A Novel Lightweight Virtual Machine Based Decompiler to Generate C/C++ Code with High Readability

Gengbiao Chen  Zhuo Wang  Ruoyu Zhang  Kan Zhou
Shiqiu Huang  Kangqi Ni  Zhengwei Qi
School of Software, Shanghai Jiao Tong University
Shanghai, China
{kawar, horreaper, holmeszry, zhoukan, hsqfire, vincent.nkq, qizhwei}@sjtu.edu.cn

ABSTRACT

As a key part of reverse engineering, decompilation plays a very important role in software security and maintenance. Many decompilation techniques and tools have been developed while all of them have defects in different aspects. For example, IDA Hex_rays generates pseudocodes with poor readability, and Boomerang is unable to identify composite structures such as Classes and multidimensional arrays. In this paper we present C-Decompiler, an integrated C/C++ decompiler based on lightweight virtual machine, which is capable to analyze the data flow especially the data dependency across basic blocks with high accuracy. C-Decompiler is able to recognize composite structures and libraries, such as Standard Template Library. Our experimental results show that on average C-Decompiler has the highest reduction rate of 55.91% and lowest expansion rate of 55.79%, which prove that the code decompiled by C-Decompiler is most similar to the original code in comparison with other existing decompilers such as IDA Hex_rays and Boomerang. Furthermore, C-Decompiler is able to recognize all the functions without any false positive nor false negative, and it is convincing to support that the code decompiled by C-Decompiler is of high readability by the least redundancy and most accuracy.

Categories and Subject Descriptors
D.2.7 [Distribution, Maintenance, and Enhancement]: [Restructuring, reverse engineering, and reengineering]

General Terms
 Algorithms, Reliability

Keywords
Reverse Engineering, Decompilation

1. INTRODUCTION

So far, the issues on software security and maintenance have become hot spots. In order to adapt the software to inconstant demands and prevent it from the malicious behaviors, reverse engineering has been considered to be one of the most effective approaches and the decompilation has drawn many people’s attention as a key part of reverse engineering in the last two decades.

There are already several existing decompilation systems which can transfer the binary code into the high level source code, such as IDA Hex_rays [4], Boomerang [1] and dcc [5]. Unfortunately all of these systems suffer from disabilities in different aspects.

Current decompilers are far from mature. Figure 1 shows the C code generated by a simple procedure which is to multiply a 2-dimensional array with another one, but the output of IDA Hex_rays contains 400 LOCs(Line of Code) approximately.

Variable identification is an issue in decompilation. Figure 1(①) presents a snapshot of the variables identified by the IDA Hex_rays. These global variables have never been defined or used in the C program and they account for one-tenth of all the variables recognized. The reason might be inaccurate data flow analysis. Function identification is not well handled in the system either. In Figure 1(②), some function calls are from system libraries such as kernel32.dll and ntdll.dll while others are unknown and have never been defined or called through the whole program.

Type information recovery is another focus of recent researches. Figure 1(③) shows a part of the code transferred from the main function. The system separates the array into individual variables instead of identifying it and makes the output code poorly readable. Moreover, those variables defined as int in the original program have become much more complex after the decompilation. There are already several studies in this field. K.Dolgova [17] presented a type reconstruction algorithm for assembly code compiled by C compilers. The algorithm recovers the primitive types with lattice theory and reconstructs the composite types by building set of accessible offsets. Mycroft [20] also presented a type recovery method and accomplished a type based decompilation system.
Besides IDA Hex-rays, there are also some other decompilation systems. As a general decompiler of machine code programs, Boomerang [1] implements data flow analysis via SSA (Static Single Assignment) and presents a retargetable decompilation system. However, Boomerang is not able to identify composite structures like Classes which limits the usage of the system.

In this paper, we have implemented a decompilation system and make three main contributions:

- Binary interpretation based on lightweight virtual machine
  In order to construct the layout of the calls and gain more information about the variables, C-Decompiler [2] supervises the binary via lightweight virtual machine mechanism. A shadow stack is used to track the behaviors of the system stack.

- Inter-basic-block data flow analysis
  Since the execution path is uncertain while analyzing the binary across the basic blocks (BBs), it is very difficult to find out the dependency between variables. The disability of across BBs analysis causes great deal of redundant source code generation. In this paper, we describe an accurate method to address this problem and simplify the output.

