Recovery of Object Oriented Features from C++ Binaries

Kyungjin Yoo
School of Electrical and Computer Engineering
University Of Maryland
College Park, Maryland 20742, USA
Email: athleta@umd.edu

Rajeev Barua
School of Electrical and Computer Engineering
University Of Maryland
College Park, Maryland 20742, USA
Email: barua@umd.edu

***Abstract*—Reverse engineering is the process of examining and probing a program to determine the original design. Over the past ten years researchers have produced a number of capabilities to explore, manipulate, analyze, summarize, hyperlink, synthesize, componentize, and visualize software artifacts. Many reverse engineering tools focus on non-object-oriented software binaries with the goal of transferring discovered information into the software engineers trying to reengineer or reuse it.**

In this paper, we present a method that recovers object-oriented features from stripped C++ binaries. We discover RTTI information, class hierarchies, member functions of classes, and member variables of classes. The information obtained can be used for reengineering legacy software, and for understanding the architecture of software systems.

Our method works for stripped binaries, *i.e.*, without symbolic or relocation information. Most deployed binaries are stripped. We compare our method with the same binaries with symbolic information to measure the accuracy of our techniques. In this manner we find our methods are able to identify 80% of virtual functions, 100% of the classes, 78% of member functions, and 55% of member variables from stripped binaries, compared to the total number of those artifacts in symbolic information in equivalent non-stripped binaries.

I. Introduction

Reverse engineering is the process of discovering the technological principles of a device, object, or system through analysis of its structure, function, and operation [1]. Reverse engineering of binary executable code has been proved useful in many ways. It is performed to port a system to a newer platform, to unbundle monolithic systems into components and reuse the components individually, and to understand binary code for untrusted code and for malware. Such cyber-security applications are becoming the most common use of reverse engineering, leading to a rapid rise in its use in industry and government.

Understanding the disassembly of C++ object oriented code is needed due to the widespread use of C++ in many modern applications. Reverse engineering such binary code is commonplace today, especially for untrusted code and malware. Agencies as diverse as anti-virus companies, security consultants, code forensics consultants, law-enforcement agencies and national security agencies routinely try to understand binary code. Specific use cases of reverse engineering of binary code include (i) understanding vulnerabilities in third-party binary code; (ii) examining suspicious code to understand what it is doing; (iii) recovering a maintainable and modifiable source-level program or intermediate representation from binary code [2]; especially legacy code whose source code has been lost; and (iv) analyzing third-party binaries and adding security checks in them to protect them against malicious attacks. All of these tasks further key goals of software engineering in producing, maintaining, and updating high-quality and secure software with the least effort. In all of these cases, understanding the structure of the code is greatly aided by knowledge of the C++-specific features of code (such as class, methods, and inheritance.) However, until recently, engineering techniques have primarily aimed to recover assembly code or low-level, non-object oriented language abstractions from binary code.

In this paper, we discuss methods that allow the partial recovery of the C++-specific language constructs from binary code. Compared to the C programming language, C++ introduces several new concepts, presenting new challenges for decompilation. These challenges include reconstruction of classes and class hierarchies, virtual and non-virtual member functions, calls to virtual functions, exception raising and handling statements. We developed a technique to discover Run-Time Type Information (RTTI), class hierarchies, virtual function tables, member functions and member variables.

It is important to note that most of the use cases for reverse engineering can still be employed with partial recovery of artifacts. Most of those use cases (such as discovering vulnerabilities in C++ code, understanding untrusted code, or recovering pseudocode from binaries) will still work in a best-effort fashion even when not all C++ artifacts are found. Even for the use case of recovering functional source code or intermediate representation from a binary, partial recovery is still useful since when C++ features are not discovered, most existing binary rewriters such as SecondWrite [2], [3] still generate low-level but correct code without C++ artifacts.

II. Background

A. Objected Oriented Features of C++

Compared to the C language, C++ introduces several new concepts, including inheritance and class hierarchies, polymorphic types, virtual functions, and exception handling.

Inheritance is a feature of an object-oriented language that allows classes or objects to be defined as extensions or specializations of other classes or objects. Polymorphism is when some code or operations or objects behave differently in different contexts, enabling an object or reference to take different forms at different instances. For example, a pointer to a derived class is type-compatible with a pointer to its base class. A function or method is called *virtual* if its behavior can be overridden within an inheriting class by a function with the same signature. Exception handling constructs to handle the occurrence of exceptions, which are special conditions that change the normal flow of program execution. C++ supports the use of language constructs specific to exception handling to separate error handling and reporting code from ordinary code.

C++ binaries, including stripped binaries, contain additional information in the form of RTTI. RTTI is a mechanism that allows the type of an object to be determined during program execution. RTTI is part of the C++ language specification and RTTI is attached to the virtual function table [4] [5]. Therefore, a lot of information can be extracted from RTTI and virtual function tables.

B. Reverse Engineering of Object Oriented Binaries

Code discovery and generation from object oriented binaries is not trivial. For example, a virtual method call is implemented as an entire sequence of machine instructions that computes the destination of the call depending on the runtime type of an object pointer. Understanding and recovery of this type of code is challenging.

For quality C++ decompilation, the following features should be recovered.

- Virtual functions
- Classes, Class hierarchies, inheritance relations between classes
- Constructors and destructors
- Types of pointers to polymorphic classes
- Virtual and non-virtual member functions
- Layout and types of class members
- Calls to virtual functions
- Exception raising and handling statements

III. Detailed Background on Class Inheritance, Virtual Functions and RTTI

To understand our method, we first review the concepts of class inheritance and virtual methods.

Inheritance is a concept of object oriented program that allows an abstract data type to be extended by another one. Two different types of inheritances are defined: single inheritance and multiple inheritance. With single inheritance, a class inherits from only one single super class, whereas multiple inheritance allows several super classes to be inherited from. Moreover, for multiple inheritance, either replicated multiple inheritance or shared multiple inheritance can be specified, as will be explained later in this section.

