Reconstruction of Class Hierarchies for Decompilation of C++ Programs

A. Fokin*, K. Troshina† and A. Chernov‡
* Computational Math. and Cybernetics Dept., Moscow State University,
Leninskie Gory, Moscow, Russia
Email: apfokin@gmail.com
† Institute for System Programming, Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia
Email: katerina@ispras.ru
‡ Computational Math. and Cybernetics Dept., Moscow State University,
Leninskie Gory, Moscow, Russia
Email: cher@unicorn.cmc.msu.ru

***Abstract*—This paper presents a method for automatic reconstruction of polymorphic class hierarchies from the assembly code obtained by compiling a C++ program. If the program is compiled with run-time type information (RTTI), class hierarchy is reconstructed via analysis of RTTI structures.**

In case RTTI structures are missing in the assembly, a technique based on the analysis of virtual function tables, constructors and destructors is used.

A tool for automatic reconstruction of polymorphic class hierarchies that implements the described technique is presented. This tool is implemented as a plugin for IDA Pro Interactive Disassembler. Experimental study of the tool is provided.

***Keywords*-decompilation; reverse engineering; C++; class hierarchy reconstruction;**

I. Introduction

Decompilation from low-level machine languages to high-level languages (especially C) has gained a fair amount of attention recently. Decompilers have been improving in quality and range of accepted low-level programs. The C programming language was chosen as a target language for decompilers essentially because it is simple enough, yet powerful and widely used.

Typical applications of decompilation are binary auditing and malware analysis [1]. A lot of modern software is written in C++ or C/C++ mix, and the use of C++ in malware is also increasing [2]. However, decompilation of C++ programs into C results in undesirable artifacts, including non-decompiled assembly fragments instead of C++ exception handling operators, or a mess of C types instead of C++ inheritance hierarchy. Therefore recovering of C++ specific language features is important for quality decompilation. This work is a step on this way.

In this work we present a method for reconstruction of hierarchies of polymorphic classes from the assembly code obtained by compiling a C++ program. We presume that no modifications of the assembly code were performed after compilation (such as assembly-level obfuscation). The problem of obtaining the assembly code from an executable file lies outside the scope of this work.

If the program was compiled with RTTI (run-time type information) enabled, hierarchy of polymorphic classes is reconstructed exactly as it was in the source C++ program. As run-time type information structures store class names, class names are also recovered.

If the program was compiled without RTTI, an inheritance relation on a set of vtables (virtual function tables) is considered. This relation is reconstructed via analysis of vtables and vtable accesses. Multiple inheritance is handled via additional analysis of constructors and destructors. Virtual inheritance is not handled (virtual inheritance and inheritance utilizing virtual functions are two distinct C++ concepts, see [3] for details).

Presented methods were implemented as a plugin for IDA Pro interactive disassembler [4] and have been tested on a variety of open-source C++ software. For implementation we considered C++ ABI (application binary interface) of Microsoft Visual Studio compiler on Windows platform and C++ ABI of GNU C++ compiler on x86 Linux. We used MSVC 9.1 and g++ 4.3 for experimental study, but the tool also works for other versions of these compilers provided they use the same C++ ABI.

This paper is organized as follows. Section II discusses related work. Class hierarchy reconstruction via analysis of RTTI structures is presented in Section III. Section IV presents the methods for class hierarchy reconstruction without RTTI structures. The experimental results are discussed in Section V. Our conclusions and directions for future work are presented in the last section.

II. Related work

There are a lot of important works on decompilation. In our opinion, the following works are the closest to the topic of this paper.

Skochinsky [5] has given a detailed description of RTTI structures used by MSVC along with the implementation details of some of the C++ concepts, such as constructors, destructors, and exception handling. He has presented a tool

for automatic analysis of RTTI structures and has successfully used it for polymorphic class hierarchy reconstruction. The tool has been implemented as a script for IDA Pro interactive disassembler.

Van Emmerik et al. [6] have described the experience gained from applying a native executable decompiler assisted by a commercial disassembler and hand editing to a real-world Windows-based application. Authors were able to recover almost all original class names and a complete class hierarchy via analysis of RTTI structures.

However, RTTI structures may not be present in the assembly, and in such cases it is impossible to use the methods proposed in the above-mentioned works.

Sabanal and Yason [2] along with RTTI-based approach have proposed a technique based on the analysis of vtables and constructors that can be applied even when RTTI structures are not present in the assembly. Vtable analysis is used for polymorphic class identification and class relationship inference is done via constructor analysis. Authors have also presented several examples of successful class hierarchy reconstruction. However, 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. Presented techniques also heavily rely on the usage of Microsoft-specific **__thiscall** calling convention.

