040103F                 pop     ebp
.text:00401040                 retn
.text:00401040
.text:00401040 sub_401020      endp
```

Multiple inheritance is actually much easier to spot than single inheritance. As with the single inheritance example, the first function called could be a member function, or a base class constructor. Notice that in the disassembly, 4 bytes is added to the *this* pointer prior to calling the second function. This indicates that a different base class is being initialized.

Here's the layout for this class to help you visualize. The disassembly above belongs to the constructor of class D. Class D is derived from two other classes, A and C:

```
class A  size(4):
  +---
 0       | a1
  +---

class C  size(4):
  +---
 0       | c1
  +---
```

```
class D   size(12):
  +---
  | +--- (base class A)
 0      | | a1
  | +---
  | +--- (base class C)
 4      | | c1
  | +---
 8      | d1
  +---
```

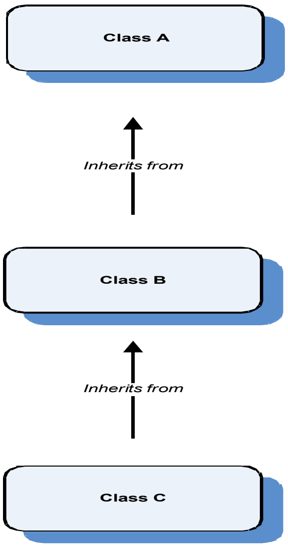
2. Polymorphic Class Relationship via RTTI

As what had been discussed in section II-B, Run-time Type Information (RTTI) can be used to identify class relationship of polymorphic classes, the related data structure used to determine this is `RTTIClassHierarchyDescriptor`. Once again, below are the fields of `RTTIClassHierarchyDescriptor` for the purpose of illustration:

Offset	Type	Name	Description
0x00	DW	signature	Always 0?
0x04	DW	attributes	Bit 0 – multiple inheritance Bit 1 – virtual inheritance
0x08	DW	numBaseClasses	Number of base classes. Count includes the class itself
0x0C	DW	pBaseClassArray	Array of RTTIBaseClassDescriptor

`RTTIClassHierarchyDescriptor` contains a field named `pBaseClassArray` which is an array of `RTTIBaseClassDescriptor (BCD)`. These `BCDs` will then eventually point to the `TypeDescriptor` of the actual base class.

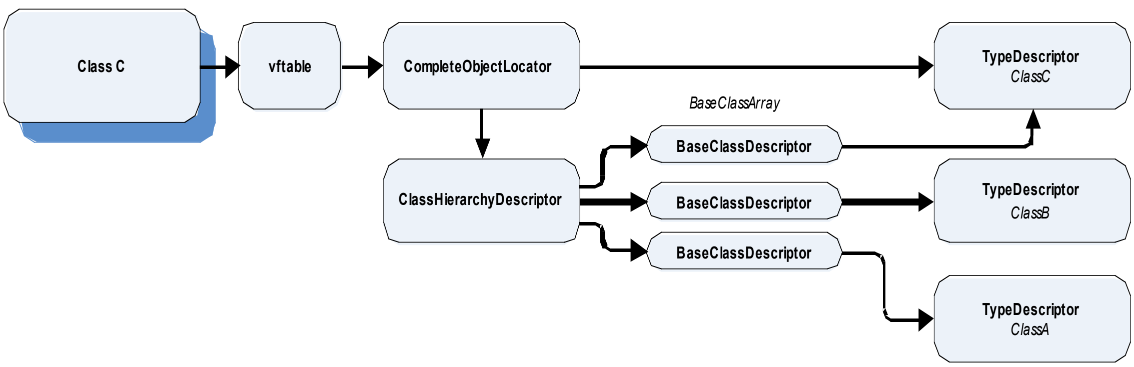
As an example, consider the following class layout:



And here is the actual class declaration pertaining to the said class layout.

