2011 18th Working Conference on Reverse Engineering

SmartDec: Approaching C++ Decompilation

Alexander Fokin*, Egor Derevenetc†, Alexander Chernov‡ and Katerina Troshina§
*‡*Computational Math. and Cybernetics Dept., Moscow State University, Moscow, Russia*
Email: {apfokin, blackav}@gmail.com
†*High Performance Computing Department, Fraunhofer ITWM, Kaiserslautern, Germany*
Email: egor.derevenetc@itwm.fhg.de
§*Select LTD, Moscow, Russia*
Email: katerina.dolgova@gmail.com

***Abstract*—Decompilation is a reconstruction of a program in a high-level language from a program in a low-level language. Typical applications of decompilation are software security assessment, malware analysis, error correction and reverse engineering for interoperability.**

Native code decompilation is traditionally considered in the context of the C programming language. C++ presents new challenges for decompilation, since the rules of translation from C++ to assembly language are far more complex than those of C. In addition, when decompiling a program that was originally written in C++, reconstruction of C++ specific constructs is desired.

In this paper we discuss new methods that allow partial recovery of C++ specific language constructs from a low-level code provided that this code was obtained from a C++ compiler. The challenges that arise when decompiling such code are described. These challenges include reconstruction of polymorphic classes, class hierarchies, member functions and exception handling constructs. An approach to decompilation that is used to overcome these challenges is presented.

SmartDec, a native code to C++ decompiler that is being developed by the authors at Select LTD is presented. It reconstructs expressions, function arguments, local and global variables, integral and composite types, loops and compound conditional statements, C++ class hierarchies and exception handling constructs. An empirical study of the decompiler is provided.

***Keywords*-C++, Decompilation, Reverse Engineering, Class Hierarchy Reconstruction, Exception Reconstruction**

I. Introduction

Today it is not uncommon for a software development company to use third-party components that are provided without source code. In such cases it is often desired to verify that these components do not include malicious code and have no security loopholes. It is also a common situation when some legacy software is used for years and no source code is available. In such situation a need may arise to fix errors in this software, improve its performance, or adapt it to the changed requirements.

Such problems are addressed by reverse engineering. Software reverse engineering may involve decompilation — translation of machine code or bytecode obtained from a compiler back into the source code in the original high level language. In order for decompilation to be correct, this source code, when compiled, must produce an executable with the same behaviour as that of the original program. Decompilation output won't be textually equivalent to the original source code, and is likely to be less comprehensible to a human.

Native code decompilation is usually considered in the context of the C programming language. A lot of research has been done on decompilation of C programs and existing decompilation methods show good results for programs that were originally written in C. Several successful commercial tools for C decompilation exist, including Hex-Rays [1], Boomerang [2] and REC [3]. However, generally they do not perform well for C++ programs.

Nowadays a great deal of software is written in C++ utilizing modern coding practices and patterns. The use of complex class hierarchies and exception handling in present-day software is becoming more and more common, even in performance-critical applications, such as database management systems. For example, C++ exceptions are used for error reporting in the kernel of the MongoDB document-oriented database [4]. As the C++ specific constructs play significant role in internal workings of present-day software, it is important to reconstruct them fully during decompilation. Besides, decompilation of C++ programs into C results in undesirable artifacts, such as unrelated compound types and unions instead of C++ inheritance hierarchy.

In this paper we discuss new methods that allow partial recovery of the C++ specific language constructs from a low-level code provided that this code was compiled by a C++ compiler. Compared to the C programming language, C++ introduces several new concepts, presenting new challenges for decompilation. These challenges include reconstruction of classes and class hierarchies, virtual and non-virtual member functions, calls to virtual functions, exception raising and handling statements. We propose solutions to these challenges and describe how they were implemented in SmartDec, a C++ decompiler.

We considered C++ ABI (application binary interface) of Microsoft Visual Studio compiler on Windows platform and C++ ABI of GNU C++ compiler on Linux. We use MSVC 10.0 and g++ 4.5.0 for empirical study, but SmartDec also

1095-1350/11 $26.00 © 2011 IEEE
DOI 10.1109/WCRE.2011.49

IEEE computer society

works for other versions of these compilers provided they use the same C++ ABI.

The rest of this paper is organized as follows. Section II discusses related work. Challenges of C++ decompilation along with possible solutions are described in Section III. In Section IV SmartDec decompiler is presented and its internal workings are described. Experimental results are discussed in Section V. Our conclusions and directions for future work are presented in the last section.

II. Related work

Currently there exists no decompiler that is capable of reconstructing C++ code. There is some support for C++ in the latest version of the Rec Studio, which supports mangled names and honors class inheritance hierarchy [3].

