2010 17th Working Conference on Reverse Engineering

A Refined Decompiler to Generate C Code with High Readability

Gengbiao Chen†, Zhuo Wang†, Ruoyu Zhang†, Kan Zhou†, Shiqiu Huang†, Kangqi Ni†, Zhengwei Qi†, Kai Chen‡, Haibing Guan‡
† School of Software
‡ School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University
{ kawar†, horreaper†, holmeszry†, zhoukan†, hsqfire†, vincent.nkq†, qizhwei†, kchen‡, hbguan‡ } @ sjtu.edu.cn

***Abstract*—As a key part of reverse engineering, decompilation plays a very important role in software security and maintenance. Unfortunately, most existing decompilation tools suffer from the low accuracy in identifying variables, functions and composite structures, which results in poor readability. To address these limitations, we present a practical decompiler called C-Decompiler for Windows C programs that (1) uses a shadow stack to perform refined data flow analysis, and (2) adopts inter-basic-block register propagation to reduce redundant variables. Our experimental results illustrate that on average C-Decompiler has the highest total percentage reduction of 55.91%, lowest variable expansion rate of 55.79% in the three tools, and the same Cyclomatic Complexity as the original source code for each test application. Furthermore, in our experiment, C-Decompiler is able to recognize functions with lower false positive and false negative rate. In the studies, we show that C-Decompiler is a practical tool to produce highly readable C code.**

***Keywords*-Reverse Engineering, Decompilation;**

I. Introduction

Source code is very valuable, especially as most of the commodity software will not provide it. Gaining information from the binary code is difficult, while it can be applied easily with the source code. Decompiling the binary code into the source code has drawn much attention in the last two decades [9].

There are already several existing decompilers which can transfer the binary code into the high level source code, such as *IDA Hex_rays* [2], *Boomerang* [1] and *dcc* [5]. Unfortunately, all of these systems suffer from the low identification accuracy of variables, functions, and types in different aspects.

Type information recovery [4] is a great challenge in recent researches. K. Dolgova [8] presented a type reconstruction algorithm for assembly code, which recovers the primitive types with lattice theory and reconstructs the composite types by building a set of accessible offsets.

Flow Graph building is another key point in the decompilation field. In [7], Cristina Cifuentes presents a flow graph structuring algorithm by using a set of generic high-level language structures such as loops and conditionals. Another inter-procedural data flow analysis method is described in [6].

In this paper, we present a practical decompiler called C-Decompiler for Windows programs with special emphasis on high readability and make two contributions:

- *Shadow stack based behavior analysis*
  In order to construct the layout of the function calls and gain more refined information about the variables, C-Decompiler adopts a shadow stack mechanism to track the behaviors of the application's stack.
- *Inter-basic-block data flow analysis*
  We present a new inter-basic-block data flow analysis to find out the relationship of variables among basic blocks, which is useful to eliminate redundant variables.

The rest of the paper is organized as follows. Section 2 describes the techniques implemented in C-Decompiler in details. In Section 3, we show the evaluation with the comparisons to IDA Hex_rays and Boomerang. Finally we conclude in Section 4.

II. Technique Description

This section presents a detailed description of our key techniques. The first one is the shadow stack mechanism introduced by Stack Monitor. The second one is inter-basic-block register propagation in Data Flow Analysis Engine.

A. Shadow Stack

The analysis of the stack [3] is a key method to identify parameters and local variables. Cristina Cifuentes [5] proposed the classic algorithm of identifying parameters and local variables. The algorithm neglects operations on the stack (eg., changing the stack pointer *esp*), which can lead to wrong data flow analysis results.

In Figure 1, an example is given to show how the classic algorithm identifies parameters and local variables. Several mistakes are made in its data flow analysis. For example, two parameters and one local variable are recognized, but there are actually one parameter and two local variables. Furthermore, according to this algorithm, the value of *ecx* in line 12 comes from line 7, but this value should be assigned from the stack in line 9.