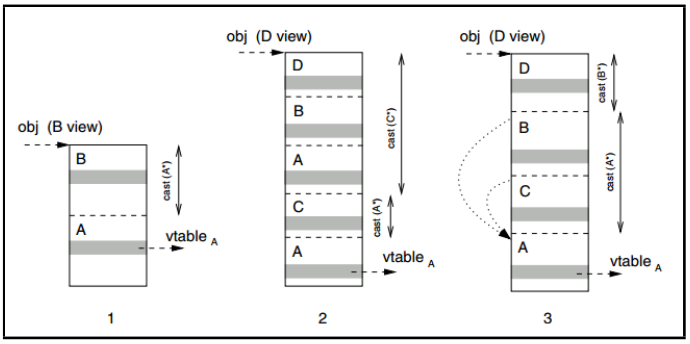

Fig. 1. Memory layout of objects of different types

Figure 1 illustrates different possibilities of arranging objects in memory. Objects with a type that is defined using single inheritance are just a sequence of the instances of all super classes. For example, instances of a class B are laid out according to Figure 1.1, if class B is defined in C++ as follows:

```
class A {...};
class B: public A {...};
```

Figure 1.2 illustrates the memory layout of an object of type D, where D is defined using replicated multiple inheritance:

```
class A {...};
class B: public A {...};
class C: public A {...};
class D: public B, public C {...};
```

For every class and super class that is used to define class D, instances are created that together compose an object of type D. For replicated multiple inheritance, the instances of B and C each contain their own instance of class A. On the other hand, shared multiple inheritance is specified using the C++ keyword *virtual*:

```
class A {...};
class B: public virtual A {...};
class C: public virtual A {...};
class D: public B, public C {...};
```

Instances of class B and class C share the same instance of class A. With an object of type D, this sharing is implemented as a pointer to the common A instance in Figure 1.3.

Method overloading is another concept in object oriented programs. For example, if a class B specifies a method with the same name as a method *foo()* in a super class A, then a call to *foo()* depends on the compile-time type of the object pointer:

```
B* b = new B();
b→foo();   /* This is a method defined
              in class B */
((A*) b)→foo(); /* This is another method
                   defined in class B */
```

Sometimes it is necessary to call the method *B::foo()*, no matter of what compile-time type the object pointer is. This is

done by specifying *foo()* as a virtual method, and the compiler generates code that looks up the correct method when the method is invoked at run-time. This special piece of lookup code is called a *method dispatcher*. If a class defines virtual methods, then instances of that class and instances of all subclasses contain a pointer to a virtual table for this particular class. This table holds the addresses of the correct virtual methods for the class and delta values that are used to correct the first parameter of the method call. This first parameter is implicitly added by the compiler and is a reference to the object that the method is invoked on. In the example above assuming that *foo()* is declared to be virtual, the object pointer b is cast to type A$*$, but the virtual method *B::foo( )* is called without a typecast and this expects a pointer to an object of type B rather than type A. Therefore, the pointer has to be corrected by adding a delta value, that corrects the object pointer from A$*$ to B$*$. This is all done by declaring *foo()* a virtual function.

Virtual method invocations on objects follow the same pattern, but they slightly differ in their implementation depending on the compile-time type of the object pointer and the object model used by the compiler. Generally speaking, a virtual method call has the following steps:

1) If the method call is performed on a type cast object pointer, then the correct sub-object (object of a sub class) is obtained first, *i.e.* the view to the object is modified,
2) given the correct view to an object, the address of the virtual table is fetched from the object, and
3) given the virtual table, the address of the virtual method is fetched from the table, as well as the delta value to correct the view to the object for the call.

RTTI contains all necessary information to recover all the above features and is placed in the binaries. RTTI in the binaries has sections for each class, and each section contains the virtual function table pointer, base class pointer, pointers to all subclasses, and the mangled name for the corresponding class. These are defined in the standard Application Binary Interface (ABI) for the platform in question. The ABI defines a binary interface between an executable program and a specific execution environment for which it is intended. The Linux Generic C++ ABI (also often called Itanium C++ ABI [4], since it was initially developed for the Itanium processor architecture), is the standard binary interface specification that was developed jointly by CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat and SGI. The following compilers comply with the Generic C++ ABI: GCC (from version 3.x upwards); Clang and llvm-gcc; Linux versions of Intel and HP compilers, and compilers from ARM. Almost all Linux-based compilers use Itanium ABI. For Windows systems, the Microsoft Visual Studio compiler (MSVC) compiler uses its own ABI which has been adopted by other Windows compilers, but these two ABIs only differ in a way that our methods can be adapted.

The Generic C++ ABI defines the following:

- Layout of both built-in and user-defined types and compiler-generated data structures in memory and peculiarities of handling them:
  - Layout of virtual function tables.
  - State of the virtual function tables during the object creation process.
  - Peculiarities of memory allocation for an array using operator *new*().
  - Initialization of guard variables, which control the initialization of function-level static variables and static class members.
  - Layout of structures used to implement RTTI.
  - Special aspects of the RTTI implementation, for example, *dynamic_cast*$\langle T \rangle(v)$ algorithm.
- Details of how virtual and non-virtual functions are called, and the behavior of constructors and destructors.
- Behavior at the linking stage:
  - Name mangling (*i.e.*, encoding) of external names (external means being visible outside the object file where they occur.)
  - Vague linkage rules. In some cases, some entities can be defined in several object files; however, in the final program, only one copy should be preserved. For example, it can happen with out-of-line functions (inline functions which the compiler has decided not to inline), virtual function tables, *typeinfo* information, and instantiated template classes.
  - Details of the unwind table layout. Unwind tables are used for unwinding the stack during the process of exception handling.

The MSVC C++ ABI is similar to the above in content.

IV. Assumptions of Our Method

Like any scheme that relies on static analysis of binary code, we have some assumptions. Two classes of assumptions are identified: those because of the underlying use of a static disassembler, and those because of our method. We discuss each in turn.