Skochinsky [5], [6] has given a detailed description of RTTI (run-time type information) and exception handling structures used by MSVC, along with implementation details of some of the C++ concepts, such as constructors and destructors. He has presented tools for reconstruction of polymorphic class hierarchies and exception handling statements in the assembly code. However, these tools are based on pattern matching and do not always provide correct results and cannot be used with compilers other than MSVC.

Sabanal and Yason [7] along with RTTI-based approach to class hierarchy reconstruction have proposed a technique based on the analysis of vtables (virtual function tables) and constructors that can be applied even when RTTI structures are not present in the assembly. Constructors are identified by searching for **operator new** calls followed by a function call. Vtable analysis is used for polymorphic class identification. Class relationship inference is done via analysis of constructors. Authors have also presented several examples of successful class hierarchy reconstruction. However, several cases in which presented techniques may fail are not considered. These cases include **operator new** overloading, constructor inlining and elimination of vtable references in constructors due to optimizations. The presented techniques also heavily rely on the use of MSVC-specific **__thiscall** calling convention.

In [8] authors have proposed a method for automatic reconstruction of class hierarchies that does not rely on RTTI and performs well in cases when aggressive optimizations are used by the compiler.

III. Challenges of C++ Decompilation

Compared to the C programming language, C++ introduces several new concepts, including:

- Polymorphic classes and class hierarchies.
- Virtual and non-virtual member functions.
- Exceptions and exception handling.

Existing decompilation methods show good results for programs that were originally written in C. However, when decompiling C++ programs, C++ exception handling blocks are decompiled as several unreferenced functions in place of **catch** blocks, virtual function calls are not recognized, and classes are not reconstructed.

For quality C++ decompilation, the following constructs must be recovered:

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

A. Virtual functions

In C++, a virtual function is a function with behaviour that can be overridden within an inheriting class by a function with the same signature [9]. Virtual functions are an important part of the implementation of polymorphism in C++.

The C++ standard [9] does not mandate exactly how the virtual function dispatch must be implemented. In GCC [10] and MSVC [11] ABIs it is implemented using vtables. Each vtable is an array of pointers to virtual functions, therefore the problem of identification of virtual functions for MSVC and GCC can be reduced to the problem of locating all vtables.

In SmartDec vtables are located by scanning the data segment of the low-level code and checking each location in it as it is described in [8].

B. Classes and class hierarchies

The first approach to class hierarchy reconstruction utilizes run-time type information as it is described in [8], [7], [6]. For each polymorphic class, an RTTI structure that contains information about its parents is emitted by the compiler. All classes and complete polymorphic class hierarchy can then be reconstructed by examining all RTTI structures.

In GCC and MSVC C++ ABIs, a pointer to the RTTI structure of a class always precedes its vtable [10], [11]. Therefore, the problem of finding RTTI structures can be reduced to the problem of locating vtables.

Layout of RTTI structures is defined by the ABI that is used by the C++ compiler. Once the layout is known, RTTI structures can be parsed, thus giving a complete polymorphic class hierarchy exactly as it was in the source C++ program. Since run-time type information structures contain mangled class names, class names can also be recovered.

However, RTTI is considered to be frequently misused [12], and some modern applications written in C++ refrain from using it. The use of RTTI increases binary size, and

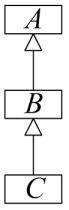

Figure 1. Example of a class hierarchy.

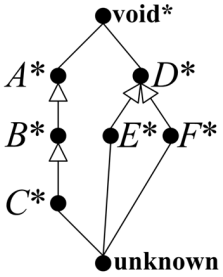

Figure 2. Example of a type lattice.

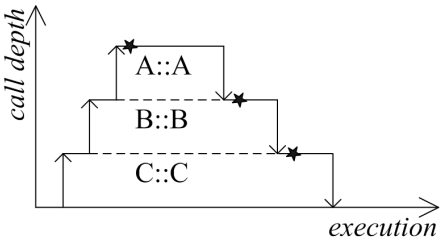

Figure 3. Outline of constructor execution for class C from Fig 1. Initializations of vtable pointer field are marked with ★.

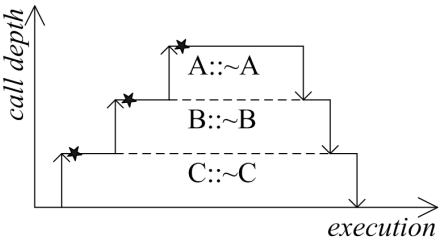

Figure 4. Outline of destructor execution for class C from Fig 1. Initializations of vtable pointer field are marked with ★.

this is why it is frequently disabled in code-size critical applications for embedded systems [13], [14]. Some frameworks replace RTTI with hand-rolled solutions because it imposes a runtime overhead or is not powerful enough for the framework's needs. Examples of such frameworks are Qt [15] and LLVM [16]. RTTI can be disabled at compile time, and in this case the above described approach to class hierarchy reconstruction cannot be used.