In order to solve the problem above, a shadow stack is introduced in our design. *VirtualESP* (Figure 2) 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. Two sets of instructions are concerned, one of which changes the value of *esp*, and the other reads or

1095-1350/10 $26.00 © 2010 IEEE
DOI 10.1109/WCRE.2010.24

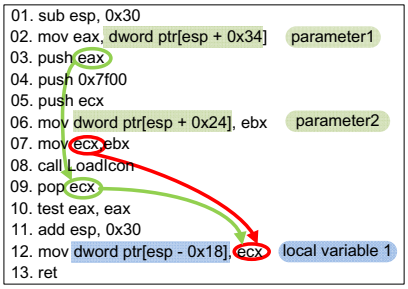

Fig. 1. An example of how the classic algorithm works. The memory locations with green mark are parameters, and blue for local variables. The red arrowed curve is the propagation path of *ecx* according to the classic algorithm, while the green curve is the correct path.

writes the stack. For the first set of instructions, the value of *VirtualESP* is modified according to how the instruction influences the stack. For example, after the first instruction *sub esp, 0x30*, *VirtualESP* is decreased by $0x30$. For the second one, we decide what the memory location actually represents, a parameter or a local variable, by adding *VirtualESP* to it. For instance, in line 6 of Figure 2, *VirtualESP* is $-0x3C$, and the memory location used in the destination operand is updated by $esp\_func + 0x24 - 0x3C = esp\_func - 0x18$. Thus in line 6 the value of *ebx* is assigned to a local variable, because its address ($esp\_func - 0x18$) is below the stack base.

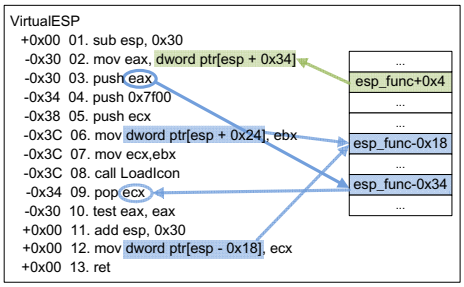

Fig. 2. With the help of the shadow stack, we can see that line 6 and line 12 write to the same memory location. Totally one parameter and two local variables are identified. Moreover, the correct data path of *ecx* is recognized.

Figure 2 shows how we handle the previous example. With the help of the shadow stack, we recognize the local variables, parameters, and propagation paths correctly.

B. Inter-basic-block register propagation

With the help of removal of temporary registers, expressions close to high-level language will be formed with the reconstructed information that is lost during the compilation. The intra-basic-blocks register propagation has already been proposed in the previous work [5], which results in the incomplete propagation.

The traditional *dcc* register propagation [5] is described in Figure 3. In this algorithm, *CanDoPropagate()* is to judge whether the *rhs-clear* condition propagates 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* (use-define chain) condition to the instruction that uses the register r [5]. 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.

Algorithm: *dcc* Register Propagation

```
pRocedure ExtRegCopyProp

initExpStk()
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
```

Fig. 3. Algorithm: intra-basic-blocks register propagation [5].

This method limits the register propagation only inside the *BB* (Basic Block). This shortcoming results 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 the *BB*.

Due to the difficulty of identifying the *ud-chain*, it is challenging to implement the register propagation across *BBs*. First, the execution path is nondeterministic. Second, the *ud-chain* of one instruction may come from different *BBs*. To address this problem, *Instruction path* is introduced in our work.

Instruction path is the sequential instructions executed in order. Instruction path has several entries along this path, but only one exit, in contrast to basic block which has one entry and one exit.

Our new inter-*BB* register propagation is listed in Figure 4. Compared to the traditional algorithm, ours has three improvements, *ConstructPath()*, *ComputeUD()*, and *CanDoAcrossBB()*. *CanDoPropagate()* and *DoPropagate()* realize the same function with the traditional algorithm. In *ConstructPath()*, the instructions 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, only the *instruction paths* related to the decompilation are constructed and saved. Each instruction may appear in different *ud-chains* because each instruction has a possibility in different *instruction paths*.

