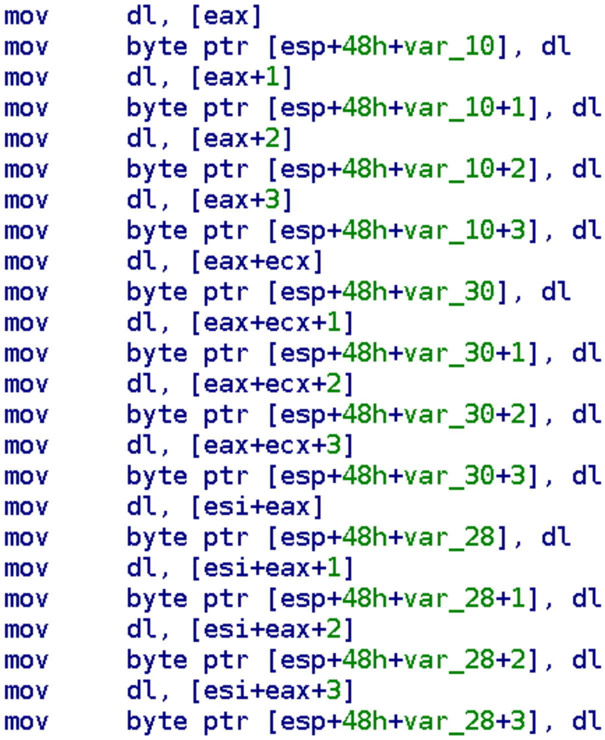
Decompilers and beyond

Ilfak Guilfanov, Hex-Rays SA, 2008

Disassemblers and debuggers are the two tools that allow reverse engineers to examine binary applications. Without them, binary codes are just sequences of hexadecimal numbers. Since humans are notoriously bad with digits, only superficial analysis can be done without these tools.

Basically, the job of a disassembler is very simple: it just maps hexadecimal numbers to instruction mnemonics. The output of such a basic disassembler is a listing with instructions. While this mapping is a big step forward and allows the user to decipher the logic of simple programs, it does not scale well. Analysis of any file bigger than a few kilobytes is problematic because instruction mnemonics are not enough to hold higher level information: labels and comments are needed, as well as facilities to change the representation on the fly.

Better disassemblers give the user commands to add labels and comments to the listing, and allow him to annotate it in other ways. This is another big step forward since it allows the user to extract high level information from the listing and put it back in a labels and comments form. The reverse engineering process is slow and error-prone but a diligent user can produce a highly readable text provided that he spends considerable amount of time on the analyzed file. Unfortunately, the amount of time required for such a feat is prohibitive.



```
mov     dl, [eax]
mov     byte ptr [esp+48h+var_10], dl
mov     dl, [eax+1]
mov     byte ptr [esp+48h+var_10+1], dl
mov     dl, [eax+2]
mov     byte ptr [esp+48h+var_10+2], dl
mov     dl, [eax+3]
mov     byte ptr [esp+48h+var_10+3], dl
mov     dl, [eax+ecx]
mov     byte ptr [esp+48h+var_30], dl
mov     dl, [eax+ecx+1]
mov     byte ptr [esp+48h+var_30+1], dl
mov     dl, [eax+ecx+2]
mov     byte ptr [esp+48h+var_30+2], dl
mov     dl, [eax+ecx+3]
mov     byte ptr [esp+48h+var_30+3], dl
mov     dl, [esi+eax]
mov     byte ptr [esp+48h+var_28], dl
mov     dl, [esi+eax+1]
mov     byte ptr [esp+48h+var_28+1], dl
mov     dl, [esi+eax+2]
mov     byte ptr [esp+48h+var_28+2], dl
mov     dl, [esi+eax+3]
mov     byte ptr [esp+48h+var_28+3], dl
```

Fig. 1: Repetitive assembly text

Another area where a good disassembler differs from a basic one is navigation. The user needs fast methods to find interesting locations in the program, to bookmark them, to keep the navigation history and so on. Ideally, all locations relevant to the current item, like a register or a memory cell, are highlighted or marked otherwise.