In the paper we present a new approach to polymorphic class hierarchy reconstruction based on the analysis of vtables, vtable accesses, constructors and destructors.

III. Analysis of RTTI structures

Run-time type information in C++ is used for implementing **dynamic_cast**<> and **typeid** operators [3]. The layout of RTTI structures is defined by the ABI that is used by the C++ compiler. Each RTTI structure contains a description of a part of program's class hierarchy [5], [7]. In this paper we presume that the format of RTTI structures used by the compiler is known and can be parsed.

Given the layout and position of RTTI structures in the assembly, parsing them imposes no difficulties. Full polymorphic class hierarchy can then be reconstructed by merging all partial class hierarchies obtained from RTTI structures.

In all ABIs known to the moment a pointer to RTTI structure of a class always precedes the corresponding vtable. Therefore, the problem of finding RTTI structures can be reduced to a problem of locating vtables. Each vtable is an array of pointers to functions, with only the first element being referenced from the program code (its address is used in constructors and in a destructor of the corresponding class). Therefore, vtables can be located by scanning the data segment and checking each location in it.

- If current location is referenced from a code segment and its value is a pointer to a function, then it is marked as a start of vtable. Otherwise, the scan is continued.
- Other elements of vtable must be unreferenced pointers to functions, so finding the size of vtable is straightforward. Vtable ends with the first location that is either referenced from the program code, or is not a pointer to a function.

IV. Absence of RTTI structures

Run-time type information in C++ programs is frequently misused [8], and some modern applications written in C++ refrain from using it. In case RTTI is not used, vtables are still generated for each polymorphic class. However, the uniqueness of vtables for each class is not guaranteed, i.e. the same vtable may be shared by several different classes.

We can presume that inheritance relation on a set of vtables is single, i.e. each vtable inherits from at most one parent vtable, as we do not handle virtual inheritance. Therefore, vtable inheritance hierarchy consists of several inheritance trees.

A set of all vtables can be built using the method described in section III. For two vtables B and D, we say that vtable B is a direct base of vtable D if one of the classes corresponding to vtable B is a direct base of one of the classes corresponding to vtable D. Simple inheritance for vtables can then be defined as a transitive closure of direct inheritance. The fact that vtable D inherits from vtable B is denoted as $D \rhd B$ and is also referred to as "B is a base of D" and "D is a child of B".

Also the following relations are used:

- vtable B does not inherit from vtable D, denoted as $B \not\rhd D$,
- vtable B inherits from vtable D, or vtable D inherits from vtable B, denoted as $B \sim D$.

Note that if both conditions $B \not\rhd D$ and $B \sim D$ are satisfied, then condition $D \lhd B$ is also satisfied, and vice-versa.

A_i denotes i-th virtual function in a vtable A.

A. Reconstructing vtable inheritance hierarchy

Several simple rules are used for the reconstruction of inheritance relation. Each rule is given in a form *if [antecendent], then [consequent]*. B and D denote two distinct vtables, and i denotes some integer.

Rule 1. *If the size of vtable B is less then the size of vtable D, then B cannot inherit from D ($B \not\rhd D$).*

In compliance with C++ inheritance rules [9], if there are more virtual functions in vtable D than in vtable B, then D cannot be a base of B.

Rule 2. *If virtual function B_i is pure and virtual function D_i is not, then vtable B cannot inherit from vtable D ($B \not\rhd D$).*

In C++ it is impossible to override a virtual function that is not pure with a pure one [9].

If it is possible to reliably determine the size of the parameters of a virtual function (for example, if a function uses a calling convention where callee clears the stack), then the following rule can be used.

Rule 3. *If sizes of the parameters of virtual functions B_i and D_i are different, then neither vtable B inherits from vtable D, nor D inherits from B ($B \not\rhd D \wedge D \not\rhd B$).*

In compliance with C++ inheritance rules [9], a virtual function cannot be overridden by a function with incompatible parameters.

Some information on inheritance relation can also be gathered through the analysis of constructors and destructors. Constructor performs the following sequence of operations [9], [10]:

1) calls constructors of direct base classes;
2) calls constructors of data members;
3) initializes vtable pointer field(s) and 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:

1) initializes vtable pointer field(s) and performs user-specified destruction code in the body of the destructor;
2) calls destructors of data members;
3) calls destructors of direct bases.

