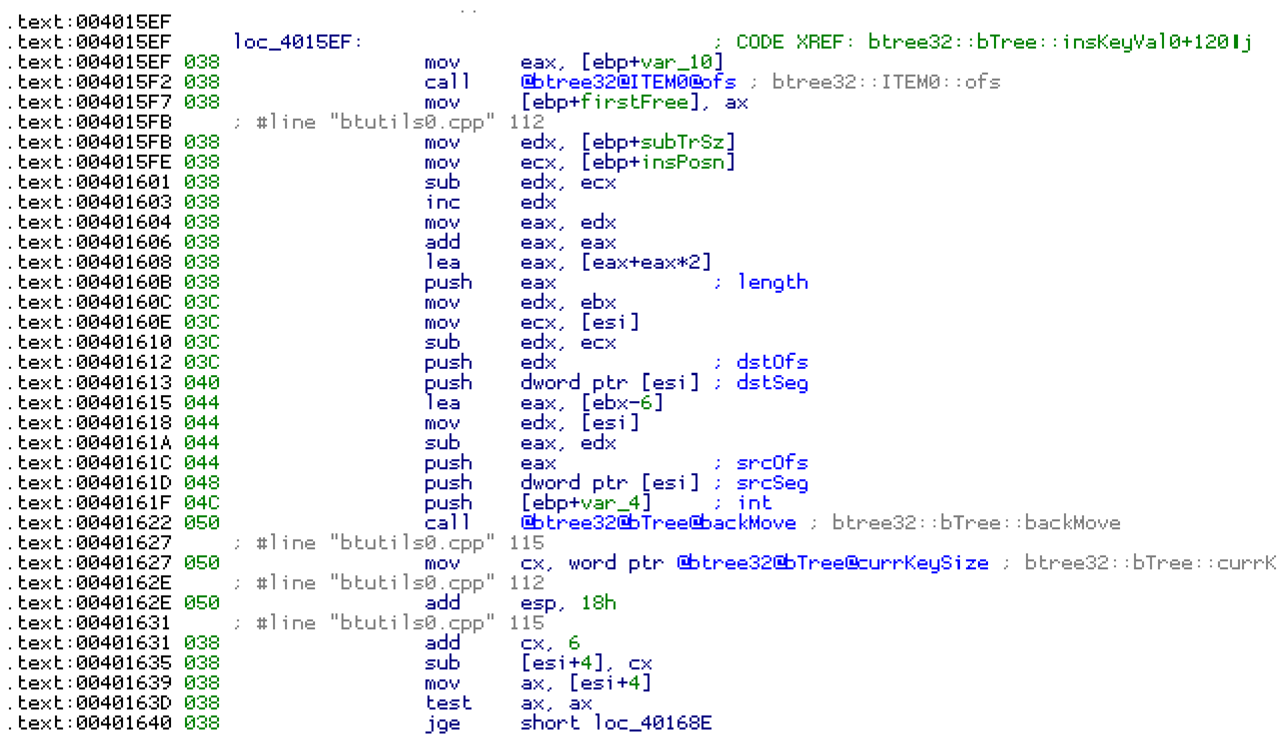
Hex-Rays
http://www.hex-rays.com
E-mail: info@hex-rays.com Tel: +32-4-222-4300 Fax: +32-4-223-5600

Decompilation vs. Disassembly

A **decompiler** represents executable binary files in a readable form. More precisely, it transforms binary code into text that software developers can read and modify. The software security industry relies on this transformation to analyze and validate programs. The analysis is performed on the binary code because the source code (the text form of the software) traditionally is not available, because it is considered a commercial secret.

