Reconstruction of C++-specific constructs for decompilation

A. Fokin
Moscow State University
Computational Math. and Cybernetics Dept.
Leninskie Gory, Moscow, Russia
apfokin@gmail.com

Y. Derevenets
Institute for System Programming
Russian Academy of Sciences
25, Alexander Solzhenitsyn st., Moscow, Russia
yegord@ispras.ru

K. Troshina
Institute for System Programming
Russian Academy of Sciences
25, Alexander Solzhenitsyn st., Moscow, Russia
katerina@ispras.ru

A. Chernov
Moscow State University
Computational Math. and Cybernetics Dept.
Leninskie Gory, Moscow, Russia
cher@unicorn.cmc.msu.ru

Abstract

This paper presents several methods useful for decompilation of C++ programs. These methods allow automatic reconstruction of polymorphic class hierarchies and exception raising and handling statements.

For reconstruction of polymorphic class hierarchies a new method that does not use run-time type information is presented. It relies only upon the way virtual table pointers are laid out inside objects and is therefore compiler-independent to a large degree.

Depending on the application binary interface used by the compiler, exception handling statements are located in the code using data flow analysis and analysis of static exception handling structures.

A tool for automatic reconstruction of polymorphic class hierarchies and exception raising and handling statements that implements the described techniques is presented. This tool is implemented as a plugin for IDA Pro Interactive Disassembler. Experimental study of the tool performance is provided.

1 Introduction

In this work we present several methods useful for decompilation of C++ programs. These methods allow automatic reconstruction of polymorphic class hierarchies and exception raising and handling statements. We presume that no modifications of the assembly code were performed after compilation (such as assembly-level obfuscation). The problem of obtaining assembly code from an executable file lies outside of the scope of this work.

For reconstruction of polymorphic class hierarchies we presume that the program was compiled without RTTI (run-time type information), as the opposite case is already well-studied [6, 10, 12]. Besides, RTTI is considered to be frequently misused [13], and some modern applications written in C++ refrain from using it.

An approach based on the analysis of vtables (virtual function tables), constructors and destructors is used for reconstruction of polymorphic class hierarchies. First, vtables are identified in the assembly. Then the functions that work with pointers to vtables are analyzed and classified as either constructors or destructors. At the same time, inheritance relation on a set of vtables is reconstructed. Correspondence between vtables and actual classes is reconstructed via analysis of constructors. Obtained information is then used for inference of the inheritance relation on a set of classes.

The presented method relies only upon the way vtable pointers are laid out inside objects. Since this part of C++ ABI (application binary interface) is implemented in the same way by most modern compilers, the presented method is compiler-independent to a large degree.

As the implementation of the exception handling mechanism may differ greatly depending on the ABI used by a compiler, several ABIs are considered. Data flow analysis is used for identifying exception handling statements in binaries compiled with Microsoft Visual C++ compiler. After static exception handling structures are parsed, **try** and **catch** block boundaries are reconstructed. GCC on x86 and x64 architectures uses Dwarf2 table-based unwinding mechanism by default. Static exception handling data used by Dwarf2 can be easily located and parsed. Additional

analysis then allows to reconstruct the boundaries of **try** and **catch** blocks.

The presented methods were implemented in a plugin for IDA Pro interactive disassembler [1] and were tested on a variety of open-source C++ software. We used MSVC 10.0 and g++ 4.4 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 2 discusses related work. A new method for class hierarchy reconstruction is presented in Section 3. Section 4 presents methods for reconstruction of exception raising and handling statements. Experimental results are discussed in Section 5. Our conclusions and directions for future work are presented in the last section.

2 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 [11, 12] has given a detailed description of RTTI and exception handling structures used by MSVC, along with the implementation details of some of the C++ concepts, such as constructors and destructors. He has presented several tools, which were implemented as scripts for IDA Pro interactive disassembler. The first tool performs automatic analysis of RTTI structures and can be used for reconstruction of polymorphic class hierarchies. The second tool is used for automatic recognition of exception handling statements in the assembly code. It does not reconstruct the boundaries of **try** blocks. These tools are based on pattern matching and do not always provide correct results and cannot be used with compilers other than MSVC.

Kocchar [8] has also examined details of the exception handling mechanism used in MSVC. He has developed his own exception handling library that replaces the default one.