In case RTTI structures are not present in the assembly, an approach based on the analysis of vtables, constructors and destructors can be used for reconstruction of polymorphic class hierarchies as it is described in [8].

In SmartDec both approaches are implemented.

C. Constructors and destructors

In SmartDec constructors and destructors are detected for polymorphic classes only. Constructors and destructors of non-polymorphic classes do not differ from ordinary member functions, and therefore are difficult to detect reliably. This is one of the directions of future work.

Constructors and destructors are detected by checking the operations they perform as it is described in [8]. A constructor of a class performs the following sequence of operations [9], [11]:

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.

Interprocedural data flow analysis is used to detect consequent vtable pointer field initializations, thus locating constructors and destructors. This approach also makes it possible to detect inlined constructors and destructors.

The order in which vtable pointer field initializations are performed is determined by the inheritance order. Consider hierarchy in Fig. 1. Outlines of constructor and destructor execution for class C are provided on Fig. 3 and 4 correspondingly. As can be seen, in a call to constructor vtable pointer field is overwritten in a "base-to-derived" order, while in a call to destructor it is overwritten in a "derived-to-base" order. Also in a call to constructor, vtable pointer field is initialized *after* a call to the constructor of the base class, while in a call to a destructor, vtable pointer field is initialized *before* a call to the destructor of the base class. These heuristics are used to distinguish constructors from destructors.

D. Class pointers

Class pointers may appear in several different contexts. In each of these contexts several different values may be assigned to a single class pointer:

- A variable or a composite type member may be assigned a value several times.
- A function may be called from several different locations, with pointers to different classes from the same hierarchy passed to it as parameters.
- A function may have several exit points, and pointers to different classes may be returned at each of them.

A reconstructed type of a pointer must be compatible with types of all values assigned to it. That is, it must either be a superclass of all of them, or a generic **void**$*$ pointer. This can be achieved using a lattice model for pointer type reconstruction. First, class hierarchy is recovered and a type lattice $(\mathbb{C}, \rhd)$ is constructed, where $\mathbb{C}$ is a set of pointer types that consists of:

- the pointers to all recovered class types;
- **void**$*$ pointer;
- special **unknown** pointer type.

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** $\triangleright A$.

Example of a Hasse diagram for type lattice is presented on Fig. 2. For convenience, inheritance relations are depicted on the diagram in UML notation.

Initially, a type of a pointer is initialized to **unknown**. On each assignment, its type is set to the least common ancestor of its current type and the type of the value that is assigned to it.

Constructors that were identified on the previous step are known to return pointers of specific class types. This type information is then iteratively propagated through the assignments and function calls using the above described model for type resolution.

E. Member functions

When reconstructing non-virtual functions, it is often desired to determine if a function at hand is a member function and to find the class that it belongs to. In GCC ABI member functions are indistinguishable from free-standing functions with **this** pointer passed as the first parameter. With such ABI reliable reconstruction of member functions is not generally possible and must be performed manually. MSVC by default uses __thiscall calling convention for member functions, which passes **this** pointer in `ECX` register. In this case member functions can be reliably distinguished. The class that the member function belongs to is then determined by the type of **this** parameter.

F. Composite types

Type reconstruction algorithms [17], [18] typically assume that copies of the same pointer always reference values of the same type. In C++ this assumption is usually broken because during program execution the same pointer may point to objects of different classes from the same hierarchy. As a result, composite type reconstruction algorithm [19] merges information about the types of the base class fields and the fields of its subclasses. Since different subclasses typically have fields of different types at the same offsets, such aggregation leads to type clashes and incorrect type reconstruction.

This problem is solved using reconstructed information about class pointers. All accesses to class fields (i.e. accesses to memory locations at constant offsets from object pointer) are considered. For each access, the sum of this constant offset and corresponding memory location's size is computed. Maximum of computed sums for each class is taken as the underapproximation of class's size.

The decompiler implements modified version of the type reconstruction algorithm [19], which avoids merging information about fields lying outside the approximated object size, thus preventing type clashes.

```
; The following code performs the call
; object->function(20).
; Pointer to object is stored in esi.
; First, vtable pointer is loaded to eax
mov  eax, dword ptr [esi]
; Virtual function pointer is loaded to edx
mov  edx, dword ptr [eax+4]
; Argument is passed on the stack
push 14h
; 'this' pointer is passed in ecx register
mov  ecx, esi
; Finally, virtual function is called
call edx
```

Figure 5. Example of a virtual function call produced by MSVC.