An example is presented in Figure 5. 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. The inter-*BB* method consumes a lot of time and space to construct and store the *instruction paths*, while it is quite acceptable to take only 2.94 seconds to decompile *Microsoft notepad.exe* with the binary

Algorithm: Inter-BB Register Propagation

```
procedure InterBBRegCopyProp

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 registers 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
```

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

Fig. 4. Algorithm: inter-basic-blocks register propagation.

Binary Code			The *dcc* Decompiler	*C-Decompiler*
01	SUB	eax, 2	if ((loc0 - 2) != 0){	if ((loc0 - 2) != 0) {
02	JE	L1	if ((loc1 - 13) != 0) {	if ((((loc0 - 2) - 13) != 0) {
03	SUB	eax, 0Dh	if ((loc2 - 258) != 0) {	if ((((loc0 - 2) - 13) - 258) != 0) {
04	JE	L2	...	...
05	SUB	eax, 102h	}else {//L3	}else {//L3
06	JE	L3	...}	...}
07	...		}else {//L2	}else {//L2
08	L1	...	...}	...}
09	L2	...	}else {//L1	}else {//L1
10	L3	...	...}	...}

Fig. 5. The comparison of the decompiled results from *dcc* and C-Decompiler. This is mainly to illuminate the difference between the traditional method and the inter-*BB* method.

code size of 67K.

III. Experimental Results

This section presents a series of programs decompiled by C-Decompiler. These programs illustrate different aspects of the decompilation process. The comparisons among the original source code and the code decompiled by C-Decompiler, Boomerang [1], and Hex_rays [2] (the decompilation plugin of the famous disassembler IDA Pro), are provided.

We use four evaluation criteria in our experiments. The first is function analysis, which evaluates the quality of identifying functions. All functions are divided into 2 types, *UDF* (user defined functions) and A&L (API and library functions). We list the quantity of *UDF* and A&L for each test program, and compute $fp\%$ (false positive) and $fn\%$ (false negative). Second, $expansion\%$ (variable expansion rate) reflects how many additional variables are used in the decompiled code, compared with the original code. Third, $reduction\%$ [5] depicts the reduction percentage of code lines when the program is represented by high level language such as C/C++ instead

(a)Original code	(b)Decompiled code

int APIENTRY _tWinMain(...)
{
MSG msg;
HACCEL hAccelTable;
LoadString(...);
LoadString(...);
MyRegisterClass(hInstance);
if (!InitInstance (hInstance, nCmdShow))
{
return FALSE;
}
hAccelTable = LoadAccelerators(...));
while (GetMessage(&msg, NULL, 0, 0))
{
if (!TranslateAccelerator(...))
{
TranslateMessage(&msg);
DispatchMessage(&msg);
}
}
return (int) msg.wParam;
}
	int _tWinMain(...){
	HACCEL loc1;
	MSG loc2;
	int loc3; /* eax */
	LoadStringW (...);
	LoadStringW (...);
	proc_1 (hInstance);
	if (proc_2 (hInstance, nCmdShow) == 0) {
	loc3 = 0;
	}
	else {
	loc1 = LoadAcceleratorsW (...);
	while((GetMessageW (loc2,...) != 0)) {
	if(TranslateAcceleratorW (...)== 0){
	TranslateMessage (&loc2);
	DispatchMessageW (&loc2);
	}
	} /* end of while */
	loc3 = loc2.wParam;
	}
	return (loc3);
	}

Fig. 6. The original and decompiled code of the case study. (a) is the original code, and (b) is the code decompiled by C-Decompiler. Only the main functions are listed, and some parameters of functions are omitted. C-Decompiler recognizes all functions and variables.

of assembly code. Fourth, CC (Cyclomatic Complexity)[1] is a software metric developed by Thomas J Mcbabe. It directly measures the number of linearly independent paths through a program's source code.

