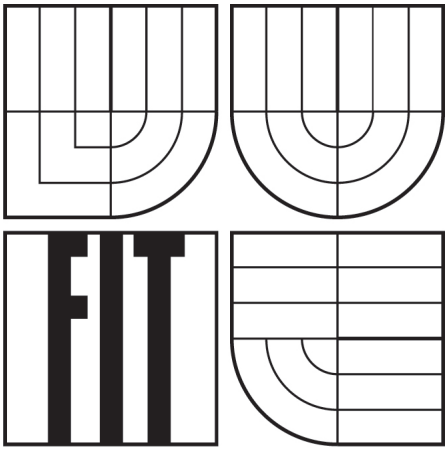

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH
TECHNOLOGIÍ

Decompilation of C++ Binaries

Programming Language Theory

Ing. Peter Matula
xmatul01[at]stud.fit.vutbr.cz
December 17, 2014

Contents

Chapter 1

Introduction

Decompilation is a reverse engineering technique performing a transformation of a platform-dependent binary file into a High Level Language (HLL) representation. Most existing decompilers are translating low-level machine code into the C language, since it is simple, yet powerful enough. Decompilers are not yet advanced enough to serve as a standalone tool, but combined with the traditional disassemblers, they allow much faster manual program analysis. Until recently, combination of a high-level C and low-level assembly representations was enough to reverse most applications. However, due to the increasing usage of C++ programming language in creation of more complex malware, understanding reversed programs have become much more difficult. For example, some of the world's most dangerous and widespread malicious programs like Agobot, variants of Mytob, Zeus trojan horse [6], Conficker.D worm [1], or Cryptolocker [2] were written in C++. To ease the analysis, new techniques reconstructing C++ features have been developed. This paper attempts to provide a review of most of the existing techniques and discover new areas that might be explored in the future.

This paper is organised as follows. Chapter 2 introduces the basic attributes and constructions of the C++ programming language. The underlying mechanisms supporting these constructions are explored in Chapter 3. The chapter is also describing how are existing decompilation techniques exploiting these mechanism in order of C++ decompilation. Chapter 4 is discussing possible research areas that were not yet explored and might become a subject of the author's further analysis. Chapter 5 concludes the paper by summarising the most important points and outlining the future work.

Chapter 2

C++ Language Overview

C++ is a general purpose programming language developed by Bjarne Stroustrup while working at Bell Labs. Even though it is mainly known as object-oriented language, it supports other paradigms as well (e.g. procedural, functional). It was designed to be efficient and flexible in order to use it for system programming. However, it spread to many other areas and became one of the most used programming languages today [4]. C++ is a compiled language available on nearly all platforms. The latest official standard [13] was ratified by the International Organization for Standardization (ISO) in fall of 2011 (it is therefore known as C++11).

This chapter introduces the basic constructions of the C++ programming language from the user's point of view. It is based on [13] and [29].

2.1 Basics

Since C++ is heavily based on the C programming language (however, it is not a strict superset of C) it inherits most of C's syntax and constructs. This includes data types, operators, control structures, functions and many other C language attributes. These basic constructions are further enhanced by principles described in the following sections of this chapter. The most elemental syntax of C++ is illustrated on the *hello world* program in Figure 2.1.

```
/* Hello world program */
# include <iostream>

int main()
{
   std::cout << "Hello, world!\n";
}
```

Figure 2.1: Hello world program in the C++ programming language.

2.2 Objects

C is an imperative programming language, where are data abstractions and operations performed on them declared separately, i.e. there is no language-supported relationship between data and functions. As was mentioned at the beginning of this chapter, C++ enhances C with an object-oriented programming (OOP) features. It introduces the concept of *classes*—data structures containing data and functions operating on them. Classes provide four features essential for OOP: *abstraction, encapsulation, inheritance*, and *polymorphism*.

Figure 2.2 shows a declaration of data type representing one point in the three-dimensional space. Furthermore, there are two functions working with this type. The first one creates `Point3D` object and the second one prints out its content. The same example is implemented as an abstract data type using C++ in Figure 2.3. This time, data (class members) are encapsulated in the class that provides methods (class member functions) for their access and manipulation. Different aspects of this example are further explored in the following sections.

```
typedef struct point3D
{
  double x;
  double y;
  double z;
} Point3D;

Point3D create_point3D(
    double x, double y, double z)
{
  Point3D ret;
  ret.x = x;
  ret.y = y;
  ret.z = z;
  return ret;
}

void print_point3D(Point3D *p)
{
  printf("%f %f %f",
         p->x, p->y, p->z);
}
```

Figure 2.2: Example of an abstract data type declaration and its manipulation functions in the C language.

```
class Point3D
{
public:
  Point3D(double x,double y,double z
     ): _x(x), _y(y), _z(z) {}
  ~Point3D() {}
  double x() {return _x}
  double y() {return _y}
  double z() {return _z}
  void x(double x) {_x = x;}
  void y(double y) {_y = z;}
  void z(double z) {_z = y;}
  void print_point3D()
  {
    printf("%f %f %f", _x, _y, _z);
  }

private:
  double _x, _y, _z;
  static int ID; ///< used by one of
      the later examples
};
```

Figure 2.3: Example of an abstract data type declaration and its manipulation methods in the C++ language.

2.2.1 Encapsulation

Information hiding (i.e. encapsulation) is a principle used to make sure data structures are used as was intended. It segregates implementation decisions from other parts of a program,

which can access the data only through a provided stable interface. In C++, it is possible to declare class members and methods as `public` (accessible anywhere), `protected` (accessible to methods of the original class, classes that inherit from it and specified friends) and `private` (accessible to member methods or friends).

All data members in the example in Figure 2.3 are private and therefore inaccessible outside the class. However, class provides sufficient interface for the basic manipulation. `Point3D(double x,double y,double z)` method is a so-called constructor—a subroutine called to create an instance of the class. On the contrary, `~Point3D()` is a destructor—an inverse function to constructor called when objects are destroyed. Method `double x()` is a so-called getter, which provides a read access to private the data member. Method `void x(double x)` is a setter writing a value of the data member.