Sabanal and Yason [10] along with RTTI-based approach to class hierarchy reconstruction 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. 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. 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. The presented techniques also heavily rely on the usage of MSVC-specific **__thiscall** calling convention.

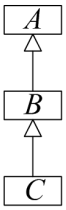

Figure 1. Example of a class hierarchy.

In [6] authors have proposed a method for automatic reconstruction of class hierarchies that does not rely on RTTI and performs well in cases when aggressive optimizations are used by the compiler. In this work we present a modification of this method that performs much faster while offering comparable error rates.

3 Class hierarchy reconstruction

For the reconstruction of polymorphic class hierarchies, vtables are located first as described in [6].

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". For example, for the hierarchy on Fig. 1 the following is true: $C \rhd A$, $C \rhd B$ and $B \rhd A$.

Class hierarchy reconstruction is performed via analysis of constructors and destructors. Constructor of a class performs the following sequence of operations [4, 7]:

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.

Note that if multiple inheritance is used, a class may have several vtable pointer fields, one for each polymorphic

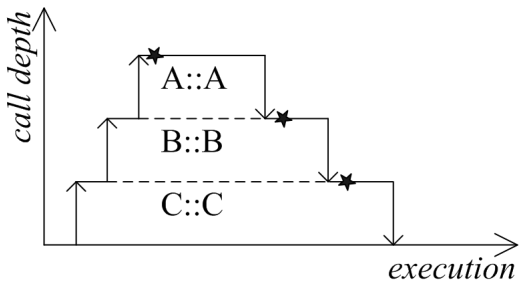

Figure 2. Outline of constructor execution for class C from Fig 1. Initializations of vtable pointer field are marked with ★.

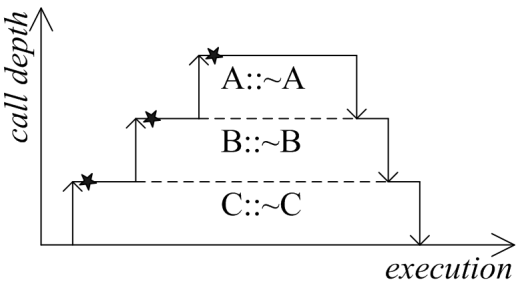

Figure 3. Outline of destructor execution for class C from Fig 1. Initializations of vtable pointer field are marked with ★.

base. In case of a "deep" inheritance hierarchy, construction and destruction of an object may require many successive initializations of vtable pointer field(s). Experiments have shown that neither GCC nor MSVC optimize away the last assignment in a non-inlined destructor. That means that non-inlined destructor of an object always overwrites each vtable pointer field with a pointer to the corresponding "most-base" vtable.

3.1 Localization of constructors and destructors

Each vtable is referenced only from constructors and destructors of the corresponding class — its address is written into vtable pointer field of the object being constructed or destructed.

Constructors and destructors can be inlined, so there can be several constructor and destructor calls inside a single assembly subroutine. To address this problem, an interprocedural data flow analysis as described in [5] is used. It is employed to detect code sites where one or more memory locations that differ from each other by a constant offset are overwritten with pointers to vtables. These code sites are referred to as *vtable access sites*. Each vtable access site is either a constructor or a destructor and is uniquely identified with a set of addresses of the instructions that initialize vtable pointer field(s). Inclusion of vtable access sites is defined in terms of inclusion of these address sets.

Note that vtable access sites are likely to span several subroutines. In this case several vtable access sites are created: one for the enclosing subroutine and one for the nested subroutine, the former including the latter. It is performed this way because vtable access site in a nested subroutine clearly corresponds to a different constructor or destructor than the one in the enclosing subroutine, and the bigger the number of recognized constructors and destructors, the easier class hierarchy reconstruction.

Consider an example on Fig. 3. As can be seen, vtable access site for C::~C includes vtable access site for B::~B because C::~C calls B::~B for **this** object.

Every vtable access site is associated with a set of pairs (*positive offset*, *vtable sequence*). Vtables are stored in *vtable sequence* in an order in which their addresses are written at the corresponding *positive offset*.

Consider an example on Fig. 3. Analysis of destructor for class C will generate three vtable access sites with the following sets of pairs associated with them: $\{(0,(C,B,A))\}$ (for C::~C), $\{(0,(B,A))\}$ (for B::~B), $\{(0,(A))\}$ (for A::~A).

