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. First, the inheritance relation on a set of virtual function tables induced by inheritance relation on a set of classes is reconstructed. Then additional analysis of constructors and destructors is used to reconstruct inheritance relation on a set of classes.

A tool for automatic reconstruction of polymorphic class hierarchies, which 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 fair amount of attention recently. Decompilers have been improving in quality and range of accepted low-level programs. The C programming language was chosen 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 in place 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 of 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. Multiple and virtual inheritance[1] are handled correctly. As run-time type information structures store class names, class names are also recovered.

If the program was compiled without RTTI, an induced 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.

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 Windows. 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

[1]Virtual inheritance and inheritance utilizing virtual functions are two distinct C++ concepts. Virtual inheritance is closely related to multiple inheritance and deals with cases of inheriting the same class more than once. Virtual inheritance does not necessarily define a polymorphic class hierarchy [3].

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 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

A. Run-time type information in C++

Run-time type information in C++ is used for implementing the following operators [3].

- The **dynamic_cast**<> operator. This operator performs type conversions with run-time checking. Normally it is used to convert a pointer to a base class to a pointer to a derived class, or to convert a lvalue referring to a base class to a reference to a derived class. A program can thereby use a class hierarchy safely.
- **typeid** operator. This operator provides a program with the ability to retrieve the actual type of an object referred to by a pointer or a reference.

The **typeid** operator can be applied to non-polymorphic types. In this case it is evaluated at compile time. On the contrary, the **dynamic_cast**<> operator works with polymorphic types only, and results in undefined behavior in case the program was compiled without RTTI.

The layout of RTTI structures used for implementing **typeid** and **dynamic_cast**<> operators is defined by the ABI (application binary interface) that is used by the C++ compiler. The layout varies from compiler to compiler and from platform to platform. For example, MSVC uses an undocumented format, which, however, has been successfully reverse-engineered and can be looked up in source code of Wine or, for example, in [5]. The ABI used by GCC can be looked up in [7]. The layout of RTTI structures used in GCC is defined in the <cxxabi.h> header file, which is supplied with the compiler.

In this paper we presume that the format of RTTI structures used by the compiler is known and can be parsed.

B. Class hierarchy reconstruction

In case RTTI structures are present in the assembly, the process of class hierarchy reconstruction can be split into the following steps.

1) Locate RTTI structures.
2) Parse RTTI structures.
3) Use obtained information to reconstruct a polymorphic class hierarchy.

As there are quite few different RTTI layouts, it is practical to repeat the above steps several times, once for each layout, and then choose the most plausible variant manually.

In certain cases it is possible to determine the compiler used (and therefore, the ABI). For example, it can be determined by analyzing the standard library the program uses. If the standard library is linked dynamically, the compiler can easily be determined given the shared library file. If the standard library is linked statically, then its vendor can be determined via function signature comparison.

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.

C. Locating the RTTI structures

In all ABIs known to the moment pointer to the RTTI structure of a class always precedes the corresponding vtable. Therefore, the problem of finding the RTTI structures can be reduced to a problem of locating vtables. For each vtable the following statements hold.

- Vtable is an array of pointers to functions.
- Only the first element of the vtable is referenced from the program code — its address is used in constructors and 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 code segment and its value is a pointer to a function, then it is marked as a start of vtable. Otherwise, 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 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, uniqueness of vtables for each class is not guaranteed, i.e. the same vtable may be shared by several different classes. For example, if some class B inheriting from class A does not override any of the virtual functions of class A, then classes A and B can share the same vtable. This is impossible if RTTI is used, since RTTI structure is accessed through vtable pointer field, and class A and class B are distinct classes and their corresponding RTTI structures differ.

We can presume that induced 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 chapter III-C. 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. In this case we also say that vtable D inherits directly from vtable B, and that vtable D is a direct child of vtable B. 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".

Note that $\rhd$ is a relation that defines a strict partial order on a set of all vtables. 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 $B \not\lhd D$ and $B \sim D$ are satisfied, then condition $D \rhd B$ is also satisfied, and vice-versa. In terms of relations it means that relation $\rhd$ is an intersection of relations $\not\lhd$ and $\sim$.

Information on inheritance relation between two vtables can be stored as a three-bit field, with each bit indicating the presence of one of the relations $\not\rhd$, $\not\lhd$ and $\sim$. These bits are referred to as $i_{\not\rhd}$, $i_{\not\lhd}$ and $i_{\sim}$. For example, if for vtable pair (D, B) all bits are set to zero, then nothing is known about the inheritance relation between vtables D and B. If bits $i_{\not\lhd}$ and $i_{\sim}$ are set, then it is known that $D \rhd B$. If bits $i_{\not\lhd}$ and $i_{\not\rhd}$ are set, then it is known that vtables D and B do not inherit from each other. All three bits being set is an indication of a conflict. A three-bit field that encodes a relation between two vtables getting into conflict state during reconstruction can be a consequence of some of the presumptions on the structure of the program being false. The details on resolving such conflicts are provided in the description of the algorithm in chapter IV-C.