2.2.2 Inheritance

Inheritance is a code reuse mechanism allowing to derive a class from one (single inheritance) or more (multiple inheritance) base classes, inheriting their properties and behaviour. Derived class may extend or override the original implementation while maintaining the same interface. Relationships between classes form a hierarchy. The access specifiers (`public`, `protected`, `private`) may be used to determine which base class members can be accessed by unrelated and derived classes. The inheritance may be also declared virtual, in which case only one instance of a base class exists in the hierarchy. This helps to solve potential problems caused by multiple inheritance.

Figure 2.4 demonstrates inheritance on three classes. Class `Point3D` is derived from `Point2D`, which is derived from `Point`. Each derivation extends the base class by adding the support for a new dimension, ultimately forming a class with the same behaviour as the one in Figure 2.3. However, this time an implementation is much more flexible, since each class can be used independently on the others.

2.2.3 Polymorphism

Polymorphism enables (1) single interface for entities of different types, and (2) different object behaviour under different circumstances. C++ supports both compile-time and run-time polymorphism.

Compile-time polymorphism

As the name suggests, this type of polymorphism is resolved by the compiler during the source code compilation. Therefore, there is no run-time performance overhead, but there is also no possibility of run-time decisions. The first technique of compile-time (static) polymorphism is function overloading. It allows to declare two or more same-named functions, providing they have different arguments. Compiler can then use these parameters to distinguish which function is actually being called. The second way to achieve static polymorphism is using generic programming techniques described in Section 2.3.

```
class Point
{
public:
  // declaration of: Point(), ~Point(), double x(), void x(double x)
protected:
  double _x;
}
class Point2D : public Point
{
public:
  // declaration of: Point2D(), ~Point2D(), double y(), void y(double y)
protected:
  double _y;
}
class Point3D : public Point2D
{
public:
  // declaration of: Point3D(), ~Point3D(), double z(), void z(double z)
protected:
  double _x;
}
```

Figure 2.4: Example of inheritance in the C++ language.

Run-time polymorphism

In C++, run-time (dynamic) polymorphism is achieved through class inheritance. The key feature is that pointers and references to a base class can refer to objects of derived classes. Since variable assignment occurs at run-time, it is not possible to resolve type of pointed-to object during the compilation. Polymorphism is supported in the following ways (examples are taken from Figure 2.5):

1. Through a set of implicit conversions such as: `Polygon * ppoly1 = new Rectangle;`. This allows to store pointer to derived object into a pointer variable of the base class type.

2. Through virtual function mechanism (e.g. `ppoly1->area()`). Function to call is determined by the type of the object. Therefore, only the `Polygon` members can be accessed using the base type pointers `ppoly1, ppoly2`. However, virtual member functions such as `Polygon::area()` can be redefined in derived classes while preserving its calling properties through references. This means that once classes `Rectangle` and `Triangle` define their own `area()` methods, it is no longer possible for a compiler to determine which method implementation is actually going to be called. The decision is therefore delayed until run-time, when process called dynamic dispatch determines which polymorphic method to call.

3. Through the **`dynamic_cast`** and **`typeid`** operators (e.g. `Rectangle *r = dynamic_cast <Rectangle*>(ppoly1)`). The first one allows to safely convert object of general type into more specific type. The later one allows to query type's information.

```
class Polygon
{
public:
  void set_values(int w, int h) { width = w; height = h; }
  virtual ~Polygon() {}
  virtual int area() { return 0; }
protected:
  int width, height;
}
class Rectangle : public Polygon
{
public:
  int area() { return width * height; }
};

class Triangle : public Polygon
{
public:
  int area() { _area = width * height / 2; return _area; }
private:
  int _area;
};
int main()
{
  Polygon * ppoly1 = new Rectangle;
  Polygon * ppoly2 = new Triangle;
  ppoly1->set_values (4,5);           // Polygon::set_values()
  ppoly2->set_values (4,5);           // Polygon::set_values()
  cout << ppoly1->area() << '\n';     // Rectangle::area()
  cout << ppoly2->area() << '\n';     // Triangle::area()
  if (Rectangle *r = dynamic_cast<Rectangle*>(ppoly1))
    cout << r->area() << '\n';        // Rectangle::area()
}
```

Figure 2.5: Example of polymorphism in the C++ language.

2.3 Templates

C++ templates are the foundation of generic programming and template metaprogramming. They allow to write the code that is independent of any particular data type. A template is a generic blueprint used to create concrete function or class implementations for all of the necessary data types. This process called instantiation is performed by a compiler at compile-time.

Figure 2.6 shows an example of simple function template in C++. Template function `max()` is parameterised by one type `T`. In the example, function is used on three places with three different data types (`int`, `double`, `std::string`). Since all of these types support operations performed by the function, it is possible for a compiler to instantiate three different variants of the generic function.

```
template <typename T>
T const& max(const T &a, const T &b)
{
  return (a < b) ? (b) : (a);
}
void use_template(void)
{
  cout << max(42, 743) << endl;           // T = int
  cout << max(3.1415, 2.72) << endl;      // T = double
  cout << max("hello", "world") << endl; // T = std::string
}
```

Figure 2.6: Example of function template in the C++ language.

2.4 Exception handling

Exception handling (EH) is a way to transfer control to handler function in case of detection of some run-time problem or error. When an error occurs, an exception is thrown and the current and all parent scopes are exited until the exception is caught by the nearest suitable handler. Information about the problem is carried along as an object.