```
class ClassA {…}
class ClassB : public ClassA {…}
class ClassC : public ClassB {…}
```

To illustrate, below is a layout between the relationships of `RTTIClassHierarchyDescriptor`, `RTTIBaseClassDescriptor` and `TypeDescriptor` representing `ClassC`.



As you would have noticed, one caveat is that `pBaseClassArray` also points to the `BCD` of non-direct base classes. In this case, `ClassA`'s `BaseClassDescriptor`. One solution to this is to also parse the `ClassHierarchyDescriptor` of `ClassB` and determine if `ClassA` is a base class of `ClassB`, if it is, then `ClassA` is not a direct base of `ClassC` and the appropriate inheritance can be deduced.

E. Identifying Class Members

Identifying class members is a straight-forward, albeit slow and tedious, process. We can identify class member variables by looking for accesses to offsets relative to the *this* pointer:

```
.text:00401003       push    ecx
.text:00401004       mov     [ebp+var_4], ecx ; ecx = this pointer
.text:00401007       mov     eax, [ebp+var_4]
.text:0040100A       mov     dword ptr [eax + 8], 12345h   ; write to 3rd member
                                                           ; variable
```

We can also identify virtual function members by looking for indirect calls to pointers located at offsets relative to this objects virtual function table:

```
.text:00401C21       mov     ecx, [ebp+var_1C] ; ecx = this pointer
.text:00401C24       mov     edx, [ecx] ; edx = ptr to vftable
.text:00401C26       mov     ecx, [ebp+var_1C]
.text:00401C29       mov     eax, [edx+4]   ; eax = address of 2nd virtual
                                            ; function in vftable
.text:00401C2C       call    eax ; call virtual function
```

Non-virtual member functions can be identified by checking if the *this* pointer is passed as a hidden parameter to the function call.

```
.text:00401AFC       push    0CCh
.text:00401B01       lea     ecx, [ebp+var_C] ; ecx = this pointer
.text:00401B04       call    sub_401110
```

To make sure that this is indeed a member function, we can check if the called function uses *ecx* without first initializing it. Let’s look at sub_401110’s code

```
.text:00401110                 push    ebp
.text:00401111                 mov     ebp, esp
.text:00401113                 push    ecx
.text:00401114                 mov     [ebp+var_4], ecx ; ecx used
```

III. Automation

This section discusses that approaches we had used to automate extraction of class information. For this purpose, we will discuss a tool we had created to perform this task and provide information on how we implemented this tool.

A. OOP_RE

`OOP_RE` is the name of the tool we had created in-house to automate class information extraction. The information extracted includes identified classes (including class name if RTTI is available), class relationships and class members. It also enhances disassemblies by commenting identified C++-related structures. OOP_RE is developed using python and runs in the IDAPython platform. IDAPython allows us to quickly and efficiently write and debug OOP_RE.

B. Why a Static Approach?

One of the first decisions we had to make is if we would develop a tool to perform static or dynamic analysis. We chose the static approach because it is difficult to do runtime analysis on some platforms that heavily use C++ such as Symbian - if the tool will be updated to handle compiled Symbian applications. However, a hybrid approach (static plus dynamic analysis) is also preferable since it may produce more accurate results.

C. Automated Analysis Strategies

1. Polymorphic Class Identification via RTTI

The first step the tool does is to collect RTTI information if it is available. Leveraging RTTI data allows the tool to quickly and accurately extract the following:

1) Polymorphic Classes
2) Polymorphic class Name
3) Polymorphic class Hierarchy
4) Polymorphic class virtual table and virtual functions
5) Polymorphic class Constructors/Destructors

To search for RTTI-related structures, this tool first attempts to identify virtual function tables since the structure `RTTICompleteObjectLocator` is just below these virtual function tables. In order to identify virtual function tables, the tool perform the following checks:

1) If the Item is a `DWORD`
2) If the Item is a pointer to a Code
3) If the Item is being referenced by a Code and the instruction in this referencing code is a `MOV` instruction (suggesting a vftable assignment)

Once the vftables are identified, the tool will verify if the `DWORD` below the vftable is an actual `RTTICompleteObjectLocator`. This is verified by parsing `RTTICompleteObject Locator` and verifying if `RTTICompleteObjectLocator.pTypeDescriptor is a` valid `TypeDescriptor`. One method to verify a `TypeDescriptor` is by checking if `TypeDescriptor.name` starts with a string “`.?AV`” which is used as a prefix for class names.

In the example below, the identified vftable is at `004165B4`:

```
.rdata:004165B0         dd offset ClassB_RTTICompleteObjectLocator@00
.rdata:004165B4 ClassB_vftable
.rdata:004165B4         dd offset sub_401410 ; DATA XREF: sub_401280+38□o
.rdata:004165B4                                      ; sub_401320+29□o
.rdata:004165B8         dd offset sub_401460
.rdata:004165BC         dd offset sub_401230
```

The tool will then identify if `004165B0` is a valid `RTTICompleteObjectLocator`, by checking the `TypeDescriptor` pointed to by the `RTTICompleteObjectLocator`.