A_i denotes i-th virtual function in a vtable A. The pointer in a virtual function table may point to an *adjuster thunk* that adjusts the passed **this** pointer and then transfers control to the actual function body. In case there is no adjuster thunk, zero adjustment value is presumed. A_i^{adj} denotes **this** adjustment value of i-th virtual function in vtable A.

A. Retrieving inheritance relation

Several simple rules are used for inheritance relation reconstruction. 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 the 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[2] 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]. For pure virtual functions both GCC and MSVC store a pointer to stub function in a vtable. This stub function terminates a program with appropriate error message. It can be reliably identified via signature matching.

If it is possible to reliably determine the size of the parameters of a virtual function (for example, if the 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 ($B \not\rhd D \wedge B \not\lhd D$).*

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

The fact that some function appears on the same position in two vtables means that it was either inherited by one of these vtables from another, or it was inherited from some common base, or this situation was a result of optimization.

[2] A virtual function is pure if it has no body, and therefore cannot be called. See [3] for details.

```
class A {
public:
  virtual void f() { /* ... */ }
};

class B: public A {
public:
  virtual void f() { /* ... */ }
};

class C: public B {
public:
  virtual void f() { A::f(); }
};

class D {
public:
  virtual void f() {
    ((A*)this)->A::f();
  }
};
```

Fig. 1. Example program for rule 4.

Experiments with GCC and MSVC have shown that neither GCC nor MSVC perform pairwise function comparison during compilation to eliminate identical functions. However, during compilation of a single compilation unit, MSVC eliminates identical functions with simple bodies like **return** */* const */*. Further experiments have shown that there is a limit on the length of the assembly code of a function that can be eliminated this way. A function is called *primitive* if its assembly code is shorter than this limit, i.e. it could have been generated as a result of identical function elimination.

Rule 4. *Let* **f** *be a set of all vtables having virtual function B_i on i-th position with* ***this*** *adjustment value of B_i^{adj}. Obviously, vtable B lies in* **f**. *Then the subgraph of vtable inheritance hierarchy induced*[3] *by* **f** *is a tree.*

There are several cases when this rule cannot be applied. Consider the program on fig. 1. Due to optimizations a pointer to function A::f might have been stored directly in the vtable of class D, which is unrelated to class A, without the body of the function D::f being generated. But such trickery is rare in a real code, because it gives no real benefits while introducing the danger of invoking undefined behavior. While a call to function A::f in function D::f is a "dangerous" C++, a similar function call in C::f is perfectly safe. Due to optimizations a pointer to function A::f might have been stored directly in the vtable of class C. This also breaks rule 4. As it is presented on fig. 2(a), the subgraph of inheritance hierarchy induced by the set $\{A, C\}$ is not a tree. This case is a fine example of *code smell* [11] because the approach taken only complicates the hierarchy, while the intended behavior can be achieved by simply inheriting class C directly from class A. Such cases are also rare in a real code. We will presume that they do not occur at all.

[3]Given graph $G = (V, E)$ and a subset of vertices $U \in V$, the subgraph of G induced by U, denoted as $G[U]$, is a graph with vertex set U and edge set consisting of all the edges from G connecting vertices in U [10].

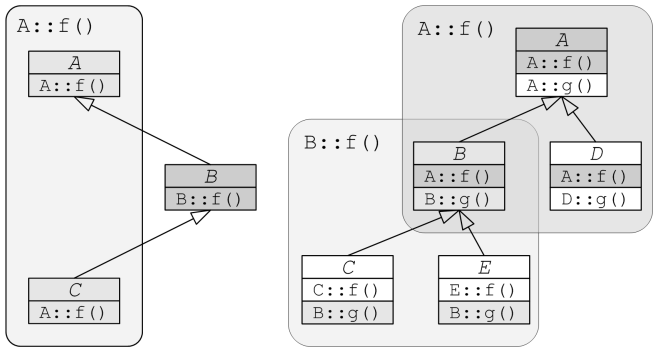

Fig. 2. Examples for rule 4.

Fig. 2(b) illustrates the application of rule 4. On this figure vtables A, B and D have the same function A::f on the first position, and therefore the subgraph of the inheritance hierarchy induced by the set $\{A, B, D\}$ is a tree. The same is true for the set $\{B, C, E\}$ since vtables B, C and E have the same function B::g on the second position.

Some information on inheritance relation can also be gathered using analysis of constructors and destructors. A constructor performs the following sequence of operations [9], [12].