3.2 Vtable inheritance reconstruction

First we consider a case where multiple inheritance is not used. In this case each vtable access site is associated with only one pair (*positive offset*, *vtable sequence*).

The order in which vtable pointer field initializations are performed is determined by the inheritance order. Consider a hierarchy on Fig. 1. Outlines of constructor and destructor execution for class C are provided on Fig. 2 and 3 correspondingly. As can be seen, in a call to constructor vtable pointer field is overwritten in a "base-to-derived" order, while in a call to destructor it is overwritten in a "derived-to-base" order.

Since each vtable access site corresponds to either a constructor or a destructor, the inheritance direction in a *vtable sequence* is either "base-to-derived" (for constructors) or "derived-to-base" (for destructors). For example, consider a vtable access site for C::C with an associated pair $(0,(A,B,C))$ (see Fig. 2). It is a constructor, therefore the inheritance direction in (A,B,C) is "base-to-derived", i.e. $A \lhd B \lhd C$. This matches the actual inheritance direction (Fig. 1). Therefore, the following rule can be applied.

Rule 1. *If vtable access site is associated with a pair*

(*offset*, $(A_1, \ldots, A_n)$), *then either* $A_1 \lhd \ldots \lhd A_n$ *or* $A_1 \rhd \ldots \rhd A_n$.

As can be seen on Fig. 2 and 3, in a call to a constructor, vtable pointer field is initialized **after** a call to the constructor of the base class, while in a call to a destructor, vtable pointer field is initialized **before** a call to the destructor of the base class. Therefore each vtable access site that spans several assembly subroutines can be identified as either a constructor or a destructor using the following rule.

Rule 2. *Let some vtable access site span several assembly subroutines. If the initialization of vtable pointer field precedes the call to the nested subroutine that also initializes that field, then this vtable access site is a destructor. Otherwise, it is a constructor.*

It can happen that all nested constructor or destructor calls were inlined. In this case all initializations of vtable pointer field of vtable access site are located in one assembly subroutine and this rule cannot be applied. Instead, the following rules are used.

Rule 3. *If one vtable access site contains the other one, then either both of them are constructors, or both of them are destructors.*

As it was denoted in Section 3.1, vtable access site inclusion is defined in terms of inclusion of associated sets of addresses of instructions that initialize vtable pointer field(s). Rule 3 is a direct consequence of the fact that a constructor cannot call a destructor for **this** object, as well as a destructor cannot call a constructor for **this** object.

Rule 4. *If vtable access site is associated with a pair* (*offset*, $(A_1, \ldots, A_{k_1}, \ldots, A_{k_2}, \ldots, A_n)$), *and the size of vtable* A_{k_1} *is less then the size of vtable* A_{k_2}, *then* $A_1 \lhd \ldots \lhd A_n$.

According to the C++ inheritance rules [4], if there are more virtual functions in vtable A_{k_2} than in vtable A_{k_1}, then A_{k_2} cannot be a base of A_{k_1}. Rule 2 is a direct consequence of this fact applied to Rule 1.

Rule 5. *If a vtable access site is associated with a pair* (*offset*, $(A_1, \ldots, A_{k_1}, \ldots, A_{k_2}, \ldots, A_n)$), *and* i*-th virtual function in vtable* A_{k_1} *is pure and* i*-th virtual function in vtable* A_{k_2} *is not, then* $A_1 \lhd \ldots \lhd A_n$.

This rule is a direct consequence of Rule 1 and the fact that in C++ it is impossible to override a virtual function that is not pure with a pure one [4].

Application of each of the Rules 1-5 gives an inheritance relation on vtables in the corresponding vtable sequence. Thus, by applying Rules 1-5 to vtable access sites, inheritance relation on a set of vtables is reconstructed.

Obviously, there may be cases when these rules won't be sufficient to completely reconstruct inheritance relation on a set of vtables. However, as it is shown in Section 5, in practice such cases are nearly nonexistent.

If multiple inheritance is used, then almost the same rules can be applied. The only difference is that instead of one (*positive offset*, *vtable sequence*) pair each vtable access site will have several. Modified rules for multiple inheritance are not presented here as they can be easily derived from the ones for single inheritance.

3.3 Polymorphic class hierarchy reconstruction

Constructors and destructors 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 classified as constructors.