G. Virtual function calls

For the reconstruction of virtual functions to be complete, calls to virtual functions must also be reconstructed. Some compiler optimizations may result in virtual calls being devirtualized and replaced by ordinary functions calls, or even inlined [20], [21], [22]. When not devirtualized or inlined, a virtual function call is compiled into a sequence of assembly instructions that extracts a function pointer from the vtable and performs an indirect call. An example of such a sequence produced by MSVC is presented in Fig. 5.

Virtual function calls are reconstructed after the types of the variables are recovered. Once the type of the object pointer that is used for a virtual function call is known, a simple pattern matching approach is used to detect virtual function calls. Each detected virtual function call may provide additional information about parameter types of the corresponding virtual function. This is why once a virtual function call is detected, the type analysis is rerun for all of the affected virtual functions.

H. Exception handling

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

The C++ standard defines the semantics of exception raising and handling, but leaves its implementation up to compiler vendors. We have considered two implementation schemes. In the first scheme, used by MSVC, the compiler generates code that continuously updates exception handling structures to reflect the current program state. A new element is added to the stack frame layout that contains the information on exception handlers that are available for the function

associated with that frame. If an exception is thrown, this element is used by the runtime support library to locate and execute the appropriate exception handler [23].

The second scheme, used by GCC, employs a table-driven approach and introduces no runtime overhead if exceptions are not used. It involves the creation of statically allocated tables that relate ranges of the program counter to the program state. When an exception is thrown, the runtime system looks up the current value of the program counter in these tables and determines which handlers are to be checked [10].

Exception handling, while a high-level concept, involves low-level manipulations that do not translate well into C. For example, non-trivial control flow of the exception handling cannot be implemented in C without assembly. Many algorithms for control flow analysis [24] do not take exception handling into account. As a result, **catch** blocks are isolated into separate functions. This is why proper reconstruction of exception handling requires intervention on several decompilation stages and cannot be implemented as a post-processing step that would fix the decompiled C code.

In SmartDec, exception handling structures are located and parsed after construction of the control flow graph, and additional edges of a special kind are inserted into it. These edges connect **catch** blocks with the functions they belong to, thus preventing them from being isolated into separate functions. On the high-level program generation step this edge kind information is used to guide the reconstruction of actual **catch** blocks.

Exception propagation involves the execution of destructors. As destructors for non-polymorphic classes are frequently inlined, SmartDec is currently unable to fully recognize and recover them. This is why destructor code is currently output as comments and must be integrated into the program manually.

Due to the differences in exception handling implementations between compilers, there is no universal way of reconstructing **try** blocks and **throw** statements. We describe how these constructs can be reconstructed for programs compiled with GCC.

1) Exception handling in GCC: GCC for x86 and x64 architectures by default uses Dwarf2 table-based unwinding mechanism. In Dwarf2 each function is associated with a set of *call sites*. Call sites are code sections that can potentially throw an exception, e.g. function calls or **throw** statements. Each call site is associated with a *landing pad*. A landing pad is a code block that calls destructors and transfers execution to the corresponding **catch** block. Example of a landing pad is presented on Fig. 6. Information about call sites and landing pads is statically allocated and is present in the low-level code. Details on the format of this information and on how exception handling is performed using it is given in [10] and [25].

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

Figure 6. Example of a landing pad.

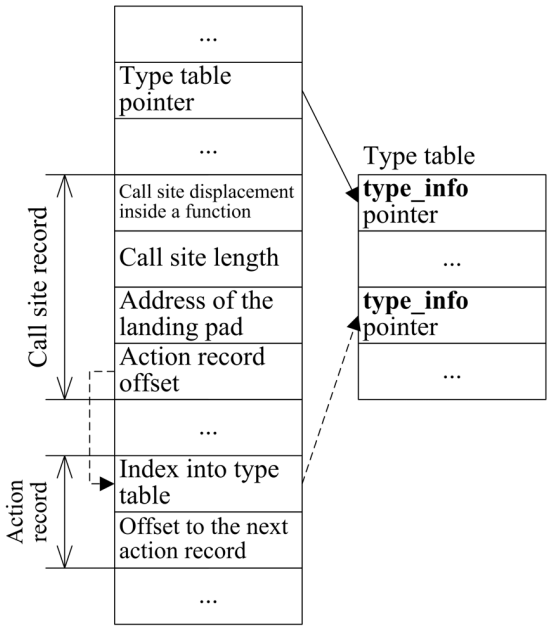

Figure 7. Structures used by GCC for exception handling.