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. The latter case is possible by using the following code:

```
this->~D(); /* destructor */
new (this) D(/* ... */);
    /* placement new */
```

or simply by executing **delete this**. Both cases, however being rare, have been seen in real-life production code.

By using interprocedural data flow analysis as it is described in [13] 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 5. *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 ($D \rhd B$).*

This rule is a direct consequence of the list of actions that must be performed by constructors and destructors. Moreover, if a call to destructor for **this** object is not followed by a call to constructor and vtable pointer field is overwritten several times, then it is possible to infer the inheritance relation between all the vtables referenced. Consider the following disassembly:

```
; this->~D(); /* destructor */
mov dword ptr [esi], offset D::vtable
; D::~D() body goes here
mov dword ptr [esi], offset C::vtable
; C::~C() body goes here
mov dword ptr [esi], offset B::vtable
; B::~B() body goes here
```

Here **this** pointer is stored in **esi** register. Given the pattern of assignments to vtable pointer field, it can be concluded that class D inherits from class C, which in turn inherits from class B.

The same technique cannot be applied if a call to destructor is followed by a call to constructor due to possibility of some assignments to vtable pointer field being optimized away.

```
; this->~D(); /* destructor */
; D::~D() body was empty
mov dword ptr [esi], offset C::vtable
; C::~C() body goes here
; B::~B() body was empty

; new (this) D();
mov dword ptr [esi], offset B::vtable
; B::B() body goes here
; C::C() body was empty
mov dword ptr [esi], offset D::vtable
; D::D() body goes here
```

However, such cases can easily be detected because the last assignment in a constructor always references the vtable being analyzed (vtable for class D in this case).

As advised by almost every guide on object-oriented design, most of polymorphic class hierarchies are designed for polymorphic deletion (i.e. deletion via a pointer to base class), and therefore use virtual destructors [14]. That means that in most cases rule 5 will produce at least one inheritance relation — between the vtable being analyzed and its corresponding "most-base" one.

B. Formalizing the problem

Rules 1-5 are applied, and all consequents are stored as a set of restrictions on the structure of inheritance hierarchy.

Using rule 5 it is possible to construct a set of all vtables in a single inheritance tree with virtual destructors. If virtual destructors are not used, it can still be done using rule 4. There are some cases where it cannot be done correctly. These cases occur if no information on inheritance relation between two vtables can be derived from the assembly. For example, if a base vtable B consists of only one pure virtual function, derived vtable D overrides it, and accesses to B are optimized away in constructors and destructor of a class corresponding to D, then in the assembly there is no evidence of the fact that vtable D inherits from vtable B. Such cases cannot be identified automatically. Presented algorithm produces incorrect results for them.

The set of all vtables is divided into several disjoint sets so that all vtables in a single set form an inheritance tree (connected inheritance hierarchy). Global inheritance hierarchy is reconstructed by processing these sets one by one. Let $\mathbf{V}$ be one of these sets.

Let $\mathbf{R}$ be a set of all restrictions concerning vtables from $\mathbf{V}$ obtained by applying rules 1, 2, 3, and 5, i.e. a set of all restrictions that can be expressed in terms of relations $\rhd$, $\not\rhd$, and $\sim$, and $\mathbf{R}_s$ be a set of all restrictions concerning vtables from $\mathbf{V}$ obtained by applying rule 4.

Since some presumptions have been made on the structure of the source C++ program in chapter IV-A, it may be possible that all obtained restrictions cannot be jointly satisfied. We presume that all restrictions from $\mathbf{R}_s$ can be jointly satisfied. If they are not, additional manual analysis will be required. Since restrictions from $\mathbf{R}$ may not be jointly satisfiable the following problem will be considered.

> Find a set of jointly satisfiable with $\mathbf{R}_s$ restrictions $\mathbf{R}' \subseteq \mathbf{R}$ so that adding any of the elements of $\mathbf{R} \setminus \mathbf{R}'$ to $\mathbf{R}'$ makes it jointly unsatisfiable with $\mathbf{R}_s$. Build a vtable inheritance tree T with nodes from $\mathbf{V}$ that satisfies all the restrictions from $\mathbf{R}'$ and $\mathbf{R}_s$. (1)

C. Reconstructing vtable inheritance hierarchy

Each of restrictions from $\mathbf{R}_s$ has the form *the subgraph of vtable inheritance hierarchy induced by* $\mathbf{f}$ *is a tree*. Let $\mathbf{F}$ be a set of all such sets $\mathbf{f}$. Then, if a tree T is a solution of problem (1), the subgraph of T induced by any of the sets $\mathbf{f} \in \mathbf{F}$ is a tree.