- Library and composite structure identification
  C++ code usually contains many library calls. The identification of those library calls can remarkably reduce the complexity of the generated code. An approach is proposed to recognize the calls to libraries such as system libraries and STL (Standard Template Library). The system also extracts the switch case structure, identifies the arrays and increases the readability of the output source code greatly.

The remaining parts of the paper are organized as follows:

Section 2 describes the techniques implemented in the system in details. Section 3 gives a complete view of the system. In Section 4, we show the evaluation with the comparison to the existing decompilation systems. The related researches are introduced in Section 5 and the conclusion is drawn in Section 6.

2. TECHNIQUES DESCRIPTION

This section presents a detailed description of our key techniques. The first one is the lightweight virtual machine introduced by C-Decompiler and the second one is inter-basic-block register propagation and the last one is STL identification.

![Figure 2: The structure of our lightweight virtual machine. It consists of an X86 parser, a storage and a stack.](image-url)
lead to wrong results. In Figure 3, an example is given to show how the classic algorithm identifies parameters and local variables. As Figure 3 shows, the quantity of parameters and local variables identified by the algorithm is wrong, and a mistake is made in the data flow analysis.

In order to solve the problems above, a lightweight virtual machine (LVM) has been introduced in our design. Figure 2 shows the structure of the virtual machine. The LVM includes an x86 instruction parser, a storage and a stack. The parser analyzes the input instructions and updates the state of the stack. The storage stores all the data used in the LVM, with VirtualESP included. VirtualESP is declared to represent the offset between esp_func and the current value of esp. esp_func stands for the value of esp at the entry of each function. The parser concerns two types of instructions, one of which changes the value of esp, and the other reads or writes the stack. For the first type of instructions, the parser revises the value of VirtualESP according to how the instruction influences the stack. For example, after the instruction sub esp, 0x30, VirtualESP is decreased by 0x30. For the second type, the parser computes the memory location with VirtualESP, and decides what the memory location represents, a parameter or a local variable. For instance, in the line 6 of the previous example, VirtualESP is -0x3C, and the memory location used in the destination operand is updated by esp_func + 0x24 - 0x3C = esp_func - 0x18. Thus in the line 6 the value of ebx is passed to a local variable, instead of a parameter.

The call and ret instructions should be taken into special consideration. According to the __stdcall calling convention, the callee functions are responsible for recovering the stack before they return. Thus VirtualESP is increased by a number decided by the parameters of the callee functions. Moreover, there might be more than one ret instruction in a function, which may cause confusion in identifying the exit of the function. We solve the problem by recording each destination address of jump instructions. If the return address is farther than any jump destination, this ret instruction represents the exit of the function.

Figure 4 shows how the LVM handles the previous example. With the help of a LVM, all the two problems have been solved. In the line 2, the source operand points to esp_func+0x4, which represents a parameter. In the line 6 and 12, both of the destination operands point to esp_func-0x18, which stands for a local variable. In the line 9, the value of ecx comes from the top of the stack, esp_func-0x34. In the line 2, eax is written to the memory location esp_func-0x34, which is not used before. Thus a new local variable has been recognized. After LoadIcon returns, esp_func-0x34 becomes the top of the stack, and the value of esp_func - 0x34 is passed to ecx in the line 9.

2.2 Inter-Basic-Block register propagation

Register propagation is an important component of data flow analysis. With the help of removal of temporary registers and abbreviation of instructions, expressions close to high-level language will be formed with the information reconstructed that is lost during the compilation. Register propagation occurs among a series of value assignment instructions where some registers play the role of temporary variables and they should be removed through the register propagation. In addition two instructions that transfer the value between by a temporary variable should be combined into one instruction.

The register propagation has already been realized in the previous work limited inside of the BB. While the register propagation inside of the BB results in the incomplete propagation. The previous register propagation is mostly based on the algorithm as follows.