Exception handling example is shown in Figure 2.7. The `try` block contains code that might raise an exception. The `catch` block serves as the exception handler routine. There might be multiple handlers for a single `try`.

```
try {
  std::vector<int> vec{1,2,3,4,5};
  int i{vec.at(10)};              // throws std::out_of_range exeption
}
catch (std::out_of_range& e) {  // handler catching std::out_of_range
  std::cerr << "Index out of range: " << e.what() << '\n';
}
catch (...) {                     // catching any other exception
  std::cerr << "Some other exception\n";
}
```

Figure 2.7: Example of exception handling in the C++ language.

2.5 Standard library

In addition to the C++ core language, the standard also specifies the C++ Standard Library. It includes many crucial features making the life of C++ programmers much easier. Some of the most important components are: containers (e.g. vectors, lists, maps), algorithms (e.g. find, for_each), input/output streams, regular expressions, smart pointers, atomic operation support and many other.

Chapter 3

C++ Language Decompilation Techniques

This chapter explains, and exploits for decompilation purposes, the underlying mechanisms behind C++ features as they are implemented in the three major compilers (i.e. GCC, Clang and MSVC). All examples are illustrated on 32-bit binaries (i.e. pointers in demonstrations have 4 bytes). The chapter is based on several articles [12, 11, 27, 28, 26] as well as on the C++ language standard [13] and an excellent Stanley Lippman's book [22].

3.1 Class memory layout

The most basic thing to understand when trying to decompile C++ binaries is class instance (i.e. object) memory layout. All major compilers are using the following object model:

- Ordinary data members are stored within each object. For example, members `x`, `y`, `z` of class `Point3D` declared in Figure 2.3 are stored sequentially (in the declaration order) as is shown in Figure 3.1.
- One instance of each static data member is allocated in the global data section outside individual objects. For example, `Point3D`'s member `ID` is shared between all class instances and compiler most likely places it into the `.data` section.
- Non-virtual methods (static and non-static) are also outside individual objects. These functions are once again shared by all instances and are placed in the code (`.text`) section. Compiler augments each of them by one parameter holding the reference to a particular object they operate on.
- In case a class inherits from one or more base classes, the derived class layout is appended to concatenated layouts of all base classes. For example, Figure 3.2 depicts a memory layout of the `Point3D` class, which was created by concatenating all inherited and newly declared `Point3D`'s members.
- Virtual functions are supported through virtual function table (i.e. vtable), explained in detail in Section 3.3. Pointer to this table is placed into each instance (typically

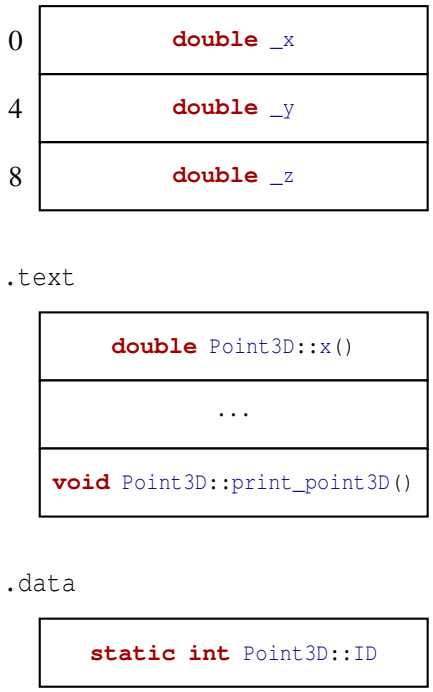

Figure 3.1: Memory layout of class `Point3D` and its methods from Figure 2.3.

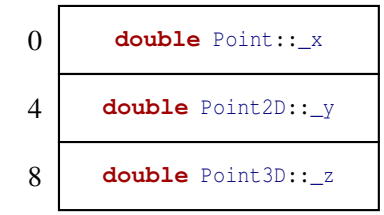

Figure 3.2: Memory layout of class `Point3D` from Figure 2.4.

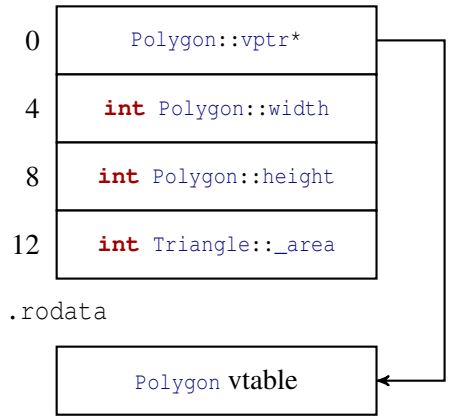

Figure 3.3: Memory layout of class `Triangle` from Figure 2.5.

at the beginning) whose class contains at least one virtual method. If a class inherits from one or more virtual bases, each embedded base object has its own virtual table pointer. In such a case, the current object's virtual functions are appended to the first base's virtual method list. For example, `Triangle`'s layout in Figure 3.3 contains reference to `Polygon`'s virtual table (inheritance), but it does not have its own vtable.

Memory layout of any class is in fact just a C style structure of its ordinary and inherited data members, plus the optional[1] virtual table pointer. Therefore, it is possible to reconstruct them using the simple [23] and composite [24] data type recovery algorithms designed for language C decompilation. There is however a problem in class pointer type propagation discussed in Section 3.6.

3.2 Run-Time Type Information

The easiest and the most rewarding approach to polymorphic class hierarchy reconstruction is Run-Time Type Information (RTTI) utilisation. RTTI is a mechanism for a run-time object type querying, which stands behind **`dynamic_cast`** and **`typeid`** operators. An RTTI structure with information about parents of each polymorphic class (i.e. class containing at least one virtual method) is created by the compiler. All that needs to be done to obtain this

[1] In the sense that class may not contain virtual methods. If it does, virtual table reference is mandatory.

knowledge is to find and parse all RTTI records. Full RTTI examination reveals a complete polymorphic class hierarchy, even with the original class names.

Fortunately, all three major compilers' ABIs stores a reference to class's RTTI as the first entry of corresponding virtual function table. Therefore, finding an RTTI is reduced to a problem of identifying virtual tables solved in Section 3.3. An actual layout of RTTI structures is also specified by the ABI, and it can be divided into two flavours: (1) GCC and Clang, (2) MSVC. The remainder of this section describes both variants. See [25] for more implementation details.