For each call site a call site record is emitted by the compiler (see Fig. 7). It contains:

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

Each action record contains an index of an element in the table of **type_info** pointers. The list of action records describes exception types that are handled by the landing pad.

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

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

Each **catch** block is referenced from its corresponding landing pad, starts with a call to __cxa_begin_catch and ends with a call to __cxa_end_catch. Thus **catch** blocks can be

reconstructed by examining the landing pad and the locations it references.

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

Therefore call sites belonging to the same **try** block can be identified by analyzing their corresponding landing pads — if two landing pads share the same dispatch block, then their corresponding call sites belong to the same **try** block. Extents of the **try** block are then reconstructed by uniting the extents of all its corresponding call sites.

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

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

To reconstruct throw statements it is sufficient to locate the calls to __cxa_throw and find the values of its parameters.

IV. Decompiler Architecture

SmartDec decompiler is an experimental tool for analysis of programs in assembly language and their transformation into high-level code in C or C++ languages. SmartDec's workflow follows the pipeline model. Decompilation comprises several phases each using results of one or more previous ones (see Fig. 8):

1) Parsing of the input assembly listing.
2) Building of the control flow graph. Isolation of functions.
3) Parsing of statically allocated exception handling structures and fixing of the control flow graph.
4) Transformation of assembly instructions into platform-independent program representation.
5) Reconstruction of classes and class hierarchies.
6) Analysis of functions:
   a) Joint reaching definitions and constant propagation analysis.
   b) Liveness analysis, dead code elimination.
   c) Reconstruction of local variables, function arguments and return values.
   d) Reconstruction of data types.
   e) Structural analysis.
7) High-level program generation, optimization and output.

The rest of this section covers implementation details of some of these phases.

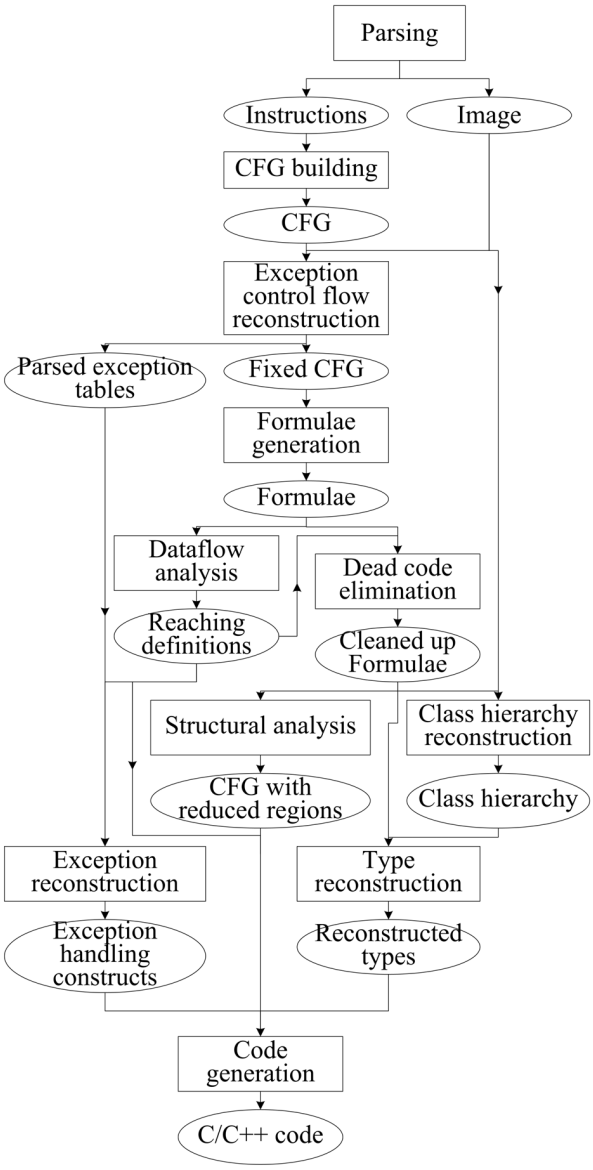

Figure 8. Decompilation workflow.

A. Parsing

Decompilation requires assembly listing of the program being analyzed to be supplied. Generation of this listing is a task of a separate disassembler tool.

Currently SmartDec can parse the output of GNU *objdump* and Microsoft *dumpbin* tools. SmartDec also integrates with IDA Pro Interactive Disassembler and can import information from its internal data structures.

At this phase the following objects are created:

- A sequence of instructions represented in a platform-dependent way.
- An image object that contains information on the sections of the binary file and provides methods for reading

its contents.