The first set of limitations are common to all static binary analysis tools. It is well known that existing static disassemblers on which they rely, such as the cutting-edge one we use [6], all have limitations. First, they do not handle self-modifying code. However, existing methods such as [7] could be integrated to statically detect the presence of run-time code generation and prevent rewriting. Second, most static disassemblers do not handle binaries containing obfuscated control flow [6].

A second set of assumptions arise because of our method. First, we assume that RTTI information is present in all the stripped input binaries that we analyze. Without RTTI in the binaries, the application is not able to use those features. Those binaries without RTTI information and without using object-oriented features need not be considered because they do not use object-oriented features and hence there are no C++ features of interest to recover. However, exception handling discovery does not rely on RTTI information; thus RTTI information is not needed to recover exception handling

features. Second, we assume that all the stripped input binaries that we analyze follow a standard C++ ABI. All Linux-based compilers use Itanium ABI [4] and MSVC uses their own ABI, but these two ABIs only differ in their layout of information in the RTTI table and exception handling table. Thus we can handle both of them by handling the table layout in a slight different way but our method overall still remains the same.

Even though our methods are compiler dependent, they are minimally so, in that they rely only on long-lived compiler standards such as C++ ABIs which must always be followed for proper execution of C++ binaries. Unlike other methods such as [8], our method does not rely on pattern matching with assembly code, which is not robust and can break even with different compiler flags used and different versions of the same compiler. Instead our method relies on finding order- and instruction-independent behavior in binary code, by analyzing fundamental compiler theoretic behavior such as dataflow relationships, to deduce the presence of C++ constructs.

V. Method

Our method searches for C++ features, and acts when it finds them. If the binary was not from C++, then none of the features will be found. We discover objected oriented features of a C++ program from its input binary in these steps:

1) We statically recover the code of virtual method calls, and recover RTTI layout by discovering the virtual function table.
2) We recover constructor dispatchers by matching compiler-independent heuristic patterns.
3) Member functions and member variables are discovered and associated with classes found earlier.
4) Exception handling is discovered separately from all others above.

Each of the above steps is discussed in more detail in the four subsections A-D below, one section for each step:

A. Virtual Function Call and RTTI Discovery

In our work, we are interested in recovering a virtual method invocation from an instruction sequence that may or may not implement a method dispatcher. We do not identify possible destination addresses with this technique, as they can only be determined at run-time when a method is invoked on an actual object. However, an indication about the layout of objects and virtual tables is obtained from our analysis. Our goal is to discover virtual function calls and RTTI information.

Consider the following example C++ source code followed by possible equivalent binary and compiler intermediate representation (IR) codes. The IR code is assumed to be automatically generated by the binary rewriter.

C++ source code:

```
obj→foo();
```

Assembly code:

```
call BB (0x488228):
00010898 r[9]:=m[r[16]+8]
0001089c r[8]:=m[r[9]+8]
000108a0 r[10]:=m[r[9]+12]
000108a4 r[8]:=r[8] + r[16]
000108a8 CALL r[10]
```

Pseudocode for assembly code:

```
1: load [object_reg + #vtableOffset],
   table_reg
2: load [table_reg + #deltaOffset],
   delta_reg
3: load [table_reg + #selectorOffset],
   method_reg
4: add object_reg, delta_reg, object_reg
5: call method_reg
```

The assembly code and its pseudocode shows the five-instruction code sequence that a C++ compiler typically generates for a virtual function call. The first instruction loads the receiver object's virtual function table pointer into a register, and the subsequent two instructions index into the virtual function table to load the target address and the receiver pointer adjustment (delta) for multiple inheritance. The fourth instruction adjusts the receiver address to allow accurate instance variable access in multiple inherited classes. Finally, the fifth instruction invokes the target function with an indirect function call.

Our technique for discovering virtual method invocations works as follows and will be illustrated as needed with the example above. Given an arbitrary basic block that ends on a computed call instruction, backward slicing [9] is applied to the basic block. Thus, those instructions that compute the target of the call instruction are extracted into a slice. Given this slice, copy propagation combined with simplification generates the call expression. Copy propagation starts with the call expression of the CALL instruction at the end of the slice (r[10] in the example above). It then replaces all occurring locations (only r[10] in the example) with those expressions that are assigned to the locations in the previous assignment instruction (here r[10] is replaced by m[r[9] + 12]). The resulting intermediate expression is then simplified, and the process is repeated until the beginning of the slice is reached. The simplification of expressions is trivial — constant folding is applied and the expressions are rearranged in such a way that constants go to the right hand side of operations, and locations to the left hand side. Furthermore, every expression is matched against a small set of common idioms. With the same technique, an expression for the first parameter is derived from the basic block.

Using slicing techniques, copy propagation and simplification, a call expression and a first-parameter expression can be derived from a basic block. These two expressions must match a particular pattern and meet certain conditions in order to be recognized as a virtual method call.

The compile-time type of an object pointer and the object model dictate the instruction sequence of the dispatcher code. Even though the output of different compilers on different platforms was analyzed, we found similarities that led to the derivation of three sets of normal forms. These normal forms

are dependent on the nature of virtual function accesses in C++, and are not compiler specific.

The first set contains three normal forms that describe the retrieval of the correct virtual table from a given object pointer:

(1) ***vtbl*←*obj + vtoffset*** The object pointer is not cast, and the address of the virtual table is fetched from a fixed offset from object. For example a simple call like obj→foo() creates this form of dispatcher code.

(2) ***vtbl*←*obj + view + vtoffset*** For a cast object pointer the *view* constant is added to the object pointer first to modify the *view* to the object according to the cast. Then the address of the virtual function table is fetched. In Figure 1 this is illustrated by casting the object pointer to A* in the first and the second picture. A method call ((A*) obj)→foo() for Figure 1.1 or ((A*)((C*) obj))→foo() for Figure 1.2 relate to this normal form. The cast to C* and then to A* are merged into one single *view* constant, since the compiler is able to determine both *view* values at compile time.