Let set $\mathbf{F}^*$ be a set containing:

- all sets from $\mathbf{F}$,
- all non-empty sets obtained by intersecting several sets from $\mathbf{F}$,
- set $\mathbf{V}$,
- all sets of the form $\{B\}$, where $B \in \mathbf{V}$.

Note that if all subgraphs of vtable inheritance hierarchy induced by elements of $\mathbf{F}$ are trees, then the same is also true for $\mathbf{F}^*$.

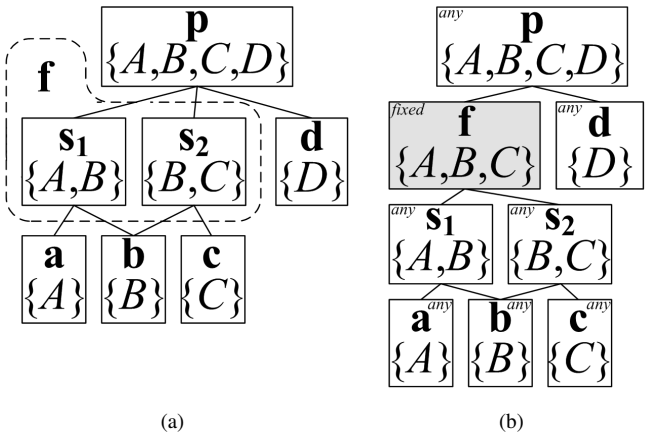

Fig. 3. Example of graphs **H** and **G**.

To construct a solution of the problem (1) graph **G** that "encodes" all the restriction from $\mathbf{R}_s$, and only them, is built. Graph **G** is constructed by modifying graph **H**, which is a Hasse diagram[4] for relation $\subset$ (*is subset of*) on set $\mathbf{F}^*$.

Nodes of Hasse diagram **H** are considered to be sets $\mathbf{f} \in \mathbf{F}^*$, and are assumed to possess all traits of a set — for example, a node can be a subset of another node. For each node in graph **H** a set of child nodes is defined as follows: node **c** is a child of node **p** if there exists an edge in graph **H** connecting **p** and **c**, and $\mathbf{c} \subset \mathbf{p}$.

Then graph **G** is built by adding new nodes to Hasse diagram **H**. For each node **p** of **H** a set of its child nodes is considered. If two child nodes of **p** have non-empty intersection, then these nodes are considered *connected*. The set of child nodes of **p** is then divided into connected components, and a node is added for each connected component consisting of more than one child as it is shown on fig. 3.

Example of Hasse diagram **H** is presented on fig. 3(a). Child nodes $\mathbf{s}_1$ and $\mathbf{s}_2$ of node **p** form a connected component and therefore for these nodes a new node **f** is added, as it is shown on fig. 3(b). Vtable set of this node is the union of all nodes in its corresponding connected component. Note that the resulting graph **G** is also a Hasse diagram.

All newly added nodes are marked as *fixed*and all the rest are marked as *any*. The meaning behind these names is as follows.

- Children of *any*-nodes do not intersect. Therefore, if induced inheritance trees have been built for each child of a given *any*-node, then induced inheritance tree for this *any*-node can be constructed by connecting these trees with edges — total $n-1$ edges, where n is the number of children of this *any*-node.
- Children of *fixed*-nodes intersect. Therefore, induced inheritance trees of the children of a *fixed*-node fully determine the structure of the induced inheritance tree of this *fixed*-node (i.e. its induced inheritance tree is *fixed*).

For the graph **G** the following statement holds.

Statement 1. *If there exists a tree satisfying all the restrictions from* $\mathbf{R}_s$*, then such a tree can be constructed by building induced inheritance trees for any-nodes of graph* **G** *as it was described above. It will satisfy all the restrictions from* $\mathbf{R}_s$*, regardless of how the edges to be added were chosen at each any-node.*

The proof is quite complicated, so it is not provided here. The important idea behind this statement is that the graph **G** "encodes" all the restrictions from $\mathbf{R}_s$, and only them. Solution of the problem (1) is built using this graph.

First, it is checked whether there are no cycles of *any*-nodes in **G**. It can be proven that if such cycle exists, then the problem (1) has no solution. In this case additional manual analysis is required.

Each node of **G** containing only one vtable is defined as a *leaf* node for this vtable. For each node **f** and for each vtable A the shortest path from **f** to the leaf node for A that does not ascend into *fixed*-nodes is found. The second node in this path, which is adjacent to **f**, indicates a *direction* from **f** to A, and is denoted as $\mathbf{D}^{\mathbf{f}}_A$. In case such path does not exist, $\mathbf{D}^{\mathbf{f}}_A$ is considered to be the closest *fixed*-parent of **f**.