Classes are reconstructed via analysis of vtable access sites that are constructors. Classes are uniquely identified with a set of (*offset*, *vtable*) pairs, *offset* being the offset of *vtable* in this class. This set of pairs can be constructed by picking the last vtable from each of vtable sequences associated with a class.

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 it is presumed that all vtables belonging to a class were generated as a result of inheritance.

Constructors 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 as described in [6]. As a result, for each polymorphic class in a program the following is reconstructed:

- vtables;
- direct base classes;
- offsets to the instances of direct base classes inside **this** object;
- constructors and destructors.

4 Reconstruction of exception raising and handling statements

C++ standard defines semantics of exception raising and handling, but leaves its implementation up to compiler vendors. The way exception handling is implemented depends on the ABI used by the compiler.

Exception handling in C++ consists of the following steps:

1) throwing an exception;
2) stack unwinding;
3) execution of exception handler.

All the information necessary to perform these steps must be present in the assembly program.

4.1 Exception handling in MSVC

In MSVC C++ exception handling is implemented on top of OS-supplied Structured Exception Handling (SEH) mechanism. For each thread a stack of EXCEPTION_REGISTRATION structures is maintained, with **fs:[0]** pointing to the top of it. Each such structure contains a pointer to an exception handling routine. When exception is raised, control is transferred to the kernel, which then calls the relevant exception handling routine [9]. Normally, EXCEPTION_REGISTRATION structure is allocated on the stack at the beginning of a function and **fs:[0]** is modified to point to it.

For C++ exception handling extended EXCEPTION_REGISTRATION structure is used [11].

```
struct EXCEPTION_REGISTRATION {
  /** Previous element in a stack. */
  EXCEPTION_REGISTRATION *prev;
  /** Exception handling routine. */
  void *handler;
  /** Current unwind state id. */
  int id;
  /** Saved ebp register. */
  DWORD ebp;
};
```

When an exception is thrown during execution of a function, destructors for stack-allocated objects must be called. Unwind state field of EXCEPTION_REGISTRATION structure is an integer describing the set of objects that must be destroyed when an exception is thrown. It is modified every time an object is constructed or destructed and when execution enters **try** and **catch** blocks. Normally, when an object is constructed, unwind state is incremented by one, and when it is destructed, it is decremented by one.

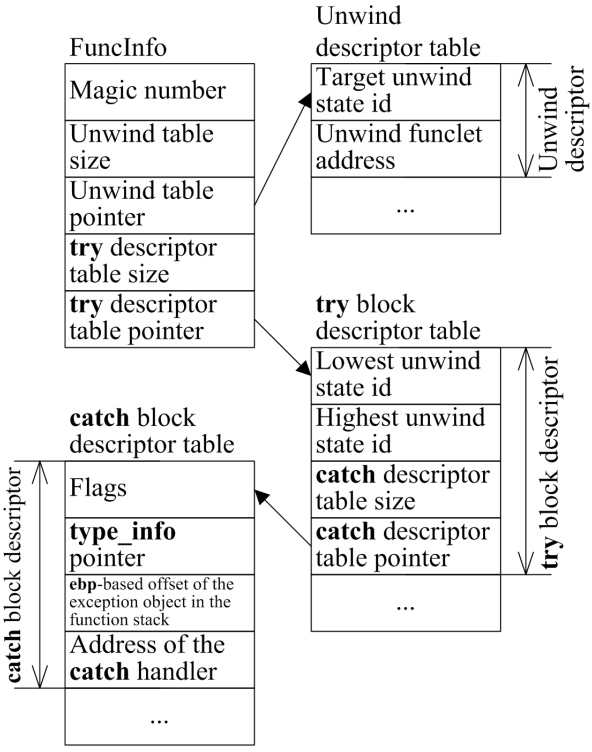

Figure 4. Structures used by MSVC for exception handling.

For each function MSVC registers a SEH exception handler which ends with a call to __CxxFrameHandler function. This function takes a pointer to FuncInfo structure in **eax** register. FuncInfo structure fully describes all **try** and **catch** blocks and unwindable objects in a function (see Fig. 4). This structure contains [11]:

- pointer to the table of unwind descriptors. Elements of this table describe which destructors must be called for each unwind state;
- pointer to the table of **try** block descriptors.

In turn, each **try** block descriptor contains the following information:

- the highest and the lowest unwind states inside the **try** block;
- pointer to the table of associated **catch** block descriptors for this **try** block. Each element in this table contains a pointer to the **type_info** of the exception class and a pointer to the catch handler.