Programs to transform binary code into text form have always existed. Simple one-to-one mapping of processor instruction codes into instruction mnemonics is performed by **disassemblers.** Many disassemblers are available on the market, both free and commercial. The most powerful disassembler is IDA Pro, published by Datarescue. It can handle binary code for a huge number of processors and has open architecture that allows developers to write add-on analytic modules.



```
.text:004015EF
.text:004015EF     loc_4015EF:                             ; CODE XREF: btree32::bTree::insKeyVal0+120↑j
.text:004015EF 038                 mov     eax, [ebp+var_10]
.text:004015F2 038                 call    @btree32@ITEM0@ofs ; btree32::ITEM0::ofs
.text:004015F7 038                 mov     [ebp+firstFree], ax
.text:004015FB     ; #line "btutils0.cpp" 112
.text:004015FB 038                 mov     edx, [ebp+subTrSz]
.text:004015FE 038                 mov     ecx, [ebp+insPosn]
.text:00401601 038                 sub     edx, ecx
.text:00401603 038                 inc     edx
.text:00401604 038                 mov     eax, edx
.text:00401606 038                 add     eax, eax
.text:00401608 038                 lea     eax, [eax+eax*2]
.text:0040160B 038                 push    eax                ; length
.text:0040160C 03C                 mov     edx, ebx
.text:0040160E 03C                 mov     ecx, [esi]
.text:00401610 03C                 sub     edx, ecx
.text:00401612 03C                 push    edx                ; dstOfs
.text:00401613 040                 push    dword ptr [esi] ; dstSeg
.text:00401615 044                 lea     eax, [ebx-6]
.text:00401618 044                 mov     edx, [esi]
.text:0040161A 044                 sub     eax, edx
.text:0040161C 044                 push    eax                ; srcOfs
.text:0040161D 048                 push    dword ptr [esi] ; srcSeg
.text:0040161F 04C                 push    [ebp+var_4]        ; int
.text:00401622 050                 call    @btree32@bTree@backMove ; btree32::bTree::backMove
.text:00401627     ; #line "btutils0.cpp" 115
.text:00401627 050                 mov     cx, word ptr @btree32@bTree@currKeySize ; btree32::bTree::currK
.text:0040162E     ; #line "btutils0.cpp" 112
.text:0040162E 050                 add     esp, 18h
.text:00401631     ; #line "btutils0.cpp" 115
.text:00401631 038                 add     cx, 6
.text:00401635 038                 sub     [esi+4], cx
.text:00401639 038                 mov     ax, [esi+4]
.text:0040163D 038                 test    ax, ax
.text:00401640 038                 jge     short loc_40168E
```

Pic 1. Disassembler output

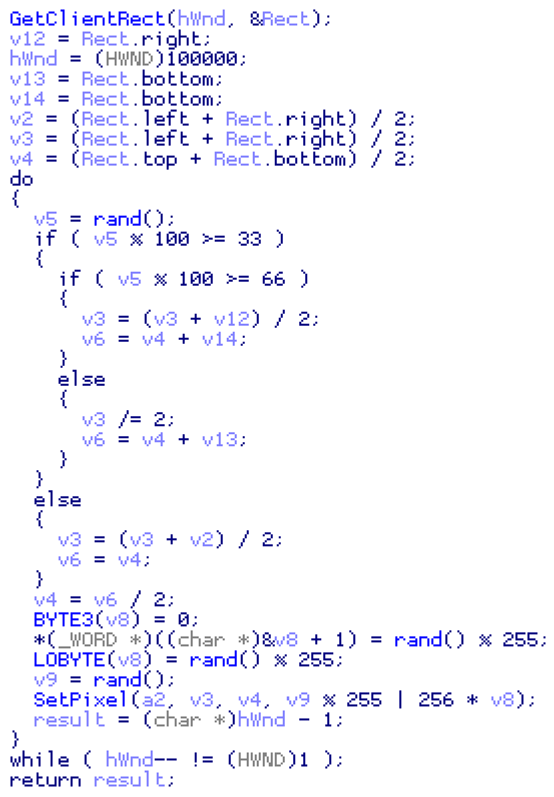
Decompilers are different from disassemblers in one very **important aspect**. While both generate human readable text, decompilers generate much higher level text, which is more concise and much easier to read.

Compared to low level assembly language, high level language representation has several advantages:

- ✔ It is consise.
- ✔ It is structured.
- ✔ It doesn't require developers to know the assembly language.
- ✔ It recognizes and converts low level idioms into high level notions.
- ✔ It is less confusing and therefore easier to understand.
- ✔ It is less repetitive and less distracting.
- ✔ It uses data flow analysis.

Let's consider these points in detail.

Usually the decompiler's output is **five to ten times shorter** than the disassembler's output. For example, a typical modern program contains from 400KB to 5MB of binary code. The disassembler's output for such a program will include around 5-100MB of text, which can take anything from several weeks to several months to analyze completely. Analysts cannot spend this much time on a single program for economic reasons.



```
GetClientRect(hWnd, &Rect);
v12 = Rect.right;
hWnd = (HWND)100000;
v13 = Rect.bottom;
v14 = Rect.bottom;
v2 = (Rect.left + Rect.right) / 2;
v3 = (Rect.left + Rect.right) / 2;
v4 = (Rect.top + Rect.bottom) / 2;
do
{
  v5 = rand();
  if ( v5 % 100 >= 33 )
  {
    if ( v5 % 100 >= 66 )
    {
      v3 = (v3 + v12) / 2;
      v6 = v4 + v14;
    }
    else
    {
      v3 /= 2;
      v6 = v4 + v13;
    }
  }
  else
  {
    v3 = (v3 + v2) / 2;
    v6 = v4;
  }
  v4 = v6 / 2;
  BYTE3(v8) = 0;
  *(_WORD *)((char *)&v8 + 1) = rand() % 255;
  LOBYTE(v8) = rand() % 255;
  v9 = rand();
  SetPixel(a2, v3, v4, v9 % 255 | 256 * v8);
  result = (char *)hWnd - 1;
}
while ( hWnd-- != (HWND)1 );
return result;
```

Pic 2. Decompiler output

The decompiler's output for a typical program will be from 400KB to 10MB. Although this is still a big volume to read and understand (about the size of a thick book), the time needed for analysis time is divided by 10 or more.