```
.rdata:00418A28 ClassB_RTTICompleteObjectLocator@00
.rdata:00418A28         dd 0          ; signature
.rdata:00418A2C         dd 0          ; offset
.rdata:00418A30         dd 0          ; cdOffset
.rdata:00418A34         dd offset ClassB_TypeDescriptor
.rdata:00418A38         dd offset ClassB_RTTIClassHierarchyDescriptor
```

A `TypeDescriptor` is then validated by checking `TypeDescriptor.name for ".?AV"`

```
.data:0041B01C ClassB_TypeDescriptor
                            dd offset type_info_vftable
.data:0041B020              dd 0                  ;spare
.data:0041B024 a_?avclassb@@    db '.?AVClassB@@',0   ; name
```

Once the all the `RTTICompleteObjectLocator` is verified, the tool will parse all RTTI-related data structures to and create classes from the identified `TypeDescriptors`. Below is a list class information that is extracted using RTTI data:

```
new_class
  - Identified from TypeDescriptors
new_class.class_name
  - Identified from TypeDescriptor.name
new_class.vftable/vfuncs
  - Identified from vftable-RTTICompleteObjectLocator relationship
new_class.ctors_dtors
  - Identified from functions referencing the vftable
new_class.base_classes
  - Identified from RTTICompleteObjectLocator.pClassHierarchyDescriptor
```

2. Polymorphic Class Identification via vftables (w/o RTTI)

If RTTI data is not available, the tool will try to identify polymorphic classes by searching for vftables (the method is described section C.1). Once a vftable is identified, the following class information is extracted / generated:

```
new_class
  - Identified from vftable
new_class.class_name
  - Auto-generated (based from vftable address, etc.)
new_class.vftable/vfuncs
  - Identified from vftable
new_class.ctors_dtors
  - Identified from functions referencing the vftable
```

Notice that the base classes is not yet identified, the base classes of the identified class will be identified by constructor analysis which is described later.

3. Class Identification via Constructor / Destructor Search

Automation techniques to be discussed from this point on require us to be able to track values in registers and variables. To do this, we need to have a decent data flow analyzer. As most researchers who have tackled this problem before will attest, data flow analysis is a hard problem. Fortunately, we don't have to cover general cases, and we can get by with a simple data flow analyzer that will work in our specific case. At the very least, our data flow analyzer should be able to do decent register and pointer tracking.

Out tool will track a register or variable from a specific starting point. Subsequent instructions will be tracked and split into blocks. Each block will have a tracked variable assigned, which indicates which register/pointer is being tracked in that particular block. During tracking, one of the following things could occur:

1) If the variable/register is overwritten, stop tracking
2) If EAX is being tracked and a call is encountered, stop tracking. (We assume that all calls return values in EAX).
3) If a call is encountered, treat the next instruction as a new block
4) If a conditional jump is encountered, follow the register/variable in both branches, starting a new block on each branch.
5) If the register/variable was copied into another variable, start a new block and track both the old variable and the new one starting on this block.
6) Otherwise, track next instruction.

To identify constructors for objects that are dynamically allocated, the following algorithm can be applied:

1) Look for calls to new() .
2) Track the value returned in EAX
3) When tracking is done, look for the earliest call where the tracked register/variable is ECX. Mark this function as constructor.

For local objects, we do the same thing. Instead of initially tracking returned values of new(), we first locate instructions where an address of a stack variable is written to ECX, then start tracking ECX

There is a possibility that some of the constructors identified are overloaded and actually belong to one class. We can filter out non-overloaded constructors by checking the value passed to new(). If the object size is unique, then the corresponding constructor is not overloaded. We can then identify if the remaining constructors are overloaded by checking if their characteristics are identical with other classes e.g. has the same vftable, has the same member functions, etc.

4. Class Relationship Inferencing

As discussed in section II-D, relationships between classes can be determined by analyzing constructors. We can automate constructor analysis by tracking the current object's *this* pointer (ECX) within the constructor. When tracking is done, check blocks with ECX as the tracked variable, and see if there is a call to a function that has been identified as a constructor. If there is, this constructor is possibly a constructor for a base class. To handle multiple inheritance, our tool should also be able to track pointers to offsets relative to the class's address. We will then track these pointers using the aforementioned procedure to identify other base classes.

5. Class Member Identification

Member Variable Identification

To identify member variables, we have to track the this pointer from the point the object is initialized. We then note accesses to offsets relative to the *this* pointer. These offsets will then be recorded as possible member variables.

Non-virtual Function Identification

The tool will track an initial register or pointer, which in our case should point to a *this* pointer for the current class.

Once tracking is done, note all blocks where ECX is the tracked variable, then mark the call in that block, if there is any, as a member of the current class.

Virtual Function Identification

To identify virtual functions, we simply have to locate vftables first through constructor analysis.

After all of this is done, we then reconstruct the class using the results of these analysis.

D. Enhancing the Disassembly

1. Reconstructing and Commenting Structures

Once class information is extracted, `OOP_RE` will reconstruct, name and comment C++-related data structures using `doDwrd()`, `make_ascii_string()` and `set_name()`.

For RTTI data, `OOP_RE` properly changes the data types of data structure members and add comments to clarify the disassembly.

Here is an example for a vftable and `RTTICompleteObjectLocator` pointers:

```
Original
.rdata:004165A0         dd offset unk_4189E0
.rdata:004165A4 off_4165A4
                        dd offset sub_401170    ; DATA XREF:...
.rdata:004165A8         dd offset sub_4011C0
.rdata:004165AC         dd offset sub_401230
.rdata:004165B0         dd offset unk_418A28

Processed
.rdata:004165A0  dd offset oop_re$ClassA$RTTICompleteObjectLocator@00
.rdata:004165A4 oop_re$ClassA$vftable@00
                 dd offset sub_401170 ; DATA XREF: ...
.rdata:004165A8  dd offset sub_4011C0
.rdata:004165AC  dd offset sub_401230
.rdata:004165B0  dd offset oop_re$ClassB$RTTICompleteObjectLocator@00
```

And another example for the actual `RTTICompleteObjectLocator` structure:

```
Original
.rdata:004189E0 dword_4189E0    dd 0     ; DATA XREF:...
.rdata:004189E4                 dd 0
.rdata:004189E8                 dd 0
.rdata:004189EC                 dd offset off_41B004
.rdata:004189F0                 dd offset unk_4189F4

Processed
.rdata:004189E0 oop_re$ClassA$RTTICompleteObjectLocator@00
                 dd 0    ; RTTICompleteObjectLocator.signature
.rdata:004189E4  dd 0   ; RTTICompleteObjectLocator.offset
.rdata:004189E8  dd 0    ; RTTICompleteObjectLocator.cdOffset
.rdata:004189EC  dd offset oop_re$ClassA$TypeDescriptor
.rdata:004189F0  dd offset oop_re$ClassA$RTTIClassHierarchyDescriptor
```

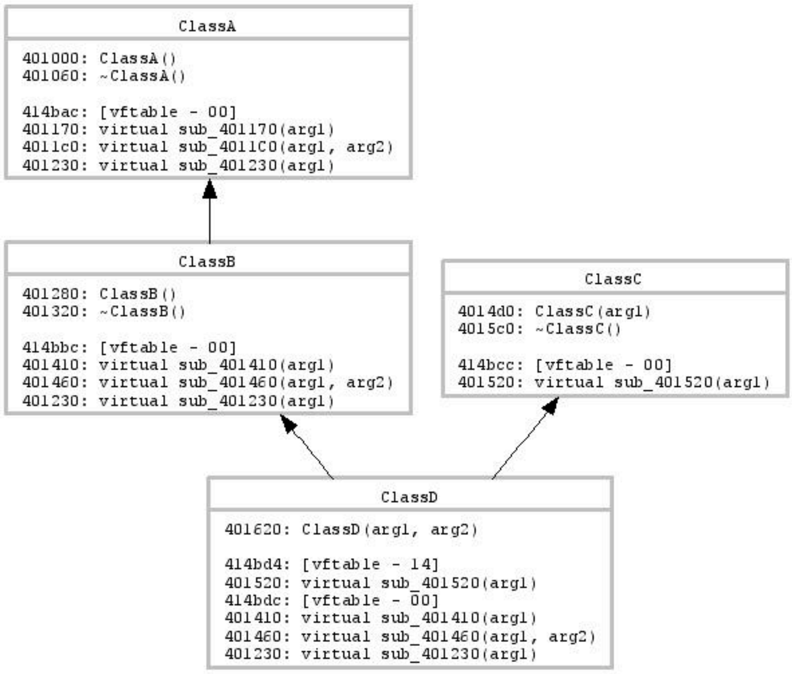
2. Improving the Call Graph

The results of the analysis done can be applied back to the IDA disassembly, for example, by adding cross-references on virtual function calls. This will yield a more accurate call graph, which in turn would result in improvements in the outcome of binary comparison tools such as BinDiff and DarunGrim. Locating vtables can also be used in a binary diffing technique, as described in Rafal Wojtczuk's blog entry (see References).

E. Visualization: UML Diagrams

Finally, the coolest part – generating a UML class diagram for class members and class hierarchy. For this purpose, we had used `pydot`. `OOP_RE` basically creates a node for each class and then create edges from each of the base classes.