There are other facilities a good disassembler should have: a database engine, support for structures, functions, facilities to change the data representation on the fly, etc. IDA Pro, as an example of such an advanced disassembler, has all these features. However, since the output is just a disassembly listing, it has the following insurmountable shortcoming: we continue to work with individual instructions and this makes the conceptual level of the result very low.

In other words, disassembly listings have a very low abstraction level. It is the analyst's job to mentally map assembly instructions to higher level abstractions and concepts. This skill of low to high abstraction mapping is considered as something difficult but after the initially steep learning curve it becomes an easy and even boring task. Maybe it happens so because of highly repetitive patterns in binary files or because of other reasons, but the fact remains: working with disassembly listings is nothing like reading a captivating story. The listings are highly repetitive yet require full concentration because the minutest detail may change the meaning of the code chunk in question or even the entire application. Everyone (except the persons at the beginning of the learning curve) who is exposed to the disassembler output tries to minimize the amount of text he needs to read and decipher.

Working with disassemblers is considered as an task for highly skilled specialists because of the initial learning curve. It is steep enough for the majority to give up and find a more interesting and rewarding activity.

The above-described issues become more acute because softwares grow over time and the amount of hours spent on analysis skyrocket. Meanwhile, we need to analyze applications faster in order to be able to react to proliferating and mutating malware. In some cases, late analysis is worthless because the attack pattern has already changed several times.

The need for a better tool was apparent in the reverse engineering community since long ago. Decompilation is the next logic step, yet a very tough one.

Compilers and decompilers

It is customary to compare compilers and decompilers. The have very similar structures:

Phase	**Compiler**	**Decompiler**
Preprocessing	Performs macro substitution and file inclusion	Parse the input file, its header, section information, relocations, etc.
Lexical analysis	Converts stream of characters into stream of token	Converts stream of bytes into stream of instructions
Syntax analysis	*Builds syntax tree*	*Builds syntax tree*
Code generation	Generates machine code	Generates high level code
Optimization	Modifies the output code to be shorter or faster	Modifies the output code to be shorter or more readable

The comparison is correct but superficial: an inordinate amount of work is comprised in the “syntax analysis” phase. For compilers, this step might include not only syntax analysis but also type analysis and semantic analysis. For decompilers, before we build anything high level like a syntax tree we have to solve lots of problems at the instruction level, perform data flow analysis, etc.

In fact, compilers are in a more privileged position. First of all, the input language is defined strictly. When the input does not conform the language standards, the compiler may simply print an error message and be done with it. Second, the compiler has plenty of information on functions, variables, on their types, etc. This information must be enough to generate valid code. If not, the compiler is allowed to spit out an error message and stop. Finally, the output of a compiler may be ugly and completely unreadable (who will ever read it except a bunch of geeks? ;)

Decompilers do not benefit from anything similar. Just the contrary:

- the input is informal and sometimes deliberately obfuscated
- many decompilation problems are unsolved or proved to be unsolvable in generic case
- the output is examined in details by a human being and any suboptimality is noticed

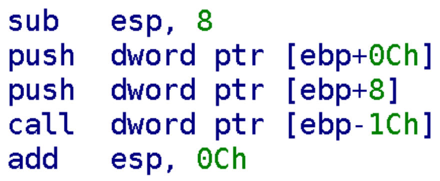
Conclusion: robust machine code decompilers are impossible. If we ever build a decompiler, it will always have some imperfections and eventually generate a wrong output. Our best hope is to diminish the undesired effects as much as possible. Even with the most sophisticated decompiler, we have to stay alert and be ready to help the decompiler by providing additional information. Ideally, the decompiler would issue warnings in suspicious situations and accept the user input to fine tune or correct the result.