In the algorithm, CanDoPropagate() is to judge whether the rhs-clear condition [5] is satisfied to propagate the register. The rhs-clear path is an x-clear path from the identifiers x in an expression that defines a register r that satisfies ud-chain...
**Algorithm: dcc Register Propagation**

procedure ExtRegCopyProp
  /* Pre: dead-register analysis has been performed.*/
  *dead-condition code analysis has been performed.
  *register arguments have been detected.
  *function return registers have been detected.
  * Post: temporary registers are removed from the
  *intermediate code. */
  intExpStk();
  for (all basic blocks b of function ) do
    for (all instructions j in b) do
      for (all registers r used by instruction j) do
        if ((ud(r) = ( def)) && CanDoPropagate())
          /* uniquely defined at instruction def*/
          DoPropagate();
          end if
        end for
      end for
    end for
  end for

**Algorithm: Inter-BB Register Propagation**

procedure RegPropAcrossBB
  for (all ret instructions k in the program) do
    ConstructPath(path);
    /*Construct all the instruction paths for all the ret*/
    end for
  for (all instructions j using register r) do
    XBB_ud(r) = ComputeUD();
    /*Compute the ud-chains based on constructed*/
    instruction paths
    end for
  for (all instructions j in function) do
    for (all registers r used by instruction j) do
      if ((XBB_ud(r) = (def)) && CanDoAcrossBB())
        DoPropagate();
        end if
      end for
    end for
  end for

procedure CanDoAcrossBB
  if (path of r is unique) //r is only appear in one path.
    CanDoPropagate();
  end if

in the loop are added only once and all the paths are constructed from the first instruction to all the ret instructions if a program has multiple ret instructions. To be practical, not all the instruction paths but all the instruction paths related to the decompilation are constructed and saved. Each instruction may appear in different ud-chains because each instruction has possibility of being in different instruction paths.

**The dcc Decompiler**

<table>
<thead>
<tr>
<th>The dcc Decompiler</th>
<th>C-Decompiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ((loc0 - 2) != 0) { if ((loc0 - 2) != 0) {</td>
<td>if ((loc0 - 2) != 0) {</td>
</tr>
<tr>
<td>if ((loc1 - 13) != 0) { if ((loc0 - 2) - 13) != 0) { code0;</td>
<td>if (loc0 - 2) - 13) - 258) != 0) {</td>
</tr>
<tr>
<td>if ((loc2 - 258) != 0) { code0;</td>
<td>code0;</td>
</tr>
<tr>
<td>} else { code3; }</td>
<td>} else { code3; }</td>
</tr>
<tr>
<td>} else { code2; }</td>
<td>} else { code2; }</td>
</tr>
<tr>
<td>} else { code1; }</td>
<td>} else { code1; }</td>
</tr>
</tbody>
</table>

Figure 5: The comparison of the decompiled codes from the dcc decompiler and C-Decompiler. This is mainly to illuminate the difference of the common method and the inter-BB method. The code decompiled by the dcc decompiler is on the left and the one by C-Decompiler is presented on the right.

Then the decompiled code by the inter-BB method is compared with the one by the previous method, and the result is presented in Figure 5. Obviously the inter-BB method reconstructs the code with high accuracy and readability. The variables loc0, loc1, loc2 from dcc are actually related to loc0 in our result with both accurate and clear information about the relationship of the variables. However the inter-BB method may lead to increase of the overhead. It consumes a lot of time and space to construct and store the instruction paths. In consideration of the requirement of the decompilation which is needed only once for each program, the speed and space are not the utmost concerns like

(1) use of chain) condition to the instruction that uses the register r. If there is no other definition of x along the path, x-clear turns out to be true here. And DoPropagate() is to execute the register propagation. The previous method limits the register propagation inside of the BB. Consider the following example.

01 SUB eax, 2 06 JE L3
02 JE L1 07 code0
03 SUB eax, 0Dh 08 L1: code1
04 JE L2 09 L2: code2
05 SUB eax, 102h 10 L3: code3

In the example eax in instruction 1 cannot be propagated to the instruction 3 in another BB, because instruction 3 is not executed as expected after the execution of instruction 1. Similarly eax in instruction 3 cannot be propagated to the instruction 5. In addition eax in instruction 3 and 5 are considered as two new variables, and the relationship among the variables is imprecise. The result decompiled by dcc is displayed in Figure 5. The shortages result from the execution of register propagation in the phase of data flow analysis. At that moment relationships of BBs cannot be acquired since the control flow analysis has not been performed yet. Thus the propagation is restricted inside of the BB.

It is a challenge to realize the register propagation across BBs, because it is hard to identify the ud-chain. On one hand the following instructions of other BBs are not certain to be executed and on the other hand the ud-chain of one instruction may come from different instructions in different BBs. To solve the problem instruction path is introduced in our work.

Definition 1: Instruction path is a set of sequential instructions, and the successors in the set are certain to be executed if any instruction of the set is executed.