This section is divided into 4 parts. At first the benchmark programs are introduced. Then a case study is presented. In the third part, the decompiled results of the programs are provided. In the last part, we focus on the efficiency of the decompilers. The experiment data are collected on the platform with the configuration listed in Table I. Furthermore, most of the code decompiled by C-Decompiler can be directly compiled, and the generated executables have the same functions as the original ones.

CPU	*intel Core2* 3.00GHz
Memory	*DDR2* 3GB
Disk	*Seagate SATA* 1TB
Compiler	*Visual C++ 2008*
OS	*Windows XP SP3*

TABLE I
The Experiment Testbed.

A. Benchmark Programs

Our benchmark suite consists of 7 programs, which are listed in Table II. These programs evaluate decompilation quality in various aspects. The *LOC* (lines of code) of these programs are ranging from 23 to 3349.

B. Case study

We perform this case study on Win32.exe, whose source code is automatically generated by *Visual Studio 2008* when we start up a new Windows project. Its *LOC* is 191.

Figure 6 lists the original and decompiled code. The decompiled code from IDA Hex_rays and Boomerang is omitted, but their results are shown in Table III.

[1]http://en.wikipedia.org/wiki/Cyclomatic_complexity

TABLE II
Evaluation Programs

Programs	Source	LOC	Description
hallint	Plum-hall benchmark	34	A computation-intensive program to evaluate the ability to decompile operators.
fibo	A math algorithm	124	A function call intensive program. It calculates Fibonacci numbers by a recursive algorithm.
Win32	Generated by Visual Studio	205	An A&L intensive program. This program creates an empty window with Windows APIs.
Notepad	Windows XP	-	A comprehensive test for decompilers. It is one of the most popular tools under Windows.
specrand	SPECINT 2006 benchmark	77	The benchmark generates a sequence of pseudorandom numbers starting with a known seed.
MatrixMul	A math algorithm	23	This program multiples two matrixes, which are represented by 2 two-dimension arrays.
TextEditor	An open source tool	3349	TextEditor is a real world word processor, and is taken as a comprehensive test for decompilers.

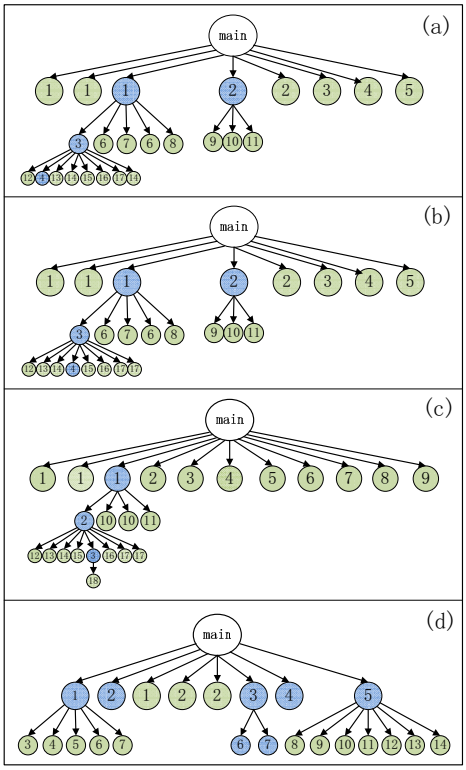

Fig. 7. Function-call trees of the code decompiled. (a), (b), (c) and (d) are the function-call trees of the original code, C-Decompiler, Hex_rays, and Boomerang, respectively. The nodes are functions. The green nodes represent APIs, and the blue ones stand for *UDF*.