Another obstacle to building a working decompiler is the complexity of the decompilation problems. These problems are simple at first sight but all of them turn out to be hard or even unsolvable. Below are some examples:

Function calls

Analysis of a function call requires answers to the following questions:

- Where does the function expect its input registers?
- Where does it return the result?
- What registers or memory cells does it spoil?
- How does it change the stack pointer?
- Does it return to the caller or somewhere else?



```
sub   esp, 8
push  dword ptr [ebp+0Ch]
push  dword ptr [ebp+8]
call  dword ptr [ebp-1Ch]
add   esp, 0Ch
```

Fig. 2: A function call

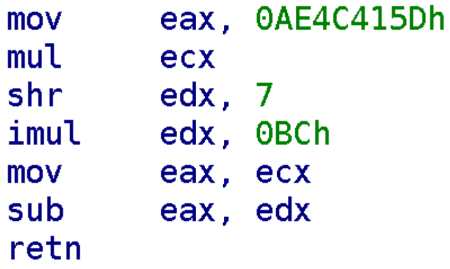
We can not answer these questions with absolute certainty. On Fig. 2, the function pointed by [ebp-1Ch] may accept 2 or more stack arguments. We can not say anything about the input register arguments. The safest bet would be to say that it accepts 3 stack arguments but the third argument is not initialized in this particular snippet. However, this assumption might turn wrong.

Function return values

These questions are really hard to answer:

- Does the function return anything?
- How big is the return value?

On Fig. 3, should EDX be considered as a returned register or not?



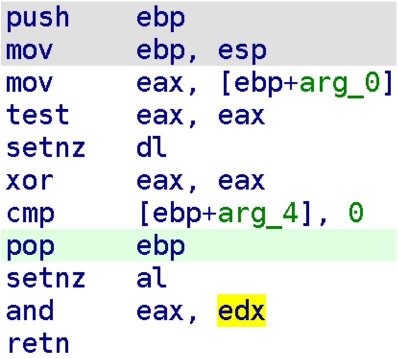
```
mov     eax, 0AE4C415Dh
mul     ecx
shr     edx, 7
imul    edx, 0BCh
mov     eax, ecx
sub     eax, edx
retn
```

Fig. 3: Function return value

Function input arguments

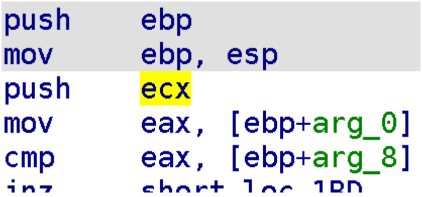
A very naive approach would be to say that if a function accesses a register then it must be considered as an input argument. Counterexamples are easy to give: a function may access a register:

- to save its value
- to allocate stack frame
- to use it as an argument



```
push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]
test    eax, eax
setnz   dl
xor     eax, eax
cmp     [ebp+arg_4], 0
pop     ebp
setnz   al
and     eax, edx
retn
```

Fig. 4: This function does not depend on the value of EDX at its entry point

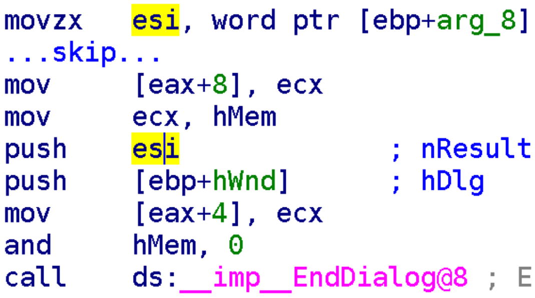


```
push    ebp
mov     ebp, esp
push    ecx
mov     eax, [ebp+arg_0]
cmp     eax, [ebp+arg_8]
jnz     short loc_1BD
```

Fig. 5: The value of ECX is irrelevant. We allocate stack frame here.