In case of a "deep" inheritance hierarchy, construction and destruction of an object may require many successive initializations of vtable pointer field(s). Where appropriate, these assignments are optimized away by the compiler. Experiments have shown that GCC and MSVC never optimize away the last assignment in a virtual destructor. That means that virtual destructor of an object always overwrites each vtable pointer field with a pointer to the corresponding "most-base" vtable.

In a call to a virtual function of some class D, vtable pointer field(s) of **this** object can be overwritten if this function is a virtual destructor, or as a result of a call to a constructor or a destructor of class D.

By using interprocedural data flow analysis as it is described in [11] for locating accesses to vtable pointer field(s) of **this** object inside a call to virtual function, it is possible to find bases of a corresponding vtable using the following rule.

Rule 4. *If in a call to virtual function D_i vtable pointer field corresponding to vtable D of **this** object is overwritten with a pointer to vtable B, then D inherits from B ($D \triangleright B$).*

As advised by almost every guide on object-oriented design, most of the polymorphic class hierarchies are designed for polymorphic deletion, and therefore use virtual destructors [12]. That means that in most cases rule 4 will produce at least one inheritance relation — between the vtable being analyzed and its corresponding "most-base" one.

Rules 1-4 are applied, and all consequents are stored as a set of restrictions on the structure of inheritance hierarchy. These restrictions are then processed to construct several inheritance trees on a set of vtables. Detailed description of the method used is provided in the documentation of the tool [13].

B. Reconstructing polymorphic class hierarchy

Correspondence between vtables and actual classes is reconstructed via constructor and destructor analysis. The information obtained is then used for multiple inheritance inference.

Interprocedural data flow analysis is used to detect code sites where one or more memory locations (each one differing from each other by a constant offset) are overwritten with pointers to vtables. These code sites are referred to as *vtable access sites*. Every such site is associated with a set of pairs (*positive offset*, *vtable sequence*). Vtables are stored in an order in which their addresses were written at the corresponding offset to the memory location that is being tracked.

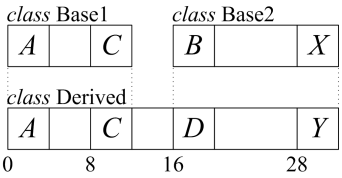

Fig. 1. Example of base class matching. Here B is a direct base of D and X is a direct base of Y.

On this step, full information on inheritance relation between vtables is available. In accordance with the actions performed in constructor and destructor, each vtable access site with one of the vtable sequences containing more than one element can be classified as either a constructor or a destructor.

Constructor and destructor calls, if not inlined, are nested: constructor calls constructors for base classes, etc. Therefore, once a vtable access site has been classified as either a constructor or a destructor, vtable access sites to the same memory location in all nested calls are classified as having the same type.

Constructors and destructor for "most-base" classes overwrite vtable pointer field only once, and therefore cannot be easily differentiated if not nested into some other constructor or destructor. That's why in this case they are associated with a corresponding set of (*offset*, *vtable sequence*) pairs and classified as constructors.

After completing vtable access site analysis, for each site classified as being a constructor the corresponding set of (*offset*, *vtable sequence*) pairs is transformed. The last vtable is picked from each sequence thus producing a set of (*offset*, *vtable*) pairs. Each such pair identifies a unique class, with vtables being the virtual tables belonging to this class and offset values being offsets to instances of the corresponding base classes.

Generally speaking, the fact that vtable belongs to a class does not necessarily mean that this class has inherited it. However, inheritance and inclusion as a field are normally indistinguishable. Besides, in most cases polymorphic classes are included by pointer. That's why by default we presume that all vtables belonging to a class have been generated as a result of inheritance.

Constructors and destructors for "most-base" abstract classes may have been optimized away by the compiler. That's why for each vtable that has no parents a class containing it at offset 0 is constructed.

After a set of classes is constructed, inheritance relation inference is performed. For each class a set of possible direct base classes is constructed. Class Base can be a direct base of class Derived if it can be "placed" inside class Base so that overlapping vtables are either equal or connected by direct inheritance. See fig. 1 for an example.

Normally, there exists only one way to "cover" a class with base classes. In case there are several coverings, then the one

with the largest number of vtables being equal to the vtables of the class at hand is selected. If there are several such coverings, the case is reported to the user.

As a result, direct base classes with offsets to the corresponding instances, constructors, destructors, and vtables are reconstructed for each class in a program.

V. Implementation and experimental results

Presented methods have been implemented as a plugin for IDA Pro interactive disassembler. The plugin is available for download at http://decompilation.info.