The number of *EL (effective lines)* in the code decompiled by C-Decompiler, IDA Hex_rays and Boomerang is 121, 1026, 455, respectively. The $reduction\%$ of the 3 decompilers are 65.30%, -368.54%, and -30.40%, respectively. The CC of the original code and the result of C-Decompiler are both 11, while the CC for Hex_rays and Boomerang are 30 and 27, respectively.

1) Function analysis: In this section, we discuss the identification accuracy of functions. Totally, there are 4 *UDF* and 17 A&L in the original code. C-Decompiler identifies 4 *UDF*, as the same number as the original code. Our tool confirms 17 A&L, all of which are *correctly identified (CI)*. The false positive rate ($fp\%$) and false negative rate ($fn\%$) are both 0%. Zero *UDF* and 31 A&L are reported by IDA Hex_rays, whose $fp\%$ is as high as 45.16%. Boomerang cannot identify all the A&L, and its $fn\%$ is 70.59%. These statistics are listed in Table III.

Figure 7 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 child 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 child 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.

2) Variable analysis: There are total 11 variables declared in the original code. Three of all the 11 variables are global ones. C-Decompiler recognizes 21 variables, four of which are global. The $expansion\%$ of C-Decompiler is 81.82%. IDA Hex_rays declared the most variables, and the $expansion\%$ is 563.64%. The rate of the code decompiled by Boomerang is 181.82%. All the statistics are listed in Table III. In all the 3 decompilers, C-Decompiler uses least variables, which makes its results easy to understand.

C. Decompilation results

This section provides the statistics of decompiled results from benchmark programs. The results of Boomerang are incomplete because it cannot generate results for *Windows* application *notepad.exe* and *TextEditor.exe*. We do not perform analysis on *notepad.exe* because its source code is inaccessible. C-Decompiler recognizes all 7 operators in *hallint.exe* and all recursive functions in *Fibo.exe*. The decompilation statistics of *specrand* and *TextEditor* are listed in Table IV and Table V, respectively.

Figure 8 shows the overall $reduction\%$ of the 3 decompilers. Figure 9 shows the overall variable expansion rate. From the two figures we conclude that on average C-Decompiler has the highest reduction rate and lowest expansion rate, which means that it produces the fewest redundant variables and code lines.

D. Decompilation Efficiency

Figure 10 shows the performance of the decompilers. C-Decompiler needs approximate 2.94 seconds to decompile *notepad.exe*. In the experiments, C-Decompiler performs the

TABLE III
Statistics of the case study

Code Origin	UDF	A&L	CI A&L	fp%	fn%	variables	expansion%	EL	reduction%	CC
source	4	17	-	-	-	11	-	349	-	11
C-Decompiler	4	17	17/17	0%	0%	20	81.82%	121	65.30%	11
IDA Hex_rays	12	31	17/17	45.16%	0%	73	563.64%	417	-368.54%	30
Boomerang	7	8	5/17	37.50%	70.59%	31	181.82%	455	-30.40%	27

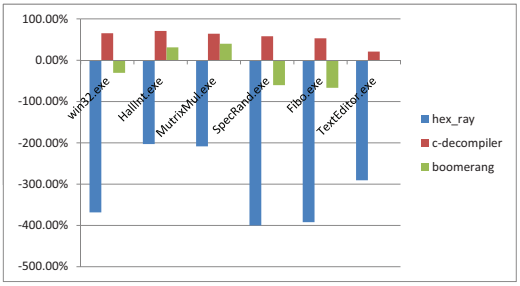

Fig. 8. Summary of reduction rate of the 3 decompilers. The red, green and blue bars stand for the $reduction\%$ of C-Decompiler, Boomerang, and Hex_rays, respectively. The higher bars mean the better performance. Generally speaking, the red bar is the highest, which means the length of the code decompiled by C-Decompiler is closest to the length of the original code.