(3) ***vtbl*←*(obj + view) + vtoffset*** This normal form relates to an object whose type definition makes use of shared multiple inheritance and where the cast of the object pointer happens to be a cast to a shared instance. Again, before the address of the virtual table is fetched from the object the *view* to the object needs to be modified. The *view* to the shared instance is obtained by dereferencing the pointer. Figure 1.3 depicts the additional indirection to the shared instance and ((A*)((B*) obj))→foo() serves as an illustration.

When the virtual table is obtained from an object, the address of the virtual method is fetched from the table as well as the delta value to correct the view to the object for the call site. The second set contains two normal forms to obtain the address of the virtual method from the virtual table:

(4) ***vmth*←*vtbl + offset*** The address of the target method can be obtained from the virtual table from a fixed offset. This form is used under single inheritance.

(5) ***vmth*←*vtbl + row + offset*** In this case the correct row in the table is selected first, and then the address of the virtual method is retrieved from their assigned columns. This form is used under multiple inheritance.

To compute the value of the first parameter (the *this* pointer) that is implicitly passed to any method call, we found two possible normal forms for the third set:

(6) ***this*←*obj*** This passes the modified view to an object as a first parameter. The destination address of the call that is stored in the virtual table is not the address of the virtual method, but trampoline code, that corrects the view to the object for the virtual method and then jumps to the actual method address.

(7) ***this*←*obj + f(obj)*** Here, the first parameter is corrected by adding the appropriate delta value from the virtual table to the object pointer. The function *f* takes an object pointer and obtains the correct delta value from the virtual table using a combination of the normal forms mentioned above.

The call expression and the first-parameter expression that are derived from a basic block must meet all of the following conditions in order to be matched as a virtual method call:

- The call expression must match normal forms to retrieve the virtual table from an object, and to obtain the correct method address from the virtual table,
- the first-parameter expression must match either one of the normal forms for the first parameter, and
- for both call and the first-parameter expressions *obj* must be stored at the same location.

Thus, a computed call basic block is identified as the implementation of a virtual method call dispatcher, if there exist two expressions (obtained using slicing, copy propagation and simplification) that match a valid combination of the normal forms mentioned above, and if those two expressions use the same location for their respective object pointer.

We illustrate this technique by analysing the IR representation of the basic block shown earlier in this section. The basic block consists of several assignment instructions and the CALL instruction at the end of the basic block. The destination address of the CALL can be found at run-time in register r[10], and the only parameter location in variable r[8].

The slice that computes the target address of the call in r[10] is

```
r[9]:=m[r[16]+8]
r[10]:=m[r[9]+12]
```

and the slice for the first parameter is

```
r[9]:=m[r[16]+8]
r[8]:=m[r[9]+8]
r[8]:=r[8]+r[16]
```

Copy propagation and simplification is applied to both slices – to call slice to create the call expression, and to the first-parameter slice to create the first-parameter expression. The output of our tool that implements our technique follows:

```
call m[m[r[16]+8]+12]
m[m[r[16]+8]+8]+r[16]
```

Both expressions are now matched against the different normal forms. The call expression matches the normal form (1) to retrieve the address of the virtual table from the object (vtbl = m[r[16]+8]), and normal form (4) to obtain the address of the target method from the virtual table (m[vtbl+12]). In this example, the first-parameter expression matches normal form (7): r[16]+f(r[16]) and *f* obtains the virtual table from the object and sign-extends the delta value from 16 to 32 bits. Finally, both expressions use register r[16] that holds the object pointer at run-time.

Since all of our criteria for a virtual method dispatcher are met by the given basic block, the CALL instruction at the end of the basic block is virtual function call.

The problem of identification of virtual functions can be reduced to the problem of locating the RTTI table which

contains virtual function tables. A pointer to the RTTI table of a class always precedes its virtual function table. Therefore, finding virtual function calls helps locate the virtual function table in the RTTI table. As we described above, we are able to retrieve the virtual function table using the three normal forms (1) to (3) above that describe the retrieval of the correct virtual table from a given object pointer.

Our methods are able to identify 80% of virtual functions. It is not identified 100% because our method cannot find the case where the virtual function is only called from within the same class as the one the virtual function itself is declared in. This case is not recognized since here the call is not through the virtual function table, which we use for recognition of virtual function calls. We recognize the more common case when a virtual function is called from outside its own class at least once in the program, in which case it is called via the virtual function table, and is therefore recognized.

All (100%) of the classes can be recovered as well as their names. The class hierarchy is reconstructed by utilizing RTTI. For each polymorphic class, an RTTI structure containing information about its parents is emitted by the compiler. The complete polymorphic class hierarchy can then be reconstructed by examining all RTTI structures found earlier. The layout of RTTI structures is defined by the ABI [4] that is used by the C++ compiler. RTTI structures can be parsed, thus yielding the complete polymorphic class hierarchy exactly as it was in the source C++ program even from stripped binaries. Thus the accuracy of recovered classes is 100%. Since RTTI structures contain mangled class names, class names can also be recovered.

B. Discovery of Classes

In order to detect classes, we first need to detect their constructors and destructors. Constructors and destructors are detected by checking the operations they perform. A constructor of a class performs the following sequence of operations: it first calls constructors of direct base classes; second it calls constructors of data members; third it initializes vtable (virtual function table) pointer field(s); and fourth it performs user-specified initialization code in the body of the constructor. Conversely, a destructor deinitializes the object in the exact reverse order to how it was initialized: it first initializes virtual function table pointer and performs user-specified destruction code in the body of the destructor; second it calls destructors of data members; third it calls destructors of direct bases.

Example of Constructor and Destructor:

```
004010AD lea ecx, [ebp+var_8]
004010B0 call sub_401000
         /*constructor*/
004010B5 mov edx, [ebp+var_8]
004010B8 push edx
004010B9 call sub_4010EA
004010BE add esp, 8
004010C3 lea ecx, [ebp+var_8]
004010C6 call sub_4010200
         /*destructor*/
```

Now that we have detected constructors and destructors, we can identify their classes by examining how objects of these classes are created in the binary code. This can provide us with hints on identifying them from the disassembly. Here are three types of objects that C++ creates.