The algorithm to solve the inter-BB register propagation is as follows:

Under construction of the instruction paths, the instructions
the correctness and the readability. It is quite acceptable to take 10-20 seconds to decompile Microsoft notepad.exe with the binary code size of 67K.

2.3 STL identification based on signature

To identify STL is one of the significant phases in the process of decompilation. Our approach, signatures are generated to identify STL. First, .cpp files containing all STL contain Classes and member functions are compiled by different compilers with different options, and a series of .obj files are generated. Then we analyze each .obj file and take the machine code of container Classes and member functions to generate signatures. All the calls will be compared with generated signatures. Any matched result is regarded as an identified STL function.

Generating signatures is challenging because various parameter types will lead to different machine code even of the same member function. On the other hand, some functions of STL may have the same signature, which results in signature conflict. Two methods are taken to solve these problems respectively.

```
(a) int vector::max_size(void)
01 push ebp
02 mov ebp,esp
03 sub esp,0CCh
04 push ebx
05 push esi
06 push edi
07 push ecx
08 lea edi,[ebp-0CCh]
09 call std::allocator<int>::max_size
10 mov esp,ebp
11 pop ebp
12 ret

(b) void vector::clear(void)
01 push ebp
02 mov ebp,esp
03 sub esp,0CCh
04 push ebx
05 push esi
06 push edi
07 push ecx
08 lea edi,[ebp-0CCh]
09 call std::allocator<int>::clear
10 mov esp,ebp
11 pop ebp
12 ret
```

Figure 6: An example of STL signature conflict. Most of the assembly code of function (a) and function (b) are the same except for the call instructions in the line 09. If relocation information of call instructions in the line 09 is replaced with “00”s, the two functions have the same signature.

2.3.1 Generation of general signature

The common parts of the machine code of different parameter types are extracted as a signature called general signature. But generating general signatures is complicated because original signatures produced for different parameter types may have different lengths when the common parts of them are extracted. To solve this problem, special symbols, i.e. “00”s, will be continually inserted into the shorter one in the position where difference in successive bytes are detected, and different bytes of them are also replaced with “00”s to extract the common parts of original signatures.

2.3.2 Resolution of signature conflict

An example of signature conflict is shown in Figure 6. To resolve signature conflicts, the original general signature will be linked by the special symbol “&” with the machine code of callee functions, the addresses of which can be found in the corresponding .obj file. After that a new unique signature is generated. Figure 7 is an example of generated STL signature.

2.3.3 STL identification with signatures

STL call is identified by comparing the machine code of a function call with general signatures. As shown in Figure 7, a general signature is pieces of sub-signatures split by “****”s, and relocation information is ignored, which accounts for the insertion of “00”s when signatures are generated. It needs to match all these pieces when a function call is identified. Figure 8 is an example of STL identification. The variable in type of vector and the callee function can be identified by C-Decompiler. However, C-Decompiler cannot identify the type information of parameters if relying on signatures only.

```
(a) int vector::max_size(void)
01 push ebp
02 mov ebp,esp
03 sub esp,0CCh
04 push ebx
05 push esi
06 push edi
07 push ecx
08 lea edi,[ebp-0CCh]
09 call std::allocator<int>::max_size
10 mov esp,ebp
11 pop ebp
12 ret

(b) void vector::clear(void)
01 push ebp
02 mov ebp,esp
03 sub esp,0CCh
04 push ebx
05 push esi
06 push edi
07 push ecx
08 lea edi,[ebp-0CCh]
09 call std::allocator<int>::clear
10 mov esp,ebp
11 pop ebp
12 ret
```

Figure 7: An example of generated STL signature. The red part is the common part of original signatures. “****” represents continual-inserted “00”s when general signature is being generated. The blue part is the machine code of the callee function to make the signature unique.

```
(a) vector<int> vi;
(b) xor esi,esi
    mov dword ptr [esp+10h],esi
    mov dword ptr [esp+20h],esi
    push ecx
    lea eax,[esp+18h]
    push eax
    lea eax,[esp+10h]
    push eax
    lea eax,[esp+18h]
    mov dword ptr[esp+3Ch],esi
    push ecx
    lea eax,[esp+24h]
    mov dword ptr[esp+18h],0Ah
    call std::vector<int>::allocator<int>::insert

(c) vector loc1;
    loc1.push(10);
```

Figure 8: An example of STL identification. Considering the original code in (a), (c) is the output of STL identification by C-Decompiler with the input of assembly code in (b).