Consider an example on fig. 3(b). Here $\mathbf{D}^{\mathbf{p}}_A = \mathbf{f}$, because **a** is a leaf node for vtable A, and the shortest path from **p** to **a** is $(\mathbf{p}, \mathbf{f}, \mathbf{s}_1, \mathbf{a})$. On the other hand, $\mathbf{D}^{\mathbf{a}}_D = \mathbf{f}$, because there exists no path from **a** to **d** that does not ascend into *fixed*-nodes. Therefore, $\mathbf{D}^{\mathbf{a}}_D$ is the closest *fixed*-parent of **a**.

To reconstruct the resulting vtable inheritance tree T, for each two vtables B and D a three-bit field $\mathbf{I}_{(B,D)}$ containing the information on inheritance relation between vtables B and D is iteratively constructed. Also, for each two child nodes $\mathbf{s}_1$ and $\mathbf{s}_2$ of each *any*-node **p** of graph **G**, reconstruction algorithm iteratively builds the following structures.

$\mathbf{N}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$ — a set of all virtual tables that must be inherited by the root vtable R of inheritance tree induced by $\mathbf{s}_1$ ($T[\mathbf{s}_1]$) in case one of R's bases lies in $\mathbf{s}_2$.

$\mathbf{P}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$ — a set of all virtual tables that must not be inherited by the root R of $T[\mathbf{s}_1]$ in case one of R's bases lies in $\mathbf{s}_2$.

$\mathbf{I}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$ — three-bit field containing information on inheritance relation between $T[\mathbf{s}_1]$ and $T[\mathbf{s}_2]$. For example, if bit $i_{\not\triangleright}$ is set in $\mathbf{I}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$, then no base of the root of $T[\mathbf{s}_1]$ lies in $\mathbf{s}_2$.

$\mathbf{R}_\times$ — a set of all restrictions from restriction set **R** that cannot be satisfied jointly with already processed restrictions.

The outline of the algorithm is as follows.

1) All sets $\mathbf{N}^*_{(*,*)}$, $\mathbf{P}^*_{(*,*)}$, $\mathbf{I}^*_{(*,*)}$ and $\mathbf{I}_{(*,*)}$ are set to empty.
2) A new restriction r from restriction set **R** is selected and placed into restriction processing queue. If it is a restriction of the form $D \triangleright B$, then it is first decomposed into $D \sim B$ and $D \not\triangleleft B$. If there are no unprocessed restrictions in **R**, then execution is continued from step 3.

[4]A Hasse diagram is a directed graph representation of a partially ordered set in which each element is represented by a node. Immediate successors of each element are connected to the corresponding node with a directed edge [10].

Algorithm *ProcessRestriction*
Input: Vtables B and D and a relation $\alpha \in \{\not\triangleright, \sim\}$ between them.
1. **set** bit i_α in $\mathbf{I}_{(B,D)}$
2. **let** $\mathbf{a}$ = leaf node for B
3. **while** $\mathbf{a} \neq \{D\}$ **do**
4. **let** $\mathbf{b} = \mathbf{D}^{\mathbf{a}}_{D}$ and $\mathbf{c} = \mathbf{D}^{\mathbf{b}}_{D}$
5. **if** $\mathbf{b}$ is an *any*-node and both $\mathbf{a}$ and $\mathbf{c}$ are children of $\mathbf{b}$ **then**
6. **if** $\alpha = \sim$ **then**
7. **add** D to $\mathbf{N}^{\mathbf{b}}_{(\mathbf{a},\mathbf{c})}$
8. **add** B to $\mathbf{N}^{\mathbf{b}}_{(\mathbf{c},\mathbf{a})}$
9. **set** bit $i_\sim$ in $\mathbf{I}^{\mathbf{b}}_{(\mathbf{c},\mathbf{a})}$
10. **else** ($*$ If we got here then $\alpha = \not\triangleright$ $*$)
11. **add** D to $\mathbf{P}^{\mathbf{b}}_{(\mathbf{a},\mathbf{c})}$
12. **elseif** $\mathbf{b}$ is a *fixed*-child of $\mathbf{a}$
13. **break**
14. $\mathbf{a} \leftarrow \mathbf{b}$

2.1) If restriction processing queue is empty, then execution continues from step 2.2. If restriction processing queue is not empty, then a single restriction is picked from it and processed using algorithm *ProcessRestriction*.

2.1.1) Algorithm *ProcessRestriction* modifies sets $\mathbf{I}_{(*,*)}$, $\mathbf{N}^{*}_{(*,*)}$, $\mathbf{P}^{*}_{(*,*)}$ and $\mathbf{I}^{*}_{(*,*)}$. For modified sets consistency maintenance rules are applied. These rules are described in the end of the chapter.

2.1.2) If modification results in a conflict in one of the sets $\mathbf{I}_{(*,*)}$ or $\mathbf{I}^{*}_{(*,*)}$, then r is added to $\mathbf{R}_\times$, all sets are returned to their state prior to processing of restriction r and execution is continued from step 2.