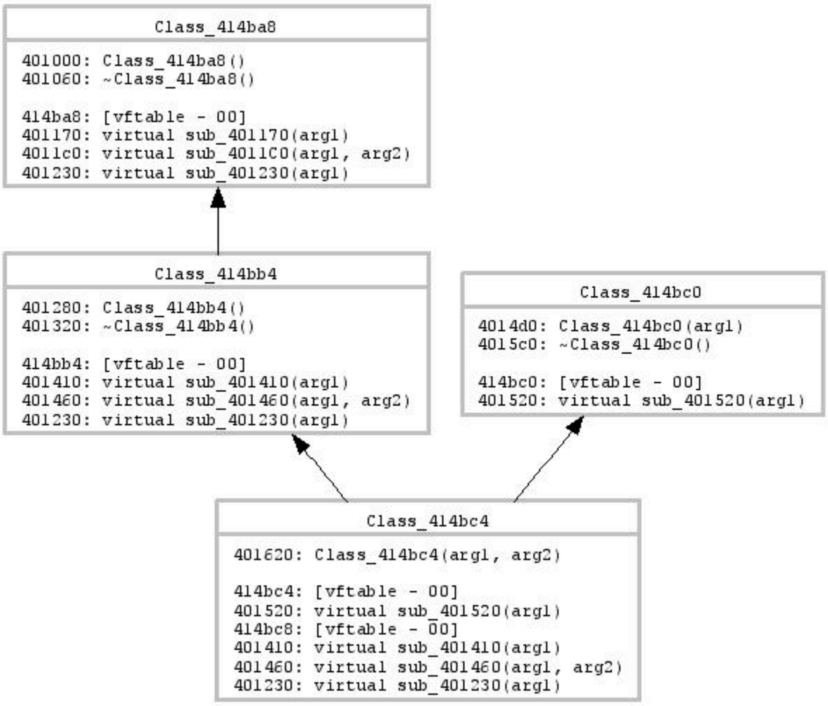
Below is an example of a generated `OOP_RE`-generated UML diagram:



This UML diagram represents the following class declaration:

```
class ClassA {...}
class ClassB : public ClassA {...}
class ClassC {...}
class ClassD : public ClassB, public ClassC {...}
```

Of course, there will be instances in which RTTI is not available; in this case, the class names are auto-generated:



These UML diagrams provide a high-level overview of the classes and how they relate to each other. This provides the reverser important information on how the application is structured in terms of classes, the reverser can then have this structure in mind while further refining the disassembly.

IV. Summary

In this paper, we had discussed ways on how to analyze and understand C++ compiled binaries. Specifically, it discusses methods on how to extract class information and class relationships. We hope that this paper will serve as a useful reference and encourage researchers to further explore the subject of C++ reversing.

V. References

Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI

https://www.openrce.org/articles/full_view/23

Igor Skochinsky

RE 2006: New Challenges Need Changing Tools

Defcon 14 talk

Halvar Flake

Inside the C++ Object Model

Stanley B. Lippman

C++: Under the Hood – Jan Gray

http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarvc/html/jangrayhood.asp

Microsoft C++ Name Mangling Scheme

http://sparcs.kaist.ac.kr/~tokigun/article/vcmangle.html#Microsoft

Binary code analysis: benefits of C++ virtual function tables detection

Rafal Wojtczuk

http://www.avertlabs.com/research/blog/?p=17

X86_RE_lib

www.sabre-security.com/x86_RE_lib.zip

Halvar Flake

IDAPython

http://d-dome.net/idapython

pydot

http://dkbza.org/pydot.html

Graphviz – Graph Visualization Software

http://www.graphviz.org