3. SYSTEM DESIGN
C-Decompiler provides a mechanism to decompile the binary into C/C++ code with high accuracy, and the general approach works as follow:

**Binary Analyzer** is the first module to process the binary code as shown in Figure 9. Unlike other decompilation systems, C-Decompiler supervises the whole procedure via a Stack Monitor by which the information about variables can be gained, such as the call layout. **Library Identifier** is also implemented to simplify the program by recognizing the library calls. These two auxiliary modules are described in Section 2.

**Semantics Analyzer** then checks the code for idioms. An idiom is a sequence of instructions whose behaviors can be summarized automatically. The semantics analysis makes the propagation of the variables and the structure of the program more intuitive.

In **IR Generator** the binary code is translated into the IR(Intermediate Representation) for further analysis and interpretation. To facilitate the data flow analysis, each instruction in the IR only contains one simple operation. Cifuentes has given a detailed description in [5].

Since IR is acquired, the system starts to run under an **IR Based Analysis Subsystem**. The subsystem is designed for analyzing the program and transferring it into source code in the environment of IR. The functional modules in this subsystem take effect by accessing and modifying IR.

**CFG Constructor** is able to construct the CFG(Control Flow Graph) of the program with the information retrieved in the former modules. When constructing the BBs, instructions will be added into the current BB until an ending instruction such as conditional jump or call is encountered. While resolving the program, the CFG building engine identifies the predecessors and successors of each BB and links them to form the CFG.

Then **Data Flow Analysis Engine** begins to build a *ud-chain* to parse the data flow of the program. As a result of the uncertainty of the execution path, the semantic of the transfer among the registers is not determined across the BBs, and the existing decompilation systems can’t link the data dependency in a global view. C-Decompiler presents a new approach to address this problem which is described in Section 2.

**Control Flow Analysis Engine** is started up to reconstruct the structure of the program after analyzing the dependency among the variables. The control flow analysis is generally implemented with the method of pattern matching [23]. In the high level language, control flow structure can be classified. There are several studies about the patterns such as branches and loops [12, 26]. By means of matching the CFG generated by the system with the patterns, C-Decompiler identifies the structure embedded in the IR.

**Source Code Generator** outputs the C/C++ code with the information gathered in the previous stages. The high level language code generation is organized according to the call graph. After the consideration of global variables, the system rebuilds the whole program by traversing the call graph. In C-Decompiler, the assembly code is also provided by an **Assembly Generator**.

4. EXPERIMENTAL RESULTS

This section presents a series of programs decompiled by C-Decompiler [2]. These programs illustrate different aspects of the decompilation process. Comparisons between the source code and the results of C-Decompiler are presented. Furthermore, the results of other 2 decompilers, Boomerang [1] and Hex-rays [4], the decompilation plugin of IDA Pro, are also provided. The comparisons contain 3 aspects, including function analysis, variable analysis and total percentage reduction [5](\(\text{reduction}\%\)). The formula of \(\text{reduction}\%\) is formula (1).

\[
\text{reduction}\% = \frac{M - N}{M} \times 100\%
\]

\[
\text{variable}\% = \frac{P - Q}{Q} \times 100\%
\]

\(M\) means the number of high-level instructions in the decompiled code, and \(N\) means the number of assembly instructions in the target executable. \(\text{expansion}\%\) represents the ratio of the quantity of variables declared in original code and decompiled code, which is computed by formula (2). When computing \(\text{expansion}\%\), \(P\) means the total number of variables in decompiled code, and \(Q\) means the amount of variables used in original code. Decompiled code with lower
4.1 Case study

In this part, we evaluate the decompilation quality of C-Decompiler by analyzing the decompiled code of Win32.exe. Win32 is a Windows application program, which creates a graphic window. The source code of Win32.exe is automatically generated by Visual Studio 2008. Figure 10 lists the original and decompiled code. The decompiled code from IDA Hexrays and Boomerang are omitted, but their results are shown in analysis processes in Table 2.

According to the decompiled code, we could have a preliminary impression that C-Decompiler is capable of producing code with relatively high readability.