Each catch handler returns the address where to resume execution, i.e. the position just after the **try** block.

4.1.1 Reconstruction of exception handling statements

To reconstruct **try** and **catch** blocks the following procedure is used:

- Locations in the program where EXCEPTION_REGISTRATION structure is allocated on the stack are found. They are simple to detect via data flow analysis as the allocation of this structure involves copying of the value of **fs:[0]** to the stack (prev field of EXCEPTION_REGISTRATION structure is initialized with it).
- Data flow analysis also makes it possible to find initial values of the other fields of EXCEPTION_REGISTRATION structure, including the pointer to exception handling routine.
- Data flow analysis of exception handling routine is performed to find the value of **eax** register before a call to __CxxFrameHandler function. This value is a pointer to the statically allocated FuncInfo structure.
- FuncInfo structure is parsed, thus giving the addresses of unwind and catch handlers.
- Value of the id field of EXCEPTION_REGISTRATION structure at each point in the function is found via data flow analysis. **try** block boundaries are then restored by inspecting the state range of **try** block descriptor and the addresses returned by corresponding **catch** handlers.

4.1.2 Reconstruction of exception raising statements

throw statements are translated by the compiler into calls of _CxxThrowException function.

```
void _CxxThrowException(
  void *exception,
  ThrowInfo *throwInfo
);
```

This function raises a SEH exception with custom parameters that include pointers to the exception object and its ThrowInfo structure [11].

```
struct ThrowInfo {
  /** Const-volatile attributes. */
  DWORD attributes;
  /** Exception destructor. */
  void (*destructor)();
  /** Forward compatibility handler. */
  DWORD compat;
  /** List of types that can catch
   * this exception, i.e. the actual
   * type and all its ancestors. */
  CatchableTypeArray *catchableTypes;
};
```

If **catch** block's parameter type is a base class and exception is its derived class, then this **catch** block should still be invoked. That's why CatchableTypeArray contains **type_info** pointers for exception class and all its direct and indirect bases. When checking whether **catch** block can process the thrown exception, **type_info** of the **catch** block's parameter is compared with all the **type_info**'s available through the ThrowInfo structure. If any match is found then the **catch** block is invoked.

To reconstruct **throw** statements it is sufficient to locate the calls to the _CxxThrowException function in the code, find the values of its parameters via data flow analysis, and parse the statically allocated ThrowInfo structure.

4.2 Exception handling in GCC

GCC under x86 and x64 architectures uses Dwarf2 table-based unwinding mechanism by default. 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*. Landing pad is a code block that calls destructors and transfers execution to the corresponding **catch** block. Information on call sites and landing pads is statically allocated and is present in the assembly. Details on how exception handling is performed using these structures is given in [2].

.eh_frame section of the assembly contains information on stack frames that is needed for stack unwinding. Its format is described in [3]. Frame descriptor for each function contains a pointer to the Language Specific Data Area (LSDA) containing exception handling information (see Fig. 5). LSDAs are stored in **.gcc_except_table** section.

For each call site LSDA contains:

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

Each action record contains an index of an element in the type table, which consists of pointers to **type_info** structures. List of action records in LSDA describes exception types that can be handled by the landing pad.

Information stored in the statically allocated structures in **.eh_frame** and **.gcc_except_table** sections must be parsed in order to reconstruct **try** and **catch** blocks. This presents no difficulties as the format of these structures is documented.

Before the runtime transfers control to a landing pad, it finds the index of exception's **type_info** in the type table. This index is passed in **edx** register to the landing pad. Dispatch to the **catch** block body is performed via a **switch** on this index.