3.2.1 GCC and Clang

Run-Time Type Information used by these compilers is shown in the form of an UML diagram in Figure 3.4. It is specified by Itanium C++ ABI [3] developed jointly by an industry coalition. An actual RTTI object referenced by the virtual function table depends on the inheritance type of class it represents:

- No inheritance – `__class_type_info` structure is used. It contains no information except a reference to class name inherited from the **`type_info`** base;
- Single inheritance – `__si_class_type_info` contains reference to the parent's RTTI;
- Multiple inheritance – `__vmi_class_type_info` contains array of structures describing all base classes, as well as inheritance type flags.

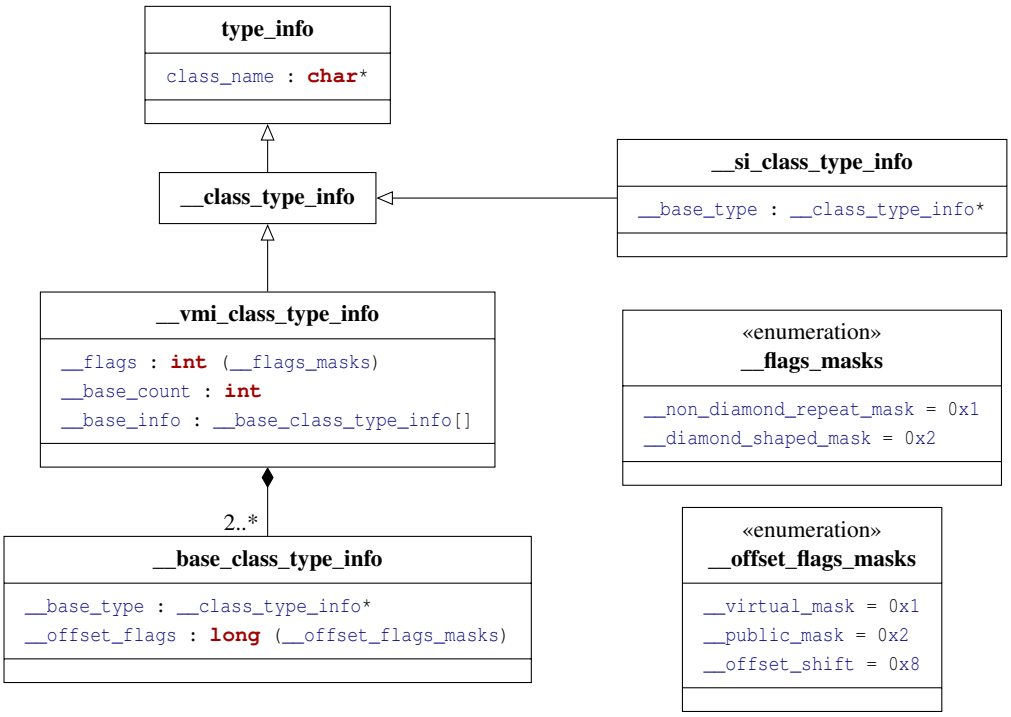

Figure 3.4: RTTI structure used by GCC and Clang compilers.

3.2.2 MSVC

Run-Time Type Information used by MSVC compiler is shown in the form of an UML diagram in Figure 3.5. It consists of these entities:

- `Complete_object_locator` – is the main structure referenced by virtual function table. It contains references to class type descriptor and class hierarchy. It is possible, that one class have several of these structures;
- `RTTI_type_descriptor` – describes data type, including its name. Name is always mangled with prefix `.?AV`;
- `RTTI_class_hierarchy` – contains array of base class references. The first element is class's own base class descriptor.
- `RTTI_base_class_descriptor` – describes one parent.

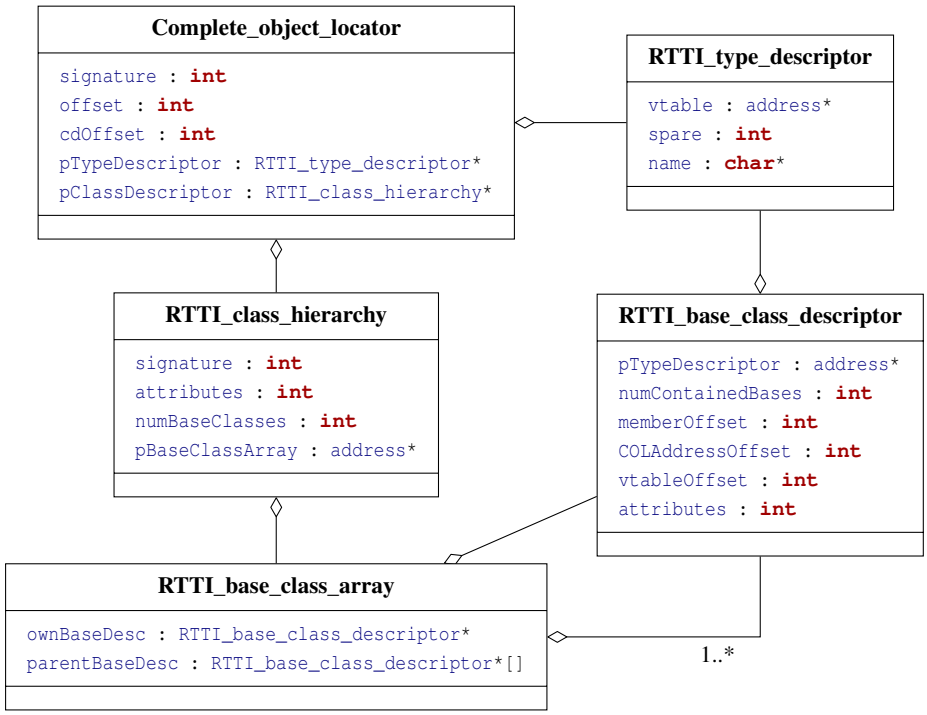

Figure 3.5: RTTI structure in MSVC compiler.