4.1.1 Function analysis

In this section, we discuss the identification accuracy of functions. All the functions appeared in the code can be divided into two types. The first one is User-Defined Functions (UDF), for example, MyRegisterClass, InitInstance and proc1. The other one is Windows API. Totally, there are 4 UDFs and 17 APIs in the original code. C-Decompiler identifies 4 UDFs, as the same number as the original code. Our tool confirms 17 APIs, all of which are correctly identified (CI). There is neither omission nor excessiveness in identifying APIs. Thus the false positive rate (fp%) and false negative rate (fn%) are both 0%. Zero UDFs and 31 APIs are reported by IDA Hexrays, whose fp% is as high as 45.16%. Boomerang can-
Figure 11: Function-call trees of the code decompiled. (a), (b), (c) and (d) are the function-call trees of the original code, C-Decompiler, Hex-rays and Boomerang respectively. The nodes are functions. The green nodes represent APIs, and the blue ones stand for UDFs.

<table>
<thead>
<tr>
<th>The original code</th>
<th>C-Decompiler</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = a + b + c;</code></td>
<td><code>loc2=((loc2+loc3)+loc5);</code></td>
</tr>
<tr>
<td><code>b = a &gt;&gt; 1;</code></td>
<td><code>loc3 = (loc2 &gt;&gt; 1);</code></td>
</tr>
<tr>
<td><code>a = b % 10;</code></td>
<td><code>loc2 = (loc3 % 10);</code></td>
</tr>
<tr>
<td><code>a = b == c;</code></td>
<td><code>loc2 = loc3 == loc5;</code></td>
</tr>
<tr>
<td>`b = a</td>
<td>c;`</td>
</tr>
<tr>
<td><code>a = lb;</code></td>
<td><code>loc2 = !loc3</code></td>
</tr>
</tbody>
</table>

Figure 12: The decompiled code fragment of Hallint. C-Decompiler recognizes all the operators.

not identify all the APIs, and its fn% is 70.59%. These statistics are listed in Table 2.

Figure 11 shows the function-call tree of the original code and decompiled code. The function-call tree represents the relationship of the functions. It is obvious that the tree of the decompiled code produced by C-Decompiler is most close to the one of the original code. The only difference between the two trees is the son nodes’ sequence of node 3. The reason is that there is a switch-case structure in the code segment of the UDF substituted by node 3. The change of the son nodes’ position has no effect on the execution of the program. The trees of IDA Hex-rays and Boomerang are quite different from the one of original code. This means the constructions of decompiled code produced by IDA Hex-rays and Boomerang are quite different from original code.

4.1.2 Variable analysis
There are total 11 variables declared in the original code, which contain 6 pointers, 3 Class variables and 2 basic type variables. Three of all the 11 variables are global ones. C-Decompiler recognizes 21 variables, four of which are global. expansion% of C-Decompiler is 81.82%. IDA Hex-rays declared the most variables, and expansion% is 563.64%. The rate of the code decompiled by Boomerang is 181.82%. All the statistics are listed in Table 2. In all the 3 decompilers, C-Decompiler uses least variables, which makes its results easy to understand.

4.2 Decompilation results
4.2.1 Computation intensive test
The computation intensive test includes two programs, Hall int.exe and Float.exe, which can be used to test the ability to decompile operators and basic variable types. Hallint.exe is a program from the Plum-Hall benchmark suite, which benchmarks integers. Hallint.exe contains 8 computation instructions with 7 operators, and repeats the operations in a loop. Float.exe is a program from page 58, Eckel Bruce. Thinking in C++ Second Edition. China Machine Press, 2008, which performs computation operations on float variables. Not only int variables, Hallint.exe can be used to test other integer variables, such as short and char. Totally, there are 5 variable types and 7 operators in the computation intensive programs. The quantity of operators and types identified by the 3 decompilers is listed in Table 3. Figure 12 shows the decompiled code fragment.

Figure 13: The recursive function in fibo. C-Decompiler recognizes the recursive function

1http://www.plumhall.com/
### Table 2: Statistics of the case study

<table>
<thead>
<tr>
<th></th>
<th>UDFs</th>
<th>APIs</th>
<th>CI APIs</th>
<th>fp%</th>
<th>fn%</th>
<th>variables</th>
<th>expansion%</th>
<th>EL</th>
<th>reduction%</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-Decompiler</td>
<td>4</td>
<td>17</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>11</td>
<td>-417</td>
<td>-</td>
<td>349</td>
</tr>
<tr>
<td>IDA Hex-rays</td>
<td>12</td>
<td>31</td>
<td>17/17</td>
<td>0%</td>
<td>0%</td>
<td>20</td>
<td>81.82%</td>
<td>121</td>
<td>65.30%</td>
</tr>
<tr>
<td>Boomerang</td>
<td>7</td>
<td>8</td>
<td>5/17</td>
<td>45.16%</td>
<td>0%</td>
<td>73</td>
<td>563.64%</td>
<td>417</td>
<td>-368.54%</td>
</tr>
</tbody>
</table>