The plugin has been tested on a variety of open-source software written in C++. The process used is as follows. First, the program is compiled with optimizations and RTTI and RTTI-aware class hierarchy reconstruction algorithm is used to collect information on polymorphic class hierarchy. The program is then recompiled with optimizations and debug information but without RTTI. Class hierarchy reconstruction algorithm that does not use RTTI structures is applied, and debug information is used to restore the correspondence between vtables and actual polymorphic classes. Results of the two class hierarchy reconstruction algorithms are then compared.

Application	doxygen	Shareaza	Notepad++
Running time	7.9s	18.9s	0.7s
Vtables found	415	1128	95
Vtable mismatches	8.6%	4.1%	4.0%
Classes found	401	1108	95
Non-classes	0.9%	0.6%	0%
Class mismatches	9.7%	6.5%	10.5%

Fig. 2. Test results.

Test results are presented on fig. 2. Tests were performed on a Core2Duo CPU at 2.8Ghz.

For each of the analyzed applications all vtables present in the assembly have been found, and no false positives were registered. “Vtable mismatches” refer to vtables with reconstructed parent vtable differing from the real one. “Non-classes” refer to reconstructed classes that were not present in the source program. “Class mismatches” refer to classes with associated vtables or parents reconstructed differently from the real ones.

All non-classes were created as a result of allocating several polymorphic classes on the stack, where they were treated as a single class.

In some of the considered mismatch cases the mismatch was caused by violation of one of presumptions the algorithm relies on (such as usage of virtual inheritance).

Most of vtable mismatch cases were registered in classes with no data members and no actions performed in constructors and destructors. Hierarchies of such classes can be rearranged in virtually any way without changing the semantics of the program. That’s why such mismatches are not actual errors.

Most of class mismatches were registered as a result of one class including the other as a field. As it was denoted in chapter IV-B, inheritance and inclusion as a field are normally indistinguishable. That’s why such mismatches do not lead to misinterpretation of program semantics.

To further decrease the mismatch rates, additional analysis is required, such as dynamic analysis and deeper analysis of the actions performed in constructors and destructors.

VI. Conclusion and further work

A method for automatic reconstruction of polymorphic class hierarchies from assembly code obtained by compiling a C++ program is presented. If the source C++ program is compiled with run-time type information enabled, then the class hierarchy is reconstructed exactly in the same form as it was in the source code.

If RTTI is not compiled into the assembly, reconstruction method is based on the analysis of virtual function tables, constructors and destructors. First, inheritance relation on a set of virtual tables is reconstructed using the analysis of virtual function tables and virtual function table accesses. Then additional analysis of constructors and destructors is used to restore classes and relations between them. Multiple inheritance is handled correctly.

Proposed methods were implemented as a plugin for IDA Pro interactive disassembler, which is available for download. The plugin has been tested on a variety of open-source software written in C++ showing low mismatch rates.

Directions for future work include reconstruction of exception handling blocks and other C++ features. The main goal is to develop several techniques that could be implemented as a set of tools for reconstructing low-level programs into C++ programming language.

References

[1] M. Van Emmerik, “Static single assignment for decompilation,” Ph.D. dissertation, The University of Queensland, Brisbane, Australia, 2007.

[2] P. Sabanal and M. Yason, “Reversing c++,” *Black Hat DC*, February 2007.

[3] B. Stroustrup, *The C++ Programming Language, 3rd edition*. Reading, MA: Addison-Wesley, 1997.

[4] Ida pro disassembler. [Online]. Available: http://www.datarescue.com/

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

[6] M. Van Emmerik and T. Waddington, “Using a decompiler for real-world source recovery,” *Proceedings of 11th Working Conference on Reverse Engineering*, pp. 27–36, November 2004.

[7] Itanium c++ abi. [Online]. Available: http://www.codesourcery.com/public/cxx-abi/abi.html

[8] B. Stroustrup, “A history of c++: 1979-1991,” *Proceedings of the second ACM SIGPLAN conference on History of programming languages*, pp. 271–297, April 1993.

[9] *ISO/IEC Standard 14882:2003. Programming languages - C++*. New York: American National Standards Institute, 2003.

[10] J. Gray. (1994) C++: Under the hood. [Online]. Available: http://msdn.microsoft.com/archive/en-us/dnarvc/html/jangrayhood.asp

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2nd ed. Addison-Wesley, August 2006.

[12] H. Sutter, “Sutter’s mill: Virtuality,” *C/C++ Users Journal*, vol. 19, no. 9, pp. 53–58, 2001.

[13] Class hierarchy reconstruction ida pro plugin. [Online]. Available: http://decompilation.info/