B. Isolation of functions

SmartDec uses a function reconstruction algorithm that is based on the analysis of the program control flow graph. In order to build this graph, instruction sequence is divided into basic blocks. The following addresses are used as basic block boundaries:

- jump and call destinations;
- addresses of instructions situated in memory immediately after jump instructions and empty memory areas.

Edges between basic blocks are added according to the control flow transfers performed by their last instructions. Functions are then discovered as connected components of undirected version of the control flow graph.

C. Intermediate representation

Representation of a function as a control flow graph with instructions inside basic blocks is not suitable for future analyses that require knowledge about semantics of instructions. Therefore, functions are translated into special intermediate representation called Formulae.

In Formulae representation instruction semantics is expressed in a sequence of *statements*. Statement is a command for a virtual machine that can be used for partial simulation of input assembly program.

This virtual machine has its own memory model. The memory of this machine consists of independent address spaces identified by integer numbers. Each address space represents a block of linear memory with bit addressing. Such memory model allows to abstract away the actual locations of the tracked values. Registers, stack offsets and global addresses can be assigned their own memory locations from different address spaces and tracked uniformly.

Statements accept expressions as arguments. These expressions are represented as expression trees. Nodes of an expression tree will be further referred to as *terms*. Currently supported types of terms include integer constant, memory location access (at a constant address), dereference (of an expression), unary and binary arithmetic operators.

The following types of statements are used:

Assignment performs assignment of one expression to another (typically a memory location access or dereference).

Kill forgets about previous assignments to the given memory location.

Jump performs unconditional jump to the given address.

Conditional jump performs a jump to the given address when the given condition is true.

Call performs a function call to the given address.

Return returns from a function call.

D. Dataflow analysis

The goal of dataflow analysis is to construct the most complete set of reaching definitions for each term. For this purpose joint reaching definition and constant propagation analysis [24] is performed. Such approach makes it possible to use already computed reaching definitions for further constant propagation and vice versa. The joint analysis is finished upon reaching a fixed point.

E. Liveness analysis and dead code elimination

The goal of liveness analysis is to find a set of terms that participate in calculations that are *useful*. A calculation is considered *useful* if its result can potentially affect program input, output or control flow. For example, calculation of a flag that is not used anywhere is not *useful*.

Liveness analysis is performed by propagating the *useful* state of each term through the Formulae program representation. After liveness analysis is complete, all the code that is not *useful* can be safely eliminated. In our experience, dead code elimination for x86 programs reduces the number of terms in a program almost twofold, thus speeding up other analysis steps accordingly.

F. Type reconstruction

A set of attributes is associated with a type of each term. Type reconstruction is done via deduction of these attributes, which is performed iteratively until a fixed point is reached. These attributes are then used directly for generation of type declarations in a high-level language.

The following type attributes are used:

size is the type size in bits;
integer is true if the type is an integer type;
float is true if the type is a float type;
pointer is true if the type is a pointer type;
pointee is a type of this type's dereference;
signed is true if the type is signed;
unsigned is true if the type is unsigned;
factor is a minimal value by which values of this type are incremented or decremented. This attribute is used for reconstruction of arrays.
offsets is a mapping from integer offset to the type of the field of this type at this offset. This attribute is used for reconstruction of composite types.

Inference of the attributes that are used for basic type reconstruction is performed according the rules described in [18] and [26]. Reconstruction of composite types is performed as it is described in [19].

G. Structural analysis

In order to reconstruct high-level language control flow statements, a variant [27] of structural analysis [24] is applied.

Structural analysis operates on control flow graphs of special kind. Such graphs have *regions* as nodes. A region

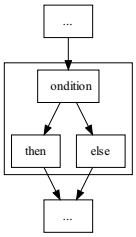

Figure 9. Example of a pattern for if-then-else region.

is a basic block or a subgraph with at most one *entry* node. The entry node of a region is a node that has incoming edges from outside the region.

Region types include cyclic regions (generic loop, while, do-while), conditionals (if-then, if-then-else, compound condition), block regions and regions of unknown type. Initially, control flow graph of a function contains single region comprising all basic blocks of this function.

Structural analysis of a region is performed via pattern matching. An example of a pattern for conditional statement is presented in Fig. 9. When a certain subgraph matches a pattern of some region type, a new region of this type is created. All nodes of the subgraph under consideration are then moved into this region, and all edges from (to) subgraph nodes are transformed into edges from (to) the region itself. Duplicate edges are then removed.

Compound conditions can be used in conditional loops and **if** statements. This is why they are reduced in the first place. This helps to correctly identify the type of the loop when it uses compound condition and disallows reconstruction of compound conditions as series of enclosed **if** statements.