**4.2.2 String operation intensive test**

String operation intensive test consists of Valstring.exe and StrAPI.exe. The source code of Valstring.exe is taken from line 187 - line 222, linux-loader.c in Valgrind 3.4.1 [22]. It reads the command line, and decides which client and tool are used. StrAPI uses 11 string-related APIs to perform various operations on strings. The statistics of APIs identified by the decompilers are listed in Table 3.

![Figure 14: Summary of reduction rate of the 3 decompilers.](image)

The red, green and blue lines and nodes stand for the reduction% of C-Decompiler, Boomerang and Hex-rays respectively. The dotted lines represent the average values. The higher lines mean the better performance. Generally speaking, the red lines are the highest, which means the length of the code decompiled by C-Decompiler is closest to the length of the original code.

**Figure 15: Summary of variable expansion rate.** From the two figures we can come to a conclusion that on average C-Decompiler has the highest reduction rate and lowest expansion rate, which means that C-Decompiler produces fewest redundant variables and code lines.

**4.3 Summary of results**

Figure 14 shows the overall reduction% of the 3 decompilers. On average, reduction% of C-Decompiler is 54.82%, which is the highest of the three decompilers. Boomerang has only 4 nodes because it cannot produce results for notepad_prime.

**5. RELATED WORK**

**5.1 Decompiler**

exe2c [5] is an early effort to decompile executable files, while its results are not satisfactory. ddc [3] by Cristina Cifuentes is considered to be the definitive work on general decompilation from binary files. This work draws heavily on standard forward engineering techniques [5, 12] and graph techniques [7] are used to restructure the generated code into standard loops, conditional statements and other high level structures [6]. But their type recovery is limited to simple data type [13]. Cristina Cifuentes keeps working on the subject and has made a lot of contributions [8–11,14] in this field.

University of Queensland Binary Translator (UQBT) also by Cristina Cifuentes performs decompilation from the binary code to an obscure low-level C form as an intermediate step in binary translation. Boomerang [1] is an open source decompiler which aims at creating a retargetable decompiler based on some of the ddc and UQBT ideas. With several front ends and a C back end, Boomerang uses an internal representation based on the SSA form. Also it pioneers data flow based type analysis [24]. The main theme of PhD thesis of Mike Van Emmerik [24] is how the SSA form enables various aspects of decompilation to be more readily analyzed. Comparing to zero false positive and zero false negative in our work, 15.29% false positive and 82.14% false negative on average are not so convincing.
An approach and technological support are presented to recover object-oriented models stereotyped with crosscutting concern indications which are identified automatically from object-oriented code specifically Java in [15]. The STARC tool [18] which uses a constraint-based search to repair broken data structures is probably not practical for large data structures in deployed systems. Philip J. Guo et al. describe a dynamic unification-based analysis [19] for inferring abstract types. And experiments indicate that the inferred abstract types are useful for program comprehension, improve both the results and the efficiency of a follow-on program analysis.

6. CONCLUSION

This paper presents C-Decompiler, a lightweight virtual machine based decompiler which is able to analyze data dependency across BBs as well as identify composite data structures and libraries including STL, and is capable to produce compilable C/C++ code with relatively high readability. In our experiments on function analysis and variable analysis, C-Decompiler can recognize all the UDFs and APIs functions, all the variables including pointers and Class variables and almost all the control statements without any false positive and false negative. Furthermore, C-Decompiler has the highest reduction rate of 55.91% and lowest expansion rate of 55.79% and can produce C/C++ code which is the most similar to original code when compared to other existing decompilers such as Boomerang and IDA Hex_rays. In the near future, we will develop a set of program analysis tools such as taint analysis based on this efficient decompiler.

7. ACKNOWLEDGEMENTS

This work is supported by National Natural Science Foundation of China (Grant No.60773093 and 60873209), the Key Program for Basic Research of Shanghai (Grant No.09JC1407900), Ministry of Education and Intel Joint Research Funding, IBM SUR Funding and IBM Research-China JP Funding.

8. REFERENCES