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 for global objects is implicitly called before *main()*, during C++ startup, and the destructor is called at the program exit.

To identify a possible global object, we first recognize the register containing the *this* pointer, which in the example above is *ecx* since it contains a pointer to a global variable. Then we look for a function called with the *this* pointer discovered above as an argument. To locate the constructor and destructor, we have to examine cross-references to this global variable. We look for locations where this variable is passed as the first argument to a function call, since the *this* pointer is always passed as the first argument to constructors. If this call lies between the path from program entry point and main(), it is 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. Space the size of the object is allocated in the stack. The constructor for local objects 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 as first argument that points to an uninitialized stack variable. The destructor is the last function called with this *this* pointer as first argument in the same block where the constructor was called.

(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.

To identify constructors for objects that are dynamically allocated, we look for the earliest call where the returned value in the call to *new()* is the *this* pointer.

C. Class, Member Functions and Variables Recovery

When reconstructing non-virtual functions, it is often desired to determine if a function at hand is a member function and to find the class that it belongs to. The class that the

member function belongs to is then determined by the type of the *this* parameter. In the Itanium ABI [4] that GCC and all other Linux-based compilers use, member functions are distinguishable from free-standing functions with the *this* pointer passed as the first parameter. MSVC as well by default uses the *this* call calling convention for member functions, which passes *this* pointer in the *ECX* register. In this case member functions can also be reliably distinguished.

Identifying class members is straight-forward. We can identify class member variables by looking for accesses to offsets relative to the *this* parameter in the method in question. We can also identify virtual function members by looking for indirect calls to pointers located at offsets relative to this object's virtual function table. Non-virtual member functions can be identified by checking if the *this* pointer is passed as a hidden parameter to the function call. To make sure that this is indeed a member function, we can check if the called function uses *ecx* without first initializing it.

Our methods are able to identify 100% of the classes, 78% of member functions, and 55% of member variables. There are three main reasons why less than 100% member functions and variables are detected. First, we cannot find 100% of the virtual function calls, so our method failed to locate all the virtual function tables. In such a case of failing to locate virtual function tables, it cannot retrieve the member functions and member variables belonging to the class. Second, another case of incomplete detection is that it is not yet implemented to detect the class from member functions and variables in C++ templates. Templates are a feature of the C++ programming language that allow functions and classes to operate with generic types. It is not able to associate the class with member functions and member variables where they are combined out of multiple classes as a template. Third, functions that are inlined at all their call sites are not discovered, since they effectively disappear as separate entities.

D. Exception handling discovery

Exception handling is a C++ concept designed to handle the occurrence of exceptions, which are special conditions that change the normal flow of program execution. Exception handling is normally used for reporting and handling errors that occur during program execution in a uniform way.

As with many language features, the C++ standard defines the semantics of exception raising and handling, but leaves its implementation up to compiler vendors. We have considered two implementation schemes. In the first scheme, used by MSVC, the compiler generates code that continuously updates exception handling structures to reflect the current program state. The structure that is updated is a new element that is added to the stack frame layout. The structure element contains the information on exception handlers that is available for the function associated with that frame. If an exception is thrown, this element is used by the run-time support library to locate and execute the appropriate exception handler [10].

The second scheme, used by GCC, employs a table-driven approach and introduces no run-time overhead if exceptions are not used. It involves the creation of statically allocated tables that relate ranges of the program counter to the program state. When an exception is thrown, the run-time system looks up the current value of the program counter in these tables and determines which handlers are to be checked [4].

Many algorithms for control flow analysis [11] do not take exception handling into account. As a result, *catch* blocks are isolated into separate functions. This is why proper reconstruction of exception handling requires intervention on several decompilation stages. It cannot be implemented as a post-processing step that would fix the decompiled C code. Exception handling, while a high-level concept, involves low-level manipulations that do not translate well into C. For example, non-trivial control flow of the exception handling cannot be implemented in C without assembly.

In our method, exception handling structures are located and parsed after construction of the control flow graph as will be described later in this section, and additional edges of a special kind are inserted into it. These edges connect *catch* blocks with the functions they belong to, thus preventing them from being isolated into separate functions. On the high-level program generation, the presence of edges of this kind is used to guide the reconstruction of actual *catch* blocks.

Due to the differences in exception handling implementations between compilers, there is no universal way of reconstructing *try* blocks and *throw* statements. We describe reconstruction for GCC-compiled programs.

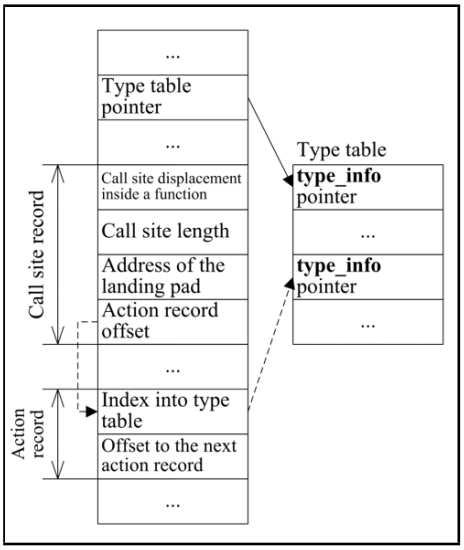

Fig. 2. Structures used by GCC for exception handling

GCC for x86 and x64 architectures by default use Dwarf2 table-based unwinding mechanism for exception handling. In Dwarf2 each function is associated with a set of call sites. Call sites are code sections that can potentially throw an exception, *e.g.* function calls or *throw* statements. Each call site is associated with a landing pad which is a code block that calls destructors and transfers execution to the corresponding *catch* block. Information about call sites and landing pads is

statically allocated and is present in the low-level code.