2.1.3) Consistency maintenance rules add new restrictions to the restriction processing queue. Execution continues from step 2.1.

2.2) It is checked whether for each *any*-node $\mathbf{p}$ there exists a tree with vertices from a set of its children satisfying $\mathbf{I}^{\mathbf{p}}_{(*,*)}$. If such tree does not exist, then r is added to $\mathbf{R}_\times$ and all sets are returned to their state prior to processing of r. Execution continues from step 2.

3) For each *any*-node $\mathbf{p}$ a rooted tree $T_{\mathbf{p}}$ with vertices from a set of its children satisfying all $\mathbf{I}^{\mathbf{p}}_{(*,*)}$ is constructed. Existence of such tree has been checked on step 2.2.

4) For each *any*-node $\mathbf{p}$ an induced inheritance tree $T[\mathbf{p}]$ is constructed. Nodes are processed in from-child-to-parent order, so that the tree for node $\mathbf{p}$ is constructed only when trees for all children of $\mathbf{p}$ have already been constructed. Trees $T[\mathbf{p}]$ and $T_{\mathbf{p}}$ can be seen as directed graphs with edges directed towards the root of the tree. For each directed edge $(\mathbf{s}_1, \mathbf{s}_2)$ of the tree $T_{\mathbf{p}}$ an edge connecting the root vtable R of $T[\mathbf{s}_1]$ and a vtable from $\mathbf{s}_2$ is added to $T[\mathbf{p}]$ so that all nodes from $\mathbf{N}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$ are reachable from R via directed edge walk, and all nodes from $\mathbf{P}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$ are not. Existence of such edge is ensured by consistency maintenance rules.

Consistency maintenance rules mentioned in the description of the algorithm propagate changes made to sets $\mathbf{N}^{*}_{(*,*)}$, $\mathbf{P}^{*}_{(*,*)}$ and $\mathbf{I}^{*}_{(*,*)}$ so that a single change in one of the sets may result in many consequent cascade changes in other sets. Some examples of consistency maintenance rules follow.

Consistency maintenance rule 1. *If for some node* $\mathbf{a}$ *and its children* $\mathbf{s}_1$ *and* $\mathbf{s}_2$*, set* $\mathbf{N}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)} \cap \mathbf{P}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$ *is not empty, then bit* $i_{\not\triangleright}$ *is set in* $\mathbf{I}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$.

In other words, the only way to satisfy both conditions $B \in \mathbf{N}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$ (vtable B must be inherited by the root vtable R of $T[\mathbf{s}_1]$ in case one of R's bases lies in $\mathbf{s}_2$) and $B \in \mathbf{N}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$ (vtable B must **not** be inherited by the root of $T[\mathbf{s}_1]$ in case one of its bases lies in $\mathbf{s}_2$) is for no base of the root of $T[\mathbf{s}_1]$ to lie in $\mathbf{s}_2$.

Consistency maintenance rule 2. *If for some node* $\mathbf{p}$ *and its children* $\mathbf{s}_1$ *and* $\mathbf{s}_2$*, condition* $\mathbf{s}_2 \subseteq \mathbf{P}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$ *is satisfied, then bit* $i_{\not\triangleright}$ *is set in* $\mathbf{I}^{\mathbf{p}}_{(\mathbf{s}_1,\mathbf{s}_2)}$.

In other words, the only way to satisfy a condition that all vtables from $\mathbf{s}_2$ must not be inherited by the root of $T[\mathbf{s}_1]$ in case one of its bases lies in $\mathbf{s}_2$ is for none of its bases to lie in $\mathbf{s}_2$.

The set of all vtables has been divided into several disjoint sets, so that all vtables in a single set form a connected inheritance hierarchy. The described algorithm is applied for each of these sets thus reconstructing the full vtable inheritance hierarchy.

D. 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 a sequence in which their addresses were written at the corresponding offset to the memory location being tracked.

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

Consider the program on fig. 4 for for two vtables B and D, D inheriting B. Associated set of pairs is $\{(0, (B, D)), (8, C)\}$. Since vtable D inherits from vtable B, this vtable access site is classified as a constructor.

In case each sequence in the associated set consists of only one element, this method cannot be used. Instead, the following rules are used.

1) Constructor and destructor calls, if not inlined, are nested: constructor calls constructors for base classes,

```
mov ecx, esi
call sub_40BBB0
mov dword ptr [esi], offset D::vtable
; ...
mov dword ptr [esi+8], offset C::vtable
; ...

sub_40BBB0 proc near
; this looks like a constructor of
; a base class
push esi
mov esi, ecx
mov dword ptr [esi], offset B::vtable
; ...
```

Fig. 4. Example program for classifying a vtable access site as either a constructor or a destructor.