3.3 Virtual functions

Even though C++ standard does not impose any specific implementation of dynamic dispatch, all major compilers support virtual methods, and therefore polymorphism, through virtual function tables (i.e. vtables). Each table starts with a reference to RTTI structure followed by an array of pointers to virtual functions. Therefore, the problem of locating RTTI information and identifying virtual functions is reduced to finding all virtual tables.

As was already outlined in Section 3.1, if class B inherits from class A, then B's virtual table is created (by compiler) from A's table in two steps: (1) addresses of A's virtual functions that are overridden in B are replaced with new addresses, and (2) addresses of new virtual functions are appended to the end of the table. In case of a virtual inheritance, location of parent's vtable is not fixed and table offsets of all ancestors must be checked. GCC and Clang solve this by adding a virtual table offset before the table itself. MSVC generates a so-called virtual base table (i.e. vbtable) containing parents' vtable offsets.

Virtual function tables resides in binarie's data section (typically in read only data, i.e. `.rodata`), and can be found by analysing each word in the section using these two rules:

- If the current word is a reference to a function (i.e. address in code segment) and it itself is referenced from the code segment, then it is consider a virtual table start.
- Consequent vtable entries must be unreferenced function addresses. Virtual table ends on the first word that is not a function address or is referenced from code.

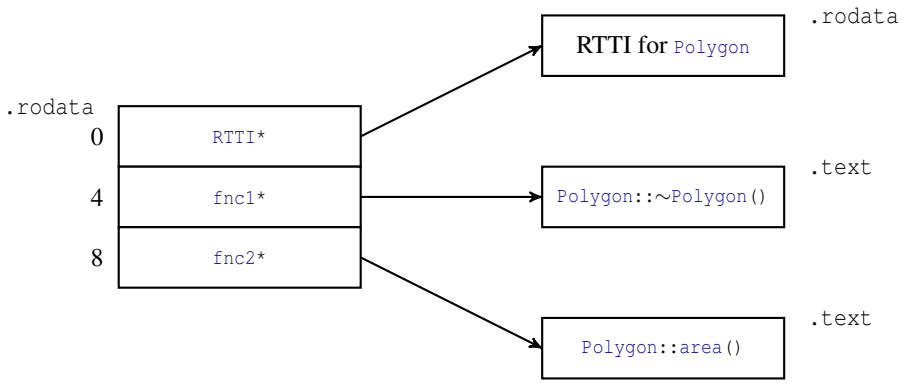

Figure 3.6: Virtual function table for the `Polygon` class from Figure 2.5.

3.4 Virtual function inheritance hierarchy

Run-Time Type Information introduced in Section 3.2 is frequently misused, and its generation is therefore often disabled. Some more complex C++ projects (e.g. Qt, LLVM) are also refraining from using it, since it introduces unnecessary overhead or is not strong enough. Therefore, in some cases the RTTI base approach to class hierarchy recovery is useless. Fortunately, virtual tables are always generated and can be used to infer hierarchy instead of RTTI. Unfortunately, the uniqueness of vtables is not guaranteed (i.e. a single vtable might be shared between multiple classes), what complicates the whole process. The following is a brief description of technique presented in [11].

Hierarchy inference runs after all vtables were identified using the algorithm from previous section. The method presumes that each vtable inherits from at most one parent virtual table, since it can not handle virtual inheritance. For two vtables A and B, A is a direct base of B if one of the classes corresponding to A is a direct base of one of the classes corresponding to B. A and B relationship can then be one of the following:

- Vtable B inherits from A is denoted as $B \triangleright A$;
- Vtable A does not inherit from B is denoted as $A \not\triangleright B$;
- Vtable A inherits from B, or B inherits from A is denoted as $A \sim B$.

The following rules are used to reconstruct inheritance relation on a set of virtual tables:

1. If the size of vtable A is less then the size of vtable B, then A cannot inherit from B: $(A \not\triangleright B)$.
2. If virtual function A_i (i-th virtual function in vtable A) is pure and virtual function B_i is not, then vtable A cannot inherit from vtable B: $(A \not\triangleright B)$.
3. If sizes of parameters of the virtual functions A_i and B_i are different, then neither vtable A inherits from vtable B, nor B inherits from A: $(A \not\triangleright B \vee B \not\triangleright A)$.

All of these rules are dealing with cases, when one virtual table can not inherit from some other vtable. In order to infer positive relationships (i.e. A does inherit from B), the additional information provided by constructor and destructor analysis is used.

3.5 Constructor and destructor identification

Current state-of-the-art decompilers detect constructors and destructors only for polymorphic classes, since non-polymorphic ones do not differ from ordinary functions. They are detected based on the operations they need to perform [13]. Class constructors have to execute the following sequence of steps (taken from [11]) in order to initialise their objects:

1. call constructors of direct base classes;
2. call constructors of data members;
3. initialise vtable pointer field(s) and perform user-specified initialisation code in the body of the constructor.

A destructors performs operations in reverse order (taken from [11]):

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

Constructors and destructors can be therefore located by detecting consequent virtual table pointer fields initialisations. Moreover, the initialisation order can be used to distinguish constructors from destructors, as well as infer the class hierarchy. For example, constructor and destructor executions for class `C` from hierarchy in Figure 3.7 are shown in Figures 3.9 and 3.10. In constructor, vtable pointer entries are overwritten after the call to parent constructor in a base-to-derived order. On the contrary, destructor first overwrites vtable pointer members in a derived-to-base order and than calls the parent destructor. Based on this information, it is not only possible to decide which functions are constructors and which destructors, but we can also find out that class `C` inherits from class `B` and `B` from `A`.