The second big difference is that the decompiler output is **structured**. Instead of a linear flow of instructions where each line is similar to all the others, the text is indented to make the program logic explicit. Control flow constructs such as conditional statements, loops, and switches are marked with the appropriate keywords.

The decompiler's output is easier to understand than the disassembler's output because it is **high level**. To be able to use a disassembler, an analyst must know the target processor's assembly language. Mainstream programmers do not use assembly languages for everyday tasks, but virtually everyone uses high level languages today. Decompilers remove the gap between the typical programming languages and the output language. More analysts can use a decompiler than a disassembler.

Decompilers convert assembly level **idioms** into high-level abstractions. Some idioms can be quite long and time consuming to analyze. The following one line code

```
x = y / 2;
```

can be transformed by the compiler into a series of 20-30 processor instructions. It takes at least 15-30 seconds for an experienced analyst to recognize the pattern and mentally replace it with the original line.. If the code includes many such idioms, an analyst is forced to take notes and mark each pattern with its short representation. All this slows down the analysis tremendously. Decompilers remove this burden from the analysts.

The amount of assembler instructions to analyze is huge. They look very similar to each other and their patterns are very repetitive. Reading disassembler output is nothing like reading a captivating story. In a compiler generated program 95% of the code will be really boring to read and analyze. It is extremely easy for an analyst to confuse two similar looking snippets of code, and simply lose his way in the output. These two factors (the size and the boring nature of the text) lead to the following phenomenon: binary programs are never fully analyzed. Analysts try to locate suspicious parts by using some heuristics and some automation tools. Exceptions happen when the program is extremely small or an analyst devotes a disproportionally huge amount of time to the analysis. Decompilers alleviate both problems: their output is shorter and **less repetitive**. The output still contains some repetition, but it is manageable by a human being. Besides, this repetition can be addressed by automating the analysis.

Repetitive patterns in the binary code call for a solution. One obvious solution is to employ the computer to find patterns and somehow reduce them into something shorter and easier for human analysts to grasp. Some disassemblers (including IDA Pro) provide a means to automate analysis. However, the number of available analytical modules stays low, so repetitive code continues to be a problem. The main reason is that recognizing binary patterns is a surprisingly difficult task. Any “simple” action, including basic arithmetic operations such as addition and subtraction, can be represented in an endless number of ways in binary form. The compiler might use the addition operator for subtraction and vice versa. It can store constant numbers somewhere in its memory and load them when needed. It can use the fact that, after some operations, the register value can be proven to be a known constant, and just use the register without reinitializing it. The diversity of methods used explains the small number of available analytical modules.

The situation is different with a decompiler. Automation becomes much easier because the decompiler provides the analyst **with high level notions**. Many patterns are automatically recognized and replaced with abstract notions. The remaining patterns can be detected easily because of the formalisms the decompiler introduces. For example, the notions of function parameters and calling conventions are strictly formalized. Decompilers make it extremely easy to find the parameters of any function call, even if those parameters are initialized far away from the call instruction. With a disassembler, this is a daunting task, which requires handling each case individually.

Decompilers, in contrast with disassemblers, perform extensive **data flow analysis** on the input. This means that questions such as, “Where is the variable initialized?” and, “Is this variable used?” can be answered immediately, without doing any extensive search over the function. Analysts routinely pose and answer these questions, and having the answers immediately increases their productivity.

So if decompilers are so beneficial for binary analysis, why is no decent decompiler available? Two reasons: 1) they are tough to build because decompilation theory is in its infancy; and 2) decompilers have to make many assumptions about the input file, and some of these assumptions may be wrong. Wrong assumptions lead to incorrect output. In order to be practically useful, decompilers must have a means to remove incorrect assumptions and be interactive in general. Building interactive applications is more difficult than building offline (batch) applications. In short, these two obstacles make creating a decompiler a difficult endeavor both in theory and in practice.

Given all the above, we are proud to present our analytical tool, the **Hex-Rays Decompiler**. It embodies almost 10 years of proprieary research and implements many new approaches to the problems discussed above. The highlights of our decompiler are:

- ✔ It can handle real world applications.
- ✔ It has both automatic (batch) and interactive modes.
- ✔ It is compiler-agnostic to the maximum possible degree.
- ✔ Its core does not depend on the processor.
- ✔ It has a type system powerful enough to express any C type.
- ✔ It has been tested on thousands of files including huge applications consisting of tens of Mbs.
- ✔ It is interactive: analysts may change the output, rename the variables and specify their type.
- ✔ It is fast: it decompiles a typical function in under a second.

To learn more about Hex-Rays Decompiler please visit our website http://www.hex-rays.com