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.

2) 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 being unidentified.
3) If all these rules fail to identify the vtable access site, then the case is reported to the user.

After completing vtable access site analysis, for each site classified as being either a constructor or unidentified, the corresponding set of (*offset*, *vtable sequence*) pairs is transformed. Last vtable is picked from each sequence thus producing a set of (*offset*, *vtable*) pairs. In accordance to the actions performed in constructor, 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. The class at hand may just include as a field some other class that owns this vtable. However, polymorphic classes are normally included by pointer. Besides, inclusion by value differs from inclusion by inheritance only if virtual destructors are involved (in this case a pointer to the same virtual destructor must be stored in all vtables of a class). 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 not have been generated by the compiler. That's why for each root vtable that contains pure virtual functions a class containing it at offset 0 is constructed.

After a set of classes is constructed, inheritance relation

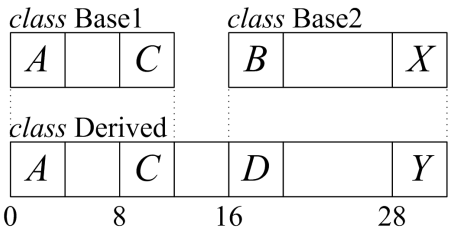

Fig. 5. Example of base class matching.

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. Example is given on fig. 5. Here vtable B is a direct base of D, and vtable X is a direct base of Y. Pair sets associated with classes are $\{(0, A), (8, C), (16, D), (28, Y)\}$ for class Derived, $\{(0, A), (8, C)\}$ for class Base1 and $\{(0, B), (12, X)\}$ for class Base2. Placement of both classes Base1 and Base2 inside class Derived satisfies the mentioned conditions, and therefore these classes can be direct bases of class Derived.

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.

Inheritance relation inference finishes class hierarchy reconstruction. As a result, for each class in a program the following is reconstructed.

1) Direct base classes with offsets to the corresponding instances.
2) Vtables and offsets to these vtables.
3) Constructors and destructors for non-"most-base" classes.

E. Complexity analysis

Algorithm for vtable search described in chapter III-C depends linearly on the size of program data segment. The complexity of data flow analysis algorithm used in chapters IV-A and IV-D depends linearly on the size of the program being analyzed.

The most computationally difficult part of the inheritance reconstruction process is the one described in chapter IV-C. Compared to it, computational complexity and memory requirements of multiple inheritance reconstruction algorithm described in chapter IV-D can be neglected.

We presume that the maximal size of vtable is a constant that can be neglected in the computational complexity estimations. Let n be a number of vtables found in the program being analyzed. The number of different functions in vtables is $O(n)$. Therefore, the size of the set **F** is also $O(n)$. The size

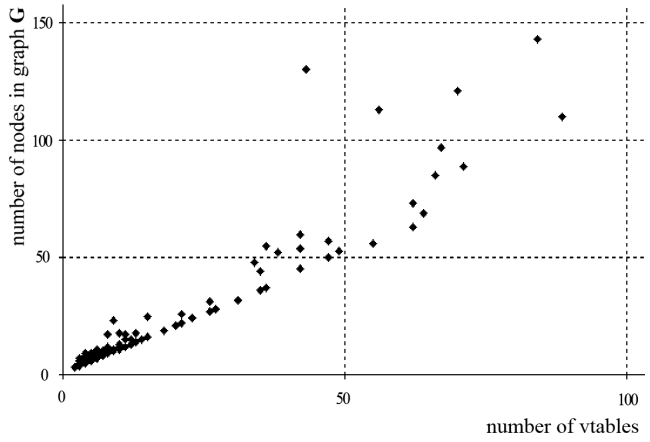

Fig. 6. Experimental results on the size of set $\mathbf{F}^*$ depending on the number of vtables.

of the set $\mathbf{F}^*$ depends exponentially on n, because it includes the intersection closure of $\mathbf{F}$. Let N be the size of the set $\mathbf{F}^*$.

It can be proven that the number of nodes in graph $\mathbf{G}$ is $O(N)$, the total size of all sets $\mathbf{N}^*_{(*,*)}$ and $\mathbf{P}^*_{(*,*)}$ and all fields $\mathbf{I}^*_{(*,*)}$ is $O(nN)$, and the total size of all fields $\mathbf{I}_{(*,*)}$ is $O(n^2)$. Therefore, the memory requirements of the presented algorithm is $O(nN)$.

Set $\mathbf{F}^*$ and graph $\mathbf{G}$ are constructed in $O(nN^2)$ operations. The total computational complexity of all other steps of the algorithm is $O(n^3N)$.