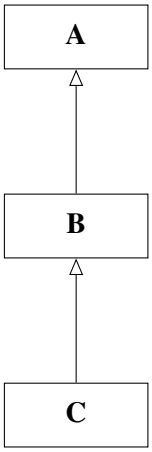

Figure 3.7: Class hierarchy example.

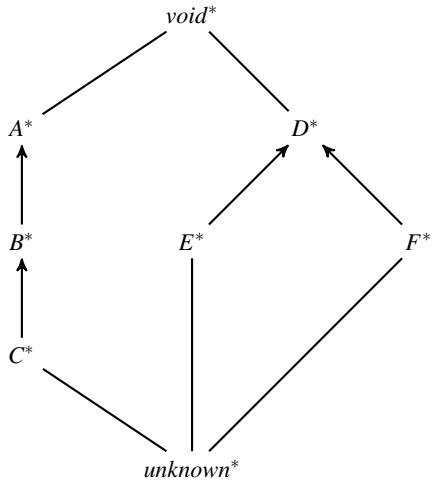

Figure 3.8: Type lattice example. UML notation is used to denote class inheritance.

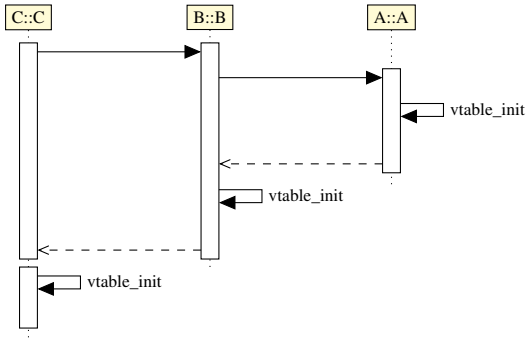

Figure 3.9: Constructor execution for class `C` from Figure 3.7. Initialisation of vtable pointer members is denoted by `vtable_init` callback.

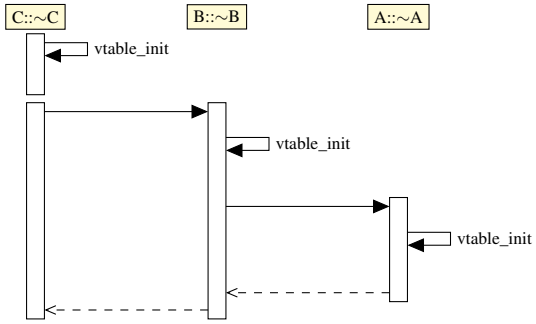

Figure 3.10: Destructor execution for class `C` from Figure 3.7. Initialisation of vtable pointer members is denoted by `vtable_init` callback.

3.6 Class pointers

As was already mentioned in Section 3.1, there is a problem with class pointer propagation. All algorithms for language C type reconstruction [23, 24, 9, 31] rely on assumption that copies of the same pointer always reference the same type. This is clearly broken by polymorphic class pointers, since they can point to objects of many different classes. A final pointer type must be compatible with all the types that can be assigned to it. That means, it is either a common superclass or generic `void*` pointer. A lattice model for pointers is used to decide. At first, class hierarchy is reconstructed, then a type lattice $(\mathbb{C}, \triangleright)$ is build. The set of pointer types $\mathbb{C}$ consists of:

- the pointers to all recovered class types;
- `void*` pointer as top;
- special `unknown` pointer as bottom.

The partial order $\triangleright$ is defined by the inheritance relation on a set of classes with addition of the following elements:

- For all $A \in \mathbb{C}$ that are the roots of inheritance hierarchy, $A \triangleright$ `void*`.
- For all $A \in \mathbb{C}$ that are the leafs of inheritance hierarchy, `unknown` $\triangleright A$.

Hasse diagram for lattice created for example in Figure 3.7 is shown in Figure 3.8. At first, all pointer types are `unknown`. Then, each assignment sets its type to the least common ancestor of its current type. At the end, each pointer is of the superclass or `void*` type.

3.7 Ordinary member functions

Apart from the virtual function identification, it would be desirable to find out which functions are ordinary member functions and to what class they belong. However, this task can not be achieved in GCC and Clang, where *this* pointer is passed as a hidden first parameter. In such a case, methods become indistinquishable from traditional C style functions unassociated with any class. However, MSVC compiler is using *__thiscall* calling convention, where *this* pointer is always passed in `ECX` register. Therefore, associated class can be reliably infered from the type of register object.

3.8 Exception handling

C++ standard defines only the exception handling semantics described in Section 2.4. The implementation is left to compiler developers. Therefore, MSVC is using a different approach than GCC and Clang. In any case, EH implementation involves fairly low-level manipulation, which is not easy to express in the C language terms. Proper exception handling reconstruction demands modifications of several decompilation analyses, otherwise the quality of produced code is very poor. The following basic EH implementation principles can be used to recover `throw` statements and `try` and `catch` blocks:

- In order to reflect the current program's state, MSVC compiler generates code, which is repeatedly updating the exception handling structures. Each function gets its own stack frame element containing information on exception handlers accessible from the associated function. This data is used by the run-time support library to execute the correct EH.
- On the other hand, GCC and Clang compilers use a table-driven approach with no run-time overhead when exceptions are not actually used. Tables map program counter ranges to their program state. Therefore, they can be used by a run-time look up to identify appropriate handler based on the execution's position at the time of an exception throw.

An exact reconstruction procedure is fairly complex and will not be covered in detail in this paper. See article [11] for more details.

3.9 Standard library function recognition

With the C++ standard library comes a lot of new standard functions that can be used by programmers in their applications. Recognising calls of such functions and exploiting their known declarations can greatly improve decompiled code's quality. Standard library functions can be linked to programs in two different ways:

- Dynamically linked functions are not actually appended to user programs. They are located in shared libraries, that are loaded alongside user applications. All function calls are indirect through a so-called stubs—jumps whose target addresses are fixed to actual standard functions' locations by the dynamic linker during loading of program into memory. In order to perform such a loading, linker must known which functions from which libraries is application going to need. Therefore, binary must contain all of the necessary information to identify required functions. This information can be exploited by the decompiler to replace jumps with the original function calls. Moreover, if the decompiler has a database of known function prototypes, which contains information about functions' parameters and their data types, it can reconstruct calls precisely and propagate type information to other objects.