For each call site a call site record is emitted by the compiler (Figure 2). It contains:

- the call site displacement inside a function;
- the call site length;
- the address of the corresponding landing pad;
- the pointer to the list of action records.

Each action record contains an index of an element in the table of *type_info* pointers. The list of action records describes exception types that are handled by the landing pad. Details on the format of this information and on how exception handling is performed using it is described in Itanium ABI [4] and Linux Standard Base specification [12].

For quality reconstruction of exception handling, the following constructs must be recovered:

- *catch* blocks;
- *try* blocks;
- *throw* statements.

Catch blocks are recovered as follows. Each *catch* block is referenced from its corresponding landing pad, starts with a call to *_cxa_begin_catch* and ends with a call to *_cxa_end_catch*. Thus *catch* blocks can be reconstructed by examining the landing pad and the locations it references.

To understand how *try* blocks are discovered, consider that different destructors must be called when an exception is thrown from different call sites. That is why the compiler generates several landing pads for each *try* block. However, the part of the landing pad that performs the dispatch to the *catch* block is shared by all landing pads for all call sites of a single *try* block.

Try blocks are recovered as follows. First, call sites belonging to the same *try* block can be identified by analyzing their corresponding landing pads if two landing pads share the same dispatch block, then their corresponding call sites belong to the same *try* block. They are not dependent upon it. Extents of the *try* block are reconstructed by uniting the extents of all its corresponding call sites.

Exception raising in GCC is performed via a call to the *_cxa_throw* function. To reconstruct throw statements, it is sufficient to locate the calls to *_cxa_throw* and find the values of its parameters.

VI. Implementation

Figure 3 presents an overview of SecondWrite [6], [3]; our executable analysis and rewriting framework. The methods in this paper have been implemented in SecondWrite. SecondWrite translates the input x86 binary code to the intermediate format of the LLVM compiler [13]. The disassembler along with the binary reader translates every x86 instruction to an equivalent LLVM instruction. A key challenge in binary frameworks is discovering which portions of the code section in an input executable are definitely code. Smithson *et. al.* [6] proposed speculative disassembly, coupled with binary characterization, to efficiently address this problem. SecondWrite speculatively disassembles the unknown portions

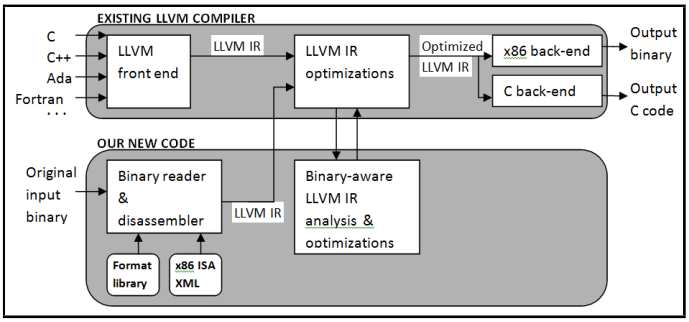

Fig. 3. SecondWrite Flow

of the code segments as if they were code. However, it also retains the unchanged code segments in the IR to guarantee the correctness of data references in case the disassembled region was actually data.

SecondWrite employs binary characterization to limit such unknown portions of code. It leverages the restriction that an indirect control transfer instruction (CTI) requires an absolute address operand, and that these address operands must appear within the code and/or data segments. The binary segments are scanned for values that lie within the range of the code segment. The resulting values are guaranteed to contain, at a minimum, all of the indirect CTI targets. Memory stack analysis is done for every procedure to detect its corresponding memory arguments as explained in [3]. The techniques presented in [3] along with [14] are used to split the physical stack into individual abstract stack frames. Global and stack regions appear as arrays of bytes in the IR.

Even though, our methods use SecondWrite framework for decompiling and rewriting the stripped input binaries, they are not dependent upon it. Our methods can be implemented in any other binary rewriter or decompiler.

VII. Results

We tested our work on stripped binaries compiled from all six C++ benchmarks presented in SPEC CPU 2006 [15] and all five benchmarks in the OOCSB suite [16]. All sixteen C++ programs from NEC lab [17] were used to test exception handling discovery since they were specifically written to heavily use and text exception handling. These are all benchmark programs that are available to compile and then provide stripped binaries from those sets. We compiled the programs to binaries using GCC which uses the Itanium ABI; but binaries compiled by MSVC are left for future work. However we know that MSVC binaries can be handled by simply re-analyzing the RTTI layout, since MSVC uses a different ABI which is also standardized just like the Itanium ABI, but otherwise the method is the same.

The characteristics of the SPEC and OOCSB benchmarks used in this study are listed in the tables I and II respectively.

We used NEC benchmarks, which are C++ programs with exceptions [17]. This benchmark set contains 16 C++ programs that cover various aspects of the exception semantics of C++. These small programs test usage of various C++

TABLE I
SUMMARY OF C++ PROGRAMS IN SPEC CPU 2006

Benchmark	Characteristics	
	Lines	*Comments*
471.omnetpp	47,910	The benchmark performs discrete event simulation of a large Ethernet network. The operation of the Ethernet MAC, traffic generator etc. are in C++.
473.astar	5,849	tt makes very little use of C++ features
444.namd	5,322	A good HPC benchmark but somewhat simple
447.dealII	198,649	Uses Boost libraries and complex template techniques. Best representative of future C++ directions.
450.soplex	41,435	Not very high on usage of C++ features.
453.povray	155,170	It is representative of C++ the way it is used currently. Has potential single hot spot in the noise function.