Exponential computational complexity may seem impractical. However, experiments have shown that this is not the case and in practice the size of the set $\mathbf{F}^*$ is not that big. For connected class hierarchies of size less than 100 classes statistics have been gathered by applying the algorithm to several open-source projects, including doxygen (source code documentation generator tool), Notepad++ (text editor), Shareaza (peer-to-peer file sharing client), mkvmerge (application for merging multimedia streams into a Matroska file), VirtualDub (video capture and processing utility), and others. Results are presented on fig. 6. As it can be seen, in most cases the size of the set $\mathbf{F}^*$ does not exceed $2n$. Therefore, the estimation of computational complexity of the algorithm is $O(n^4)$. More experimental data on running times of the algorithm is given in section V.

In most cases class hierarchies consisting of over 100 classes are hierarchies with a single "universal base class" like QObject in Qt library, CObject in MFC library and wxObject in wxWidgets library. Such hierarchies use RTTI in one form or another, and therefore the application of the algorithm presented in section IV is not needed for them. Besides, hierarchies of over 100 classes often become overly complex and hard to maintain, forcing programmers to use **dynamic_cast**<> operator to deal with unforeseen problems. The usage of this operator requires compilation with RTTI.

V. Implementation and experimental results

Presented methods have been implemented completely in C++ as a plugin for IDA Pro interactive disassembler. The plugin is available for download at web page http://www.decompilation.info. The plugin consists of several weakly bound modules, which can be reused in other contexts. These modules are as follows.

1) C++ wrapper for the functionality offered by IDA Pro.
2) Implementation of data flow analysis algorithm.
3) Implementation of class hierarchy reconstruction algorithm based on the analysis of RTTI structures.
4) Implementation of class hierarchy reconstruction algorithm described in chapters IV-C and IV-D.

The plugin has been tested on a variety of open-source software written in C++. The process used is as follows.

1) The program is compiled with optimizations and RTTI enabled.
2) RTTI-aware class hierarchy reconstruction algorithm is used to reconstruct the polymorphic class hierarchy exactly as it is in the source files.
3) The program is compiled with optimizations and debug information but without RTTI.
4) Class hierarchy reconstruction algorithm that does not use RTTI structures is applied.
5) Debug information is used to restore the correspondence between vtables and polymorphic classes.
6) Results of the two class hierarchy reconstruction algorithms are compared.

Application	doxygen	Shareaza	Notepad++
Running time	7.9s	18.9s	0.7s
Vtables found	444	1381	98
Non-vtables	6.5%	18.3%	3.0%
Vtable mismatches	8.6%	4.1%	4.0%
Classes found	401	1108	95
Class mismatches	9.7%	5.7%	4.2%

Fig. 7. Test results.

Plugin was tested on a Core2Duo CPU at 2.8Ghz. Some of the test results are presented on fig. 7.

- "Non-vtable" refers to a vtable that actually is a mere array of pointers. Debug information is used to check for this.
- "Vtable mismatch" refers to a vtable with reconstructed parent vtable differing from the real one. Note that this is not necessarily an error since in many cases small disturbances in hierarchy structure do not change the semantics of the program. For example, if all classes in a hierarchy have only two virtual functions, no data members, trivial constructors and destructors, and always override only the first function, then inheritance order does not matter because the code produced will be the same.
- "Classes found" refers to the number of polymorphic classes found during reconstruction.

- "Class mismatch" refers to a class with associated vtables or parents reconstructed differently from the real ones. Again, in many cases this is not an error due to the same reasons as with mismatched vtables.

In all tests non-vtables found during reconstruction have formed an inheritance tree consisting of a single node. Furthermore, the access pattern for these vtables was different. Normally a vtable is accessed by writing its address into memory. In case of an access to non-vtable, its address is often written into a register. Therefore, even though there were a lot of non-vtables in some test cases, these non-vtable were easily identified.

Study of vtable and class mismatch cases has shown that the results obtained do not contradict the information used in the analysis. 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). In most cases the mismatched vtable (class) just happened to be several levels up or down the hierarchy, preserving pairwise inheritance relations to the most of the other vtables (classes) in the hierarchy.

There were also a small number of cases when the tool has failed to reconstruct the inheritance relation between several classes, outputting them as unrelated. These cases are the ones pointed out in the beginning of chapter IV-B.

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 currently available for download. The plugin has been tested on a variety of open-source software written in C++ showing low mismatch rates.

Directions of future work include deeper analysis of constructors and destructors to further increase the accuracy of class hierarchy reconstruction, as well as 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] K. H. Rosen, *Discrete Mathematics and Its Applications, 5th edition*. McGraw-Hill Higher Education, 2002.

[11] M. Fowler, *Refactoring. Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

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

[13] 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.

[14] H. Sutter, "Sutter's mill: Virtuality," *C/C++ Users Journal*, vol. 19, no. 9, pp. 53–58, 2001.