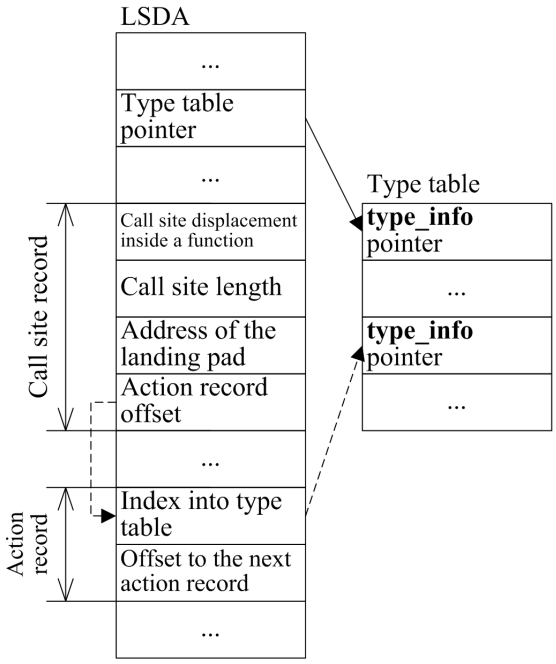

Figure 5. Structures used by GCC for exception handling.

```
; Destructor calls skipped.
; Exception pointer is passed in eax.
mov [ebp - 24], eax
; Index in the type table is passed in edx.
mov [ebp - 28], edx
cmp [ebp - 28], 2
je catch_block_2
cmp [ebp - 28], 1
je catch_block_1
mov eax, [ebp - 24]
mov [esp], eax
call _Unwind_Resume
```

Figure 6. Example of a landing pad.

Consider an example of the landing pad on Fig. 6. Since each **catch** block starts with a call to __cxa_begin_catch and ends with a call to __cxa_end_catch , **catch** blocks can be reconstructed by examining the landing pad and the locations it references.

Different destructors must be called when exception is thrown from different call sites. That's why the compiler generates several landing pads for each **try** block. However, the part of the landing pad that performs dispatch to the **catch** block (the part on Fig. 6) is shared by all landing pads for all call sites of a single **try** block.

Therefore 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. Extents of the **try** block are then reconstructed by uniting all its corresponding call sites.

Exception raising in GCC is performed via a call to the __cxa_throw function.

```
void __cxa_throw(
  void *exception,
  type_info *typeInfo,
  void (*destructor)(void *)
);
```

To reconstruct throw statements it is sufficient to locate the calls to __cxa_throw, find the values of its parameters via data flow analysis, and parse the statically allocated **type_info** structure.

Note that **type_info** structure used by GCC provides all the necessary information on the base classes [2]. That's why there is no need to supply a list of **type_info** pointers for all the bases of exception class as it is in case of MSVC.

5 Implementation and experimental results

The presented methods were implemented in a plugin for IDA Pro interactive disassembler. The plugin is available for download at `http://decompilation.info`.

The plugin was tested on a variety of open-source software written in C++. The process used for testing class hierarchy reconstruction correctness 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. This algorithm always provides correct results [6]. The program is then recompiled with optimizations and debug information but without RTTI. Class hierarchy reconstruction algorithm described in Section 3 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++	mysqld
Running time (old)	7.9s	18.9s	0.7s	n/a
Running time	3.2s	4.1s	0.6s	7.2s
Vtables found	415	1128	95	950
Non-vtables	0%	0%	0%	0.2%
Vtable mismatches	8.6%	4.1%	4.0%	4.2%
Classes found	401	1108	95	974
Non-classes	0.9%	0.6%	0%	3.4%
Class mismatches	9.7%	6.5%	10.5%	8.0%

Figure 7. Test results.

Test results for doxygen (source code documentation generator), notepad++ (text editor), shareaza (peer-to-peer file sharing client), and mysqld (server daemon of MySQL relational database management system) are presented on Fig. 7. Tests were performed on an Intel Core 2 CPU running at 2.8Ghz.

"Running time (old)" refer to the running time of the previous version of the plugin that was presented by the authors in [6]. As can be seen, new method for class hierarchy reconstruction performs in our tests up to five times faster.

For each of the analyzed applications all vtables present in the assembly were found. "Non-vtables" refer to reconstructed vtables that were not present in the source program. Static arrays of function pointers that are used in the same way as vtables fall into this category.

"Non-classes" refer to reconstructed classes that were not present in the source program. Non-classes in all tests were created as a result of allocating several polymorphic classes on the stack, where they were treated as a single class. As it was described in Section 3.1, classes are recognized by tracking accesses to memory locations that differ from each other by a constant offset. That's why in case several polymorphic classes are allocated on the stack they are treated as a single class.

Mismatch rates are calculated using a more complex procedure. Consider the following categories of classes.

- Classes that do not override any of the virtual functions of their bases. Vtables for such classes can be optimized away by the compiler, thus leading to incorrectly reconstructed bases for derived vtables. As this doesn't lead to any changes in the semantics of the program, such cases are not treated as mismatches.
- 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. Such cases are not treated as mismatches.