TABLE II
OOCSB BENCHMARK

Benchmark	Characteristics	
	Lines	*Comments*
deltablue	1,400	incremental dataflow constraint solver
idl	25,900	SueSoft's IDL compiler (version 1.3) using the demonstration back end which exercised the front end but produces no translated output
ixx	11,900	IDL parser generating C++ stubs, distributed as part of the Fresco library (which is part of X11R6). Although it performs a function similar to IDL, the programs was developed independently and is structured differently
lcom	16,200	optimizing compiler for a hardware description language developed at the University of Guelth.
richards	1,100	simple operating system simulator

exception features in realistic scenarios, some of which are close to some standard C++ collection class usage. In addition, to cover C++ exception semantic features, these benchmarks can also be used to check for certain exception-safety guarantees as defined by Stroustrup [18]. The benchmarks range in size from about 40 lines of C++ code to about 460 lines. Each benchmark set contains a number of classes, and a main function that drives the execution as a test harness, and may contain user-defined exceptions.

A. Virtual Function Call and RTTI Discovery

For testing of class hierarchy reconstruction correctness, the following automatic process is used. First, the program is compiled, and the RTTI-aware class hierarchy reconstruction algorithm is used to recover information about the polymorphic class hierarchy. This algorithm always provides correct results. Our algorithm works for stripped binaries, i.e., those without symbolic or relocation information. However, for the sake of measurements only, compiler-generated debug information is used to establish a correspondence between the class hierarchies reconstructed from the program vs. that listed in the debug information. The debug information contains information about what C++ artifacts were found in the source code after optimization. The two class hierarchies are then compared. The test results are presented in Table III.

Our methods are able to identify 80% of virtual functions. This is not 100% because of reasons in section V(A).

TABLE III
DISCOVERY OF VIRTUAL FUNCTION CALLS

Benchmark	From the source		From the stripped binary			
	Static number of virtual function call	*Dynamic number of virtual function call*	*Static number of virtual function call (percentage)*		*Dynamic number of virtual function call (percentage)*	
471.omnetpp	269	3,129	203	(75%)	2,472	(79%)
473.astar	3	24	3	(100%)	24	(100%)
444.namd	2	32	2	(100%)	32	(100%)
447.dealII	63	3,423	51	(81%)	2,721	(79%)
450.soplex	332	4,328	264	(80%)	3,543	(81%)
453.povray	65	2,314	52	(80%)	1,820	(79%)
deltablue	16	4,250	11	(69%)	3,010	(70%)
idl	516	1,756	448	(87%)	1,530	(87%)
ixx	157	102	113	(72%)	73	(72%)
lcom	321	1,103	216	(67%)	729	(66%)
richards	7	3,290	5	(71%)	2,350	(71%)
Average percentage of artifacts recovered among those in debug information			**(80%)**		**(80%)**	

B. Class, Member Functions and Member Variables

For testing the accuracy of the recovery of classes, member functions and variables, the same benchmarks are used as in part 1. Test results are presented in Table IV.

As Table IV shows, our methods are able to identify 100% of the classes, 78% of member functions, and 55% of member variables. The reasons why the detection of member functions and variables is not 100% were described in section V(C)..

TABLE IV
DISCOVERY OF CLASSES, MEMBER FUNCTIONS, AND MEMBER VARIABLES

Benchmark	From the source			From the stripped binary (and percentage)					
	Class	*Member functions*	*Member variables*	*Class*		*Member functions*		*Member variables*	
471.omnetpp	86	340	401	86	(100%)	321	(84%)	289	(65%)
473.astar	68	284	352	68	(100%)	213	(74%)	252	(66%)
444.namd	51	230	324	51	(100%)	172	(72%)	244	(68%)
447.dealII	703	1,838	2,836	703	(100%)	1,672	(79%)	1,919	(63%)
450.soplex	310	837	1,028	310	(100%)	764	(75%)	578	(52%)
453.povray	155	638	753	155	(100%)	578	(81%)	383	(44%)
deltablue	11	64	73	11	(100%)	57	(80%)	39	(47%)
idl	148	732	836	148	(100%)	671	(88%)	452	(51%)
ixx	282	1,038	1,326	282	(100%)	893	(71%)	719	(51%)
lcom	510	2,373	2,964	510	(100%)	2,291	(82%)	1,632	(52%)
richards	15	73	96	15	(100%)	63	(73%)	48	(43%)
Average percentage of artifacts recovered among those in debug information				**(100%)**		**(78%)**		**(55%)**	

C. Exception Handling Discovery

Testing of the reconstruction of exception handling constructs was performed. Description of the tests is presented

in Table V. Columns *Try*, *Catch* and *Throw* show the number of reconstructed *try* blocks, *catch* blocks and *throw* statements respectively. Our test shows that in all these tests all exception handling constructs present in the source file were reconstructed correctly. No spurious constructs were recovered.

TABLE V
NECLAB: C++ PROGRAMS WITH EXCEPTIONS

Benchmark	*Try*	*Catch*	*Throw*	Result
bintree-duplicate	**1**	**1**	**1**	**Success**
list-baditerator	**1**	**1**	**1**	**Success**
delegation-dtor-throw	**1**	**1**	**10**	**Success**
diamond-shared-inheritance	**2**	**8**	**10**	**Success**
recursive	**3**	**4**	**4**	**Success**
std-uncaught-dtor	**1**	**1**	**10**	**Success**
ctor-throw	**2**	**4**	**3**	**Success**
virtual-throw	**2**	**5**	**5**	**Success**
dynamic-cast	**2**	**2**	**0**	**Success**
io	**1**	**1**	**1**	**Success**
new-badalloc	**2**	**2**	**0**	**Success**
template	**1**	**3**	**5**	**Success**
nested-try-catch	**3**	**4**	**2**	**Success**
loop-break-continue	**3**	**4**	**2**	**Success**
nested-rethrow	**3**	**5**	**4**	**Success**
multiple-live	**2**	**2**	**2**	**Success**

VIII. Related Work

Many pieces of research have performed decompilation of C programs but they do not perform well for C++ programs. These C decompilation tools include Hex-Rays [19], Boomerang [20] and REC Studio [21] and they showed good results for C programs. However they have not attempted the recovery of C++ artifacts.