- Statically linked functions are appended to user programs and do not require any additional run-time library interaction. If the application is stripped (i.e. symbol names are removed), they become indistinguishable from user defined functions and greatly complicates any further analysis. To identify such functions, signature based solutions like IDA Pro FLIRT technology or generic detection of statically linked code described in [34, 33] is used. Combined with known function prototypes, it is possible to identify most of such functions and achieve the same high quality output as in the previous case.

Chapter 4

Possible research areas

This chapter is discussing possible research directions that could be explored in the future. Even though some of the ideas have already been proposed, or even implemented to some extent, there currently is not a usable retargetable decompiler that would integrate them into a single reverse engineering framework.

4.1 Debugging information utilisation

Debugging information is generated by compilers in order to enable debugging process of finding and fixing software bugs. Since it contains many source-code-level data mapped to machine code, it may be exploited in the process of executable file analysis. The typical use in reverse engineering is to evaluate accuracy of the analysis by comparing its result against debugging information. This approach was used in [30], where the `readelf` utility was used to extract debugging data, or in [21] and [10] by utilising the `libdwarf` library. It is however not clear whether any of these tools is capable to incorporate such information to its algorithm and produce more accurate output because of it, or if they are limited to accuracy evaluation only. Since most of the executable analysis applications are dependent on a particular architecture and platform it is also unlikely that any of them is able to process different debugging formats. Moreover, all of these works are focused on the C language analysis, and are probably unable to make use of any C++ related information.

For this reasons, it would be the best to continue in the direction presented in [20] and [19]. These papers describe two new mid-layer libraries supporting manipulation of the two major debugging information formats: Unix DWARF and Microsoft PDB. As is the case in the previously mentioned works, both libraries are currently focused only on the C language related constructions. However, it would not be complicated to expand their functionality by parsing of C++ object-oriented features. The goal is not only to use this information for analysis evaluation, but to fully incorporate it into the decompiler's analyses in order to achieve very high quality output.

4.2 Name demangling

Name mangling is a process of encoding additional information in the various source-level names in order to solve problems caused by the need for unique entity names. Fortunately for reverse engineering, the additional semantic information can be used to obtain knowledge about functions, structures, classes, namespaces etc. Unfortunately, the C++ standard does not define a common decoration scheme and each compiler vendor is free to use its own. This greatly complicates utilisation of encoded information, but at the same time offers an opportunity to solve the problem once and for all by creating a retargetable demangler. Although some decompilers like Hex-Rays [16] can demangle mangling schemes used by the most popular C++ compilers, it does so only to display the original name. Actual modification of output code (i.e. recovering function arguments and their types) is not performed.

A simple example of possible mangling is shown in Figure 4.1. Two different functions with the same name are distinguished by the compiler based on their parameters. To make linking possible, number and types of their arguments are encoded into their names. If these names make it into the binary (i.e. they are not stripped), it is possible for a decompiler to precisely recover this information, provided it is familiar with the used mangling scheme.

```
int fnc (void);   // mangled name = __f_v
int fnc (int);    // mangled name = __f_i
```

Figure 4.1: Example of a simple function name mangling in the C++ language.

4.3 Method classification

As far as the method semantics identification goes, all known C++ decompilation techniques are focused on constructor and destructor recognition in order of class hierarchy reconstruction. This reconstruction is based on the sequence of steps that must be carried out by these special methods (see Section 3.5). However, constructors and destructors are not the only functions which behaviour is prescribed by the language standard. Similar demands on responsibilities applies to some other functions such as: copy constructors or class allocation and de-allocation routines created from `new` and `delete` operators. Further analysis and classification of methods or code snippets could prove useful for the analyst trying to comprehend decompiled source codes.

4.4 Template recognition

As was explained in Section 2.3, compiler creates as many variations of template functions or classes as many different types are used to instantiate the template. Despite the fact all of them are generated from the common blueprint, they might be vastly different. For example, template function in Figure 2.6 has two parameters and a return value of the same type, and is using a ternary operator. This basic structure will be the same for all instances.

However, operator < (less than) is probably going to be implemented differently for each data type. So, the question that might be answered in the future is: "Is it somehow possible to identify methods generated by compiler from the common template?"

4.5 C++11 decompilation

The current C++ standard from 2011 [13] and the upcoming 2014 revision [14] significantly changed C++. From the reverse engineering point of view, introduced features can be divided into two groups.

1. Features providing syntactic sugar (e.g. `auto`, initializer lists) or other constructions removed by compiler and indistinguishable on the machine code level.

2. Features that might be recognised in machine code and recovered back to the high level representation.

Finding out, which new features are in the former category and how can they be reconstructed will definitely be a subject to the further research.

4.6 Dynamic analysis

All of the techniques introduced so far are performed without actually executing analysed programs. Such methods are called static and they have several advantages over the dynamic program analysis, which needs to execute inspected programs. Execution introduces a whole range of problems such as security concerns, code coverage, sufficient test inputs or analysis speed. One of the biggest problems is also the need for an adequate testing environment and instrumentation tool. For example, it would be impossible to run MIPS executable on x86 workstation without some kind of virtual processor. Moreover, just running the program would be pointless without the means to instrument its instructions. Despite these obvious drawbacks, dynamic analysis is often capable to achieve much better results than the static one. Quality can be further enhanced by combining the results of both approaches. The examples of dynamic analysis in C/C++ binary reverse engineering can be found in [18, 7, 15, 30].

However, all of these efforts are either tied to a particular architecture (e.g. by using Intel PIN instrumentation tool [17]) or are not fully integrated into a full-scale decompiler. Therefore, a primarily static retargetable reverse compiler capable of improving its results with information obtained by program execution is yet to be developed.