"Vtable mismatches" refer to vtables where the reconstructed parent vtable contradicts to the real one. For example, if a vtable was reconstructed as not having a parent, while it actually has one, then this is a mismatch. Most vtable mismatches were registered as a result of parent vtable being located in a shared library. The reason for this is that the plugin currently does not support simultaneous analysis of several assemblies.

"Class mismatches" refer to classes where reconstructed vtables or parents contradict to the real ones. Most of class mismatches fall into the following two categories:

- Classes that contain mismatched vtables.
- Classes that include other polymorphic classes as a field. As it was denoted in Section 3.3, inheritance and inclusion as a field are normally indistinguishable.

```
mov     [ebp + 4], 0   ; try {
push    ecx
mov     ecx, esi
call    loc_4011B0
mov     [ebp + 4], 0FFFFFFFFh ; } // try
jmp     short loc_401338
mov     [ebp + 16], esp ; catch(...) {
; ... body skipped ...
mov     eax, offset loc_40132F
ret ; } // catch(...)
```

Figure 8. Example of try and catch block reconstruction for MSVC.

That's why such mismatches do not lead to misinterpretation of program semantics.

Testing of exception raising and handling statement reconstruction was done manually. Several routines from the above-mentioned applications that use exception handling were selected and processed with the algorithm described in Section 4. **try** and **catch** block borders and **catch** block parameter types were then compared to the real ones. In all performed tests borders of **try** and **catch** blocks and **catch** block parameter types were reconstructed correctly.

As the current version of the plugin is not a full-fledged decompiler, results of exception raising and handling statements reconstruction can be displayed either as comments in the assembly listing, or as C++ code with inline assembly. Example of the first approach is presented on Fig. 8.

6 Conclusion and further work

Several methods useful for decompilation of C++ programs are presented. These methods allow automatic reconstruction of polymorphic class hierarchies and of exception raising and handling statements.

For reconstruction of polymorphic class hierarchies a new method that does not use RTTI is presented. It is based on analysis of virtual function tables, constructors and destructors. Assembly is first scanned for virtual function tables. Inheritance relation on a set of virtual function tables is then reconstructed using analysis of code sites that work with pointers to virtual function tables. Additional analysis of constructors and destructors is then used to restore classes and relations between them. Multiple inheritance is handled correctly. The presented method is compiler-independent to a large degree.

For reconstruction of exception raising and handling statements, ABIs used by MSVC and GCC are considered. For each ABI an overview of the exception handling mech-

anism is provided. Detailed description of the methods for reconstruction of **try** and **catch** blocks and **throw** statements is also provided.

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

Directions for future work include integration of the presented methods into a full-fledged decompiler. The main goal is to develop a set of tools for reconstructing low-level programs into C++ programming language.

References

[1] Ida pro disassembler, http://www.datarescue.com/.

[2] Itanium c++ abi, http://www.codesourcery.com/public/cxx-abi/abi.html.

[3] Linux standard base core specification 3.0rc1, chapter 8. exception frames, http://refspecs.freestandards.org/lsb_3.0.0/lsb-core-generic/lsb-core-generic/ehframechpt.html.

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

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

[6] A. Fokin, K. Troshina, and A. Chernov. Reconstruction of class hierarchies for decompilation of c++ programs. *Proceedings of 14th European Conference on Software Maintenance and Reengineering*, pages 249–252, March 2010.

[7] J. Gray. C++: Under the hood, http://msdn.microsoft.com/archive/en-us/dnarvc/html/jangrayhood.asp, March 1994.

[8] V. Kochhar. How a c++ compiler implements exception handling, http://www.codeproject.com/kb/cpp/exceptionhandler.aspx, April 2002.

[9] M. Pietrek. A crash course on the depths of win32 structured exception handling. *Microsoft Systems Journal*, January 1997.

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

[11] I. Skochinsky. Reversing microsoft visual c++ part 1: Exception handling, http://www.openrce.org/articles/full_view/21, March 2006.

[12] I. Skochinsky. Reversing microsoft visual c++ part 2: Classes, methods and rtti, http://www.openrce.org/articles/full_view/23, September 2006.

[13] B. Stroustrup. A history of c++: 1979-1991. *Proceedings of the 2nd ACM SIGPLAN Conference on History of Programming Languages*, pages 271–297, April 1993.