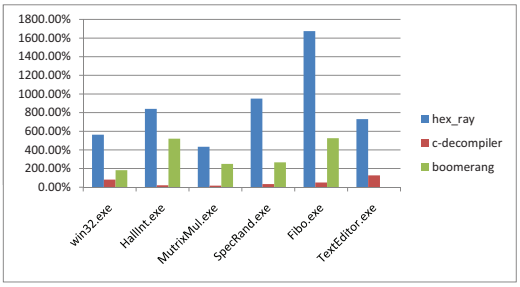

Fig. 9. Summary of variable expansion rate. The relationship of the colors and the decompilers is the same to the Figure 8. The lower bars present the better performance. Generally speaking, the red bar is the lowest. This means the quantity of variables in the code decompiled by C-Decompiler is the closest to the one of the original code.

lowest speed. However, efficiency is not a key feature for a decompiler because programs need to be decompiled only once.

IV. CONCLUSION

This paper presents C-Decompiler, an accurate decompiler which can generate highly readable source code. First, a shadow stack is implemented to gain more refined information about the variables. Second, C-Decompiler is capable of analyzing data dependency across basic blocks which greatly reduces the redundant variables. It can also extract the switch case structure, identify the arrays and increase the readability of the decompiled source code greatly.

TABLE IV
Analysis results of specrand.exe

Code Origin	UDF	A&L	CI A&L	fp%	fn%	CC
Original Source	1	11	-	-	-	5
C-Decompiler	1	11	11/11	0%	0%	5
Hex_rays	12	6	6/11	0%	45.5%	23
Boomerang	8	2	1/11	50.0%	92.6%	17

TABLE V
Analysis results of TextEditor.exe

Code Origin	UDF	A&L	CI A&L	fp%	fn%	CC
source	52	62	-	-	-	43
C-Decompiler	52	62	62/62	0%	0%	43
Hex_rays	73	128	62/62	51.56%	0%	103

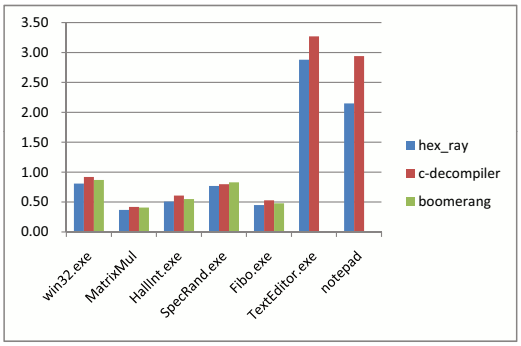

Fig. 10. The decompilation time for the three decompilers.

V. ACKNOWLEDGEMENT

This work is supported by National Natural Science Foundation of China (Grant No.60773093, 60873209, and 60970107), the Key Program for Basic Research of Shanghai (Grant No.09JC1407900, 09510701600), IBM SUR Funding and IBM Research-China JP Funding.

REFERENCES

[1] Boomerang. *http://boomerang.sourceforge.net*.
[2] IDA hex-rays. *http://www.hex-rays.com/*.
[3] C. W. Appel and A. P. Felty. A semantic model of types and machine instuctions for proof-carrying code. In *POPL*, pages 243–253, 2000.
[4] P. T. Breuer and J. P. Bowen. Decompilation: The enumeration of types and grammars. *ACM Trans. Program. Lang. Syst.*, 16(5):1613–1647, 1994.
[5] C. Cifuentes. *Reverse Compilation Techniques*. School of Computing Science PhD thesis, Queensland University of Technology, 1994.
[6] C. Cifuentes. Interprocedural data flow decompilation. *J. Prog. Lang.*, 4(2):77–99, 1996.
[7] C. Cifuentes. Structuring decompiled graphs. In *CC*, pages 91–105, 1996.
[8] K. Dolgova and A. Chernov. Automatic type reconstruction in disassembled c programs. In *WCRE*, pages 202–206. IEEE, 2008.
[9] M. Van Emmerik. *Static Single Assignment for Decompilation*. School of ITEE PhD thesis, University of Queensland, 2007.