Next, all cyclic regions are reduced. When a cyclic region is reduced, all edges corresponding to **break** and **continue** statements are removed from it. This simplifies control flow structure inside the loop and allows to recover conditional statements with **break** and **continue** statements inside. Since cyclic regions can have complex control flow structure inside, structural analysis is performed in newly created cyclic regions recursively.

Then block statements are recovered in order to assemble each branch of an **if** statement into a single node. At last, if-then and if-then-else regions are reduced.

The final fully reduced graph is used directly during code generation. Control structure in generated program is conveyed via C control statements and, when necessary, explicit control transfer operations **break**, **continue** and **goto**. Computed gotos are represented by **goto** statements having expressions as the argument.

```
struct BinaryFunction {
  virtual int calculate(int a, int b) = 0;
};

struct GCD: public BinaryFunction {
  virtual int calculate(int a, int b);
};

struct Pow: public BinaryFunction {
  virtual int calculate(int a, int b);
};

int GCD::calculate(int a, int b) {
  if (b == 0)
    return a;
  else
    return GCD::calculate(b, a % b);
}

int Pow::calculate(int a, int b) {
  int result = 1;
  for(int i = 0; i < b; i++)
    result *= a;
  return result;
}
```

Figure 10. C++ source. Irrelevant code omitted.

```
struct C0 {
  virtual int32_t f_401367(int32_t a1, int32_t a2) = 0;
}

struct C1: C0 {
  virtual int32_t f_401367(int32_t a1, int32_t a2);
}

struct C2: C0 {
  virtual int32_t f_401367(int32_t a1, int32_t a2);
}

int32_t C1::f_401367(int32_t a1, int32_t a2) {
  if (a2 == 0) {
    return a1;
  } else {
    return C1::f_401367(a2, a1 % a2);
  }
}

int32_t C2::f_401367(int32_t a1, int32_t a2) {
  int32_t v1;
  int32_t v2;
  v1 = 1;
  v2 = 0;
  while (v2 < a2) {
    v2 = v2 + 1;
    v1 = v1 * a1;
  }
  return v1;
}
```

Figure 11. Decompiled C++ source corresponding to the code on Fig. 10. Irrelevant code omitted.

V. Implementation and experimental results

SmartDec was tested on a variety of open-source software written in C++ and on several hand-crafted tests. This section presents the results of testing of C++-related functionality. The algorithms used for C decompilation do not undergo significant changes when used for decompilation of C++ programs. Experimental results of the basic type reconstruction algorithm that is used in SmartDec are

presented in [26]. Troshina et. al. presented an empirical study of the composite type reconstruction algorithm that is used in SmartDec [19]. SmartDec's structural analysis algorithm has been analyzed in [27].

As SmartDec is yet unable to produce directly compilable code, the decompiled code cannot be tested by recompilation and has to be verified manually. A fragment of one of the tests from our test suite is presented on Fig. 10. The corresponding decompiled code is presented on Fig. 11.

Testing of the reconstruction of exception handling constructs was performed manually. Description of the tests is presented in Table I. Columns "Application" and "File" contain the names of the application and the source file of this application and columns "Try", "Catch" and "Throw" show the number of reconstructed **try** blocks, **catch** blocks and **throw** statements respectively. Manual verification has shown that in all these tests all exception handling constructs present in source file were reconstructed correctly. No spurious constructs were recovered.

For testing of class hierarchy reconstruction correctness, the following automatic process is used. First, the program is compiled with optimizations and RTTI enabled, and RTTI-aware class hierarchy reconstruction algorithm is used to collect information about polymorphic class hierarchy. This algorithm always provides correct results [8]. The program is then recompiled with optimizations but without RTTI, and class hierarchy reconstruction algorithm that does not use RTTI is applied. Compiler-generated debug information is then used to restore the correspondence between class hierarchies reconstructed on the first and second steps. Two class hierarchies are then compared.

Test results are presented in Table II. For each of the analyzed applications all vtables present in the assembly were found.

"Non-vtables" row refers to the reconstructed vtables that were not present in the source program. Static arrays of function pointers that are used in the same way as vtables fall into this category.

Application	File	Try	Catch	Throw
notepad++	Parameters.cpp	1	1	4
shareaza	Security.cpp	5	5	0
shareaza	DownloadGroups.cpp	3	3	0
mongodb	database.cpp	3	3	3
—	061_multi_try.cpp	2	3	2

Table I
TEST RESULTS FOR EXCEPTION RECONSTRUCTION.