Chapter 5

Conclusion

This paper is addressing the issue of C++ decompilation. It reviews most of the existing techniques as well as outlines new promising areas for the future research. The ultimate goal is to create all-in-one reverse engineering framework, capable of all state-of-the-art analyses plus as many of the proposed techniques as possible. It should be also truly retargetable, i.e. independent on any particular target architecture, compiler, object file format or operating system. An ideal foundation for such C++ reversing tool is Retargetable Decompiler [5] developed in cooperation of Faculty of Information Technology from Brno University of Technology, and AVG technologies. The decompiler can already translate x86, ARM+Thumb, MIPS, PowerPC and Pic32 binaries in PE and ELF file formats into C language high level representation. Author's future work will be to expand its capabilities by adding the C++ decompilation support.

Such a tool would found use in the field of malware analysis. It would significantly ease up and speed up the process of manual program inspection, especially in case of more complicated malicious programs written in C++. Example of a possible use in order of static detection of C++ virtual table escape vulnerabilities can be found in [8]. Other benefits of such a tool are described in [32].

Bibliography

[1] Conficker worm. http://en.wikipedia.org/wiki/Conficker. Accessed: 2014-12-11.

[2] Cryptolocker trojan horse. http://en.wikipedia.org/wiki/CryptoLocker. Accessed: 2014-12-11.

[3] Itanium C++ ABI.

[4] Programming language popularity. http://langpop.com/. Accessed: 2014-11-2.

[5] Retargetable decompiler. http://decompiler.fit.vutbr.cz/. Accessed: 2014-12-11.

[6] Zeus trojan horse. http://en.wikipedia.org/wiki/Zeus_(Trojan_horse). Accessed: 2014-12-11.

[7] G. Chen. A novel lightweight virtual machine based decompiler to generate C/C++ code with high readability.

[8] David Dewey and Jonathon Giffin. Static detection of C++ vtable escape vulnerabilities in binary code.

[9] E. N. Dolgova and A. V. Chernov. Automatic reconstruction of data types in the decompilation problem. *Program. Comput. Softw.*, 35(2):105–119, March 2009.

[10] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. *SIGPLAN Not.*, 48(6):51–60, June 2013.

[11] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: Approaching C++ decompilation. In *WCRE*, pages 347–356. IEEE Computer Society, 2011.

[12] Alexander Fokin, Katerina Troshina, and Alexander Chernov. Reconstruction of class hierarchies for decompilation of C++ programs. In *CSMR*, pages 240–243. IEEE, 2010.

[13] International Organization for Standardization. ISO International Standard ISO/IEC 14882:2011 – Programming Language C++, 2011.

[14] Standard C++ Foundation. Working Draft, Standard for Programming Language C++, 2014.

[15] István Haller, Asia Slowinska, and Herbert Bos. Mempick: High-level data structure detection in c/c++ binaries. In Ralf Lämmel, Rocco Oliveto, and Romain Robbes, editors, *WCRE*, pages 32–41. IEEE, 2013.

[16] Hex-Rays Decompiler. www.hex-rays.com/products/decompiler/, 2013.

[17] Intel Corporation. Pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. Accessed: 2014-11-2.

[18] Jaromír Končický. Využití dynamické analýzy kódu při zpětném překladu. Master's thesis, Faculty of Information Technology, BUT, 2014.

[19] Jakub Křoustek, Peter Matula, Dušan Kolář, and Milan Zavoral. Advanced preprocessing of binary executable files and its usage in retargetable decompilation. *International Journal on Advances in Software*, 7(1):112–122, 2014.

[20] Jakub Křoustek, Peter Matula, Jaromír Končický, and Dušan Kolář. Accurate retargetable decompilation using additional debugging information. In *Proceedings of the Sixth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE'12)*, pages 79–84. International Academy, Research, and Industry Association, 2012.

[21] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *NDSS*. The Internet Society, 2011.

[22] Stanley B. Lippman. *Inside the C++ Object Model*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[23] Peter Matula and Dušan Kolář. Reconstruction of simple data types in decompilation. In *Sborník příspěvků Mezinárodní Masarykovy konference pro doktorandy a mladé vědecké pracovníky 2013*, pages 1–10. Akademické sdružení MAGNANIMITAS Assn., 2013.

[24] Peter Matula and Dušan Kolář. Composite data type recovery in a retargetable decompilation. In *Proceedings of the 9th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 63–76. NOVPRESS s.r.o., 2014.

[25] Tomáš Mihulka. Zpětný překlad vybraných konstrukcí jazyka c++. B.S. Thesis, Faculty of Information Technology, BUT, 2014.

[26] Paul Vincent Sabanal and Mark Vincent Yason. Reversing C++.

[27] Igor Skochinsky. Reversing microsoft visual C++ part I.

[28] Igor Skochinsky. Reversing microsoft visual C++ part II.

[29] Bjarne Stroustrup. *Programming Principles and Practice Using C++*. Addison Wesley Longman Publishing Co., Inc., 2009.

[30] E. N. Troshina and A. V. Chernov. Using information obtained in the course of program execution for improving the quality of data type reconstruction in decompilation. *Programming and Computer Software*, pages 343–362, 2010.

[31] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. Reconstruction of composite types for decompilation. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM '10, pages 179–188, Washington, DC, USA, 2010. IEEE Computer Society.

[32] Mike Van Emmerik and Trent Waddington. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 27–36, Washington, DC, USA, 2004. IEEE Computer Society.

[33] Lukáš Ďurfina and Dušan Kolář. Generic detection of the statically linked code. In *Proceedings of the Twelfth International Conference on Informatics INFORMATICS 2013*, pages 157–161. Faculty of Electrical Engineering and Informatics, University of Technology Košice, 2013.

[34] Lukáš Ďurfina and Dušan Kolář. Generic detection and annotations of the statically linked code. *Acta Electrotechnica et Informatica*, 2013(4):51–56, 2014.