Currently there exists no decompiler that is capable of discovering all C++ artifacts. There is some support for C++ in the latest version of the Rec Studio [21], which reconstructs mangled names of functions and class inheritance hierarchy. REC Studio is an interactive decompiler not an automatic one process decompiler like ours. It reads a Windows, Linux, Mac OS X or raw executable file, and attempts to produce a C-like representation of the code and data used to build the executable file. In contrast, our method discovers features they do not discover, such as classes, virtual functions, member functions, member variables and exception handling..

Skochinsky [8] has given a detailed description of RTTI and exception handling structures used by MSVC, along with implementation details of some C++ concepts like constructors and destructors. He presents tools for reconstruction of polymorphic class hierarchies and exception handling statements from the binary and representing in the assembly code. Unlike our method, their method does not discover virtual functions, member functions and member variables from C++ binaries. Further, these tools are based on heuristic pattern matching of assembly code. They are not robust since even when using a certain compiler, different compiler flags and future versions of the compiler can change the assembly code used, thus breaking the heuristic. In contrast our methods are not compiler dependent.

Sabanal and Yason [22] have proposed a technique for class hierarchy reconstruction based on the analysis of vtables and constructors that can be applied without relying on RTTI structures in the binary. Constructors are identified by searching for operator *new()* calls followed by a function call. vtable analysis is used for polymorphic class identification. Class relationship inference is done via analysis of constructors. They present several examples of successful class hierarchy reconstruction. However, they do not detect member functions, member variables, or exception handling statements. Moreover several cases in which presented techniques may fail are not considered. These cases include operator *new()* overloading, constructor inlining and elimination of vtable references in constructors due to optimizations. Our methods will not fail under these scenarios. The presented techniques also heavily rely on the use of MSVC-specific *this* call calling convention. In contrast, our method does not rely on any compiler-specific behavior other than the use of standard C++ ABIs, for which practically speaking, there are only two, as was discussed in section V.

Fokin [23] and Srinivasan [24] have both proposed methods for automatic reconstruction of class hierarchies that do not rely on RTTI information and perform well with aggressive compiler optimizations. However, precision is lower than with our method using RTTI, and their methods do not discover member functions or member variables. They do not discover exception handling blocks either.

In conclusion, we can see that our method has four main advantages over related work: First, most related works are focused primarily on class hierarchy recovery only. Only one recovers exception handling statements, and none recovers member functions, member variables or virtual functions. In contrast, our method recovers all of the above C++ artifacts. A second advantage of our method is that some related work relies on non-robust pattern matching approaches for assembly code, which may not work when compiler flags or versions are changed. Our method relies only on well-known standard C++ ABIs which compilers must follow and thus is more robust. Third, these advantages are shown in implemented results on large C++ programs, some hundreds of thousands of lines of code, whereas most related works are evaluated on much smaller programs or are not evaluated at all.

IX. Conclusion

In this work, we show how a stripped binary program can be analyzed to recover the source-code-level class hierarchy, member functions, member variables and exception handling features. This work can be used in variety of reverse engineering applications, both for analysis and rewriting. Our method can discover 100% of classes and exception handling, but can't discover 100% some artifacts including member function and member variables. This is due to some cases that we have not implemented to identify them such as more code sequences for virtual function calls and C++ templates. Future work may increase the percentage of artifacts recovered.

REFERENCES

[1] E. Eilam and E. J. Chikofsky, *Reversing: secrets of reverse engineering*. John Wiley and Sons, 2007.

[2] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *PLDI '13 Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 51–60.

[3] K. Anand, M. Smithson, K. ElWazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *the 8th ACM European Conference on Computer Systems*, 2013, pp. 295–308.

[4] Itanium C++. [Online]. Available: http://mentorembedded.github.io/cxx-abi/

[5] L. D. Stroustrup B., "Run-time type identication for c++(revised)." in *Proc USENIX C++ Conference*, August 1992.

[6] M. Smithson, K. Elwazeer, K. Anand, A. Kotha, and R. Barua, "Static binary rewrting without supplemental information : Overcoming the tradeoff between coverage and correctness," in *the 20th Working Conference on Reverse Engineering (WCRE)*, Koblenz, Germany, 2013.

[7] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Still: Exploit code detection via static taint and initialization analyses," in *Computer Security Applications Conference (ACSAC)*, 2008, pp. 289–298.

[8] Skochinsky. (2006) Reversing microsoft visual c++ part 2: Classes, methods and RTTI. [Online]. Available: http://www.openrce.org/articles/fullview/23

[9] F.Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[10] V. Kochhar. (2002, April) How a C++ compiler implements exception handling. [Online]. Available: http://www.codeproject.com/KB/cpp/exceptionhandler.aspx

[11] S.Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[12] Linux standard base core specification chapter 8. exception frames. [Online]. Available: http://refspecs.linuxbase.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html

[13] The llvm compiler infrastructure. [Online]. Available: http://www.llvm.org

[14] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *CC*, 2004.

[15] Spec. [Online]. Available: http://www.spec.org

[16] A C++ benchmark suite. [Online]. Available: http://www.cs.ucsb.edu/~urs/oocsb/

[17] Nec laboratories america, inc. [Online]. Available: http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php

[18] S. B., *Exception safety: concepts and techniques*. Springer Berlin Heidelberg, 2001.

[19] Hexrays. Idapro. [Online]. Available: http://www.hex-rays.com/idapro/

[20] M. Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in *Working Conference on Reverse Engineering*, 2004.

[21] G. Caprino. (2003) Rec - reverse engineering compiler. binaries free for any use. [Online]. Available: http://www.backerstreet.com/rec/rec.htm

[22] P. Sabanal and M. Yason, "Reversing C++," Black Hat DC, 2007.

[23] A. Fokin, K. Troshina, and A. Chernov, "Reconstruction of class hierarchies for decompilation of C++ programs," in *14th European Conference on Software Mantenance and Reengineering*, 2010, pp. 249–252.

[24] V. Srinivasan and T. Reps, "Software-architecture recovery from machine code," Computer Sciences Department, University of Wisconsin, Madison, WI, TR 1781, 2013.