Application	mysqld	shareaza	notepad++	doxygen
Vtables found	950	1128	95	415
Non-vtables	0.2%	0%	0%	0%
Vtable mismatches	3.1%	2.8%	4.0%	4.8%
Classes found	918	1101	95	397
Class mismatches	5.5%	5.0%	10.5%	6.3%

Table II
TEST RESULTS FOR CLASS HIERARCHY RECONSTRUCTION.

Mismatch rates are calculated taking into account the following categories of classes.

- Classes that do not override any of the virtual functions of their bases. Vtables for such classes can be optimized away by the compiler, thus leading to incorrect class hierarchy reconstruction. As this doesn't introduce any changes in the semantics of the program, such cases are not treated as mismatches.
- Classes with no data members and no actions performed in constructors and destructors. Hierarchies of such classes can be rearranged in virtually any way without changing the semantics of the program. Such cases are not treated as mismatches.

"Vtable mismatches" row refers to vtables where the reconstructed parent vtable differs from the real one. For example, if a vtable was reconstructed as not having a parent, while it actually has one, then this is a mismatch. Most of vtable mismatches were registered as a result of the parent vtable being located in a shared library. The reason for this is that SmartDec currently does not support joined analysis of several low-level programs. Other mismatches were due to vtable pointer field initializations being optimized away by the compiler.

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

- Classes that contain mismatched vtables.
- Classes that contain other polymorphic classes as fields. Inheritance and aggregation may be indistinguishable, and this is why such mismatches do not necessarily lead to misinterpretation of the program semantics.

VI. CONCLUSION AND FURTHER WORK

We have described the challenges of C++ decompilation and proposed several methods that can be used to overcome them. These methods allow automatic reconstruction of polymorphic classes and hierarchies, virtual and non-virtual member functions, constructors and destructors, calls to virtual functions, types of class members and their layout, types of variables that store pointers to polymorphic classes and exception raising and handling statements.

Proposed methods were implemented in SmartDec, a C++ decompiler that is being developed by the authors at Select LTD. SmartDec was tested on a variety of open-source software written in C++ and showed good results.

Directions for future work include generation of C++ code that can be compiled and tested without prior manual editing.

REFERENCES

[1] Hex-rays decompiler. [Online]. Available: http://www.hex-rays.com/

[2] Boomerang decompiler. [Online]. Available: http://boomerang.sourceforge.net/

[3] Rec, reverse engineering compiler. [Online]. Available: http://www.backerstreet.com/rec/rec.htm

[4] Mongodb, schema-free document-oriented database. [Online]. Available: http://www.mongodb.org/

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

[6] ——. (2006, September) Reversing microsoft visual c++ part 2: Classes, methods and rtti. [Online]. Available: http://www.openrce.org/articles/full_view/23

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

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

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

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

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

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

[13] C. A. Quiroz, "Using c++ efficiently in embedded applications," *Proceedings of the Embedded Systems Conference*, November 1998.

[14] "Technical report on c++ performance n1487=03-0070," ISO/IEC JTC1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep., 2003.

[15] Qt - a cross-platform application and ui framework, http://qt.nokia.com/products/. [Online]. Available: http://qt.nokia.com/products/

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

[17] A. Mycroft, "Type-based decompilation," *European Symp. on Programming*, vol. 1576, pp. 208–223, 1999.

[18] K. Dolgova and A. Chernov, "Automatic type reconstruction in disassembled c programs," *Proceedings of the WCRE'08*, October 2008.

[19] K. Troshina, Y. Derevenets, and A. Chernov, "Reconstruction of composite types for decompilation," *Source Code Analysis and Manipulation, IEEE International Workshop on*, vol. 0, pp. 179–188, 2010.

[20] M. Namolaru, "Devirtualization in gcc," *Proceedings of the GCC Developers Summit*, pp. 125–134, June 2006.

[21] D. Bacon and P. Sweeney, "Fast static analysis of c++ virtual function calls," *SIGPLAN Not.*, vol. 31, pp. 324–341, October 1996. [Online]. Available: http://doi.acm.org/10.1145/236338.236371

[22] S. Porat, D. Bernstein, Y. Fedorov, J. Rodrigue, and E. Yahav, "Compiler optimization of c++ virtual function calls," *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems*, pp. 3–14, 1996.

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

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

[25] Linux standard base core specification 3.0rc1, chapter 8. exception frames. [Online]. Available: http://refspecs.freestandards.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html

[26] K. Troshina, A. Chernov, and Y. Derevenets, "C decompilation: Is it possible?" *Proceedings of International Workshop on Program Understanding*, pp. 18–27, June 2009.

[27] E. Derevenets and K. Dolgova, "Structural analysis in the problem of decompilation," *Applied Mathematics*, vol. 4, pp. 87–99, August 2009.