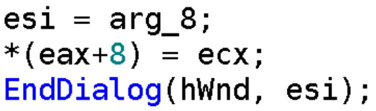
Indirect accesses

Pointers are a big problem because a decompiler has no information on the object boundaries and possible pointer aliases. In the absence of this information, it can only make some assumptions (which may turn wrong). For example, in the Fig. 6, the value of arg_8 is copied to ESI and used as an argument to EndDialog. The decompiler could generate an output like on Fig. 7.



```
movzx   esi, word ptr [ebp+arg_8]
...skip...
mov     [eax+8], ecx
mov     ecx, hMem
push    esi             ; nResult
push    [ebp+hWnd]      ; hDlg
mov     [eax+4], ecx
and     hMem, 0
call    ds:__imp__EndDialog@8 ; E
```

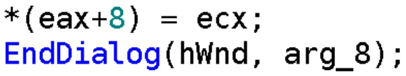
Fig. 7: May we propagate arg_8 to the call?



```
esi = arg_8;
*(eax+8) = ecx;
EndDialog(hWnd, esi);
```

Fig. 6: Acceptable but long result

However, Fig. 8 is nicer:



```
*(eax+8) = ecx;
EndDialog(hWnd, arg_8);
```

Fig. 8: Shorter (better) output

But in order to generate the second output safely, we have to prove that esi and arg_8 do not change their values until the call instruction. For ESI it is quite simple (if we may assume that function calls do not change this register; usually they do not). But for arg_8 it is much more difficult. We have to trace all pointers in the program, which is an impossible task.

Indirect jumps

Indirect jumps are used for switch idioms and tail calls. Recognizing them is necessary to build the control flow graph.

I think I gave enough examples. It is just the top of the iceberg, problems are lurking everywhere:

- Save-restore (push/pop) pairs
- Partial register accesses (al/ah/ax/eax)
- 64-bit arithmetic
- Compiler idioms
- Variable live ranges (for stack variables)
- Lost type information
- Pointers vs. numbers
- Virtual functions
- Recursive functions

While the situation seems hopeless, we can still deal wih some common cases. Even if we handle 80% and speed up the analysis (say) ten times, it is worth trying. Looking back, I can say that yes, it was worth trying because sometimes the speedup factor is not ten but even higher. Recently I got a message from a happy customer saying “it took me 3 hours against 250 hours with a disassembler”, which makes the factor almost 100.

To have some precise idea of what the initial version of the decompiler will and will not do, I decided on the following limitations:

- Compiler generated output (no hostile adversary generating increasingly complex input)
- Only 32-bit code
- No floating point, exception handling and other fancy stuff

Basic ideas

1. Make some configurable assumptions about the input (calling conventions, stack frames, memory model, etc). The user will be able to control the decompiler by specifying the missing information. In simple cases, the decompiler will deduce or guess it.
2. Use sound theoretical approach to solvable problems (data flow analysis on registers, peephole optimization within basic blocks, instruction simplification, etc)
3. Use heuristics for unsolvable problems (indirect jumps, function prolog/epilogs, call arguments)
4. Prefer to generate ugly but correct output rather than nice but incorrect code. Let the user embellish the code by adding information.
5. Let the user guide the decompilation in difficult cases (specify indirect call targets, function prototypes, etc)
6. Interactivity is necessary to achieve good results

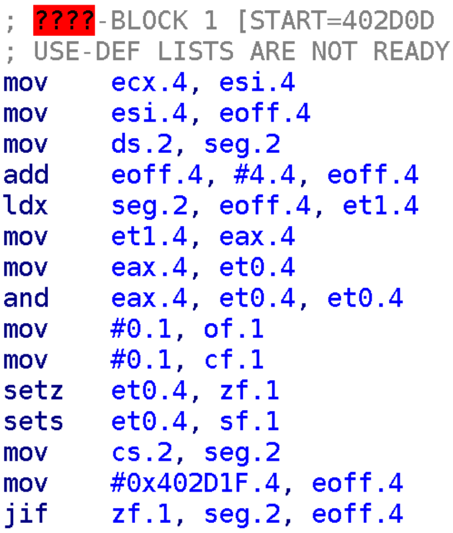
Decompilation phases

I decided to create the following decompilation phases:

Microcode generation	Analyzes the function prolog and epilog, switch idioms, overall verify the function. This phase ensures that the function is well defined. The initial version of the decompiler was rejecting ill defined functions, the current version generates an output for all functions
Local optimization	Simplifies instructions, propagate expressions, determines block types and control graph edges. Strictly speaking, this step should not be considered as a separate phase because it is called from global optimization and local variable allocation steps to improve intermediate results. This step uses standard peephole optimization methods and the methods that can be applied to linear code sequences. Nothing fancy here.
Global optimization	Globally propagates expressions, deletes dead code, resolve memory references, analyzes call instructions, determines input/output registers of the function. Heavily uses data flow analysis. Call instructions and input/output registers use heuristics in addition to data flow. Has some rudimentary pointer analysis but it would be nice to have a better model of the process memory and perform full alias analysis. Currently, the data flow analysis can answer may/must questions. It can trace registers and local stack frame. For registers, it traces every register byte separately. For the stack frame, variable memory granularity is used to be able to handle arbitrarily big stacks.
Local variable allocation	Determines variable live ranges and their sizes, gets rid of all stack and register references, schedules instruction combinations, assigns simple types to all variables. An algorithm used by this phase is worth a separate paper. It can handle 64-bit arithmetic performed on 32bit registers. Variable access patterns are used to determine the best variable boundaries and generate local variable live ranges.
Structural analysis	Analyzes the control flow graph and creates while/if/switch and other constructs. The used algorithm is very robust and can handle irreducible graphs as well. The output of this phase is a control tree, which describes what control constructs will be used in the initial pseudocode. The output has some unnecessary gotos and some constructs forbidden in the C language (e.g. an expression with embedded loop)
Pseudocode generation	Based on the microcode and structural analysis results, generate the output text. Very straightforward and easy.
Pseudocode transformation	Massages the output to make it more readable, create for-loops, remove superfluous gotos, creates break/continue, add/remote casts, etc. This phase is essential for obtaining a readable output.
Type analysis	Analyzes pseudocode, builds type equations and solve them, modifies variable types. I believe that this is the right moment for the type analysis.
Final touch	Renames variables, create va_list etc. Just simple embellishments

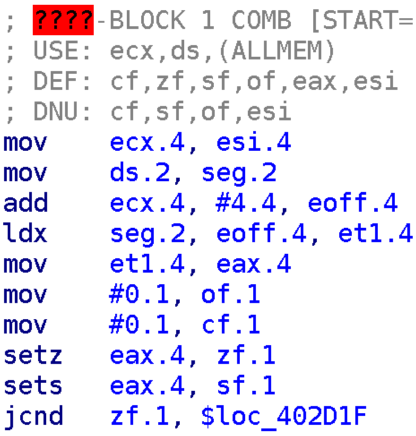
Microcode generation

All assembler instructions are converted to microcode. The microcode language is very detailed and precisely represents how each instruction modifies the memory, registers, and processor condition codes. Typically one CPU instruction is converted into 5-15 microinstructions. Many of these microinstructions are redundant and get destroyed at the preoptimization step (a very simple peephole optimizer). The microcode instructions have zero to 3 operands. The size of each operand is specified individually. Processor condition codes are represented by virtual registers with corresponding names.



```
; ????-BLOCK 1 [START=402D0D
; USE-DEF LISTS ARE NOT READY
mov    ecx.4, esi.4
mov    esi.4, eoff.4
mov    ds.2, seg.2
add    eoff.4, #4.4, eoff.4
ldx    seg.2, eoff.4, et1.4
mov    et1.4, eax.4
mov    eax.4, et0.4
and    eax.4, et0.4, et0.4
mov    #0.1, of.1
mov    #0.1, cf.1
setz   et0.4, zf.1
sets   et0.4, sf.1
mov    cs.2, seg.2
mov    #0x402D1F.4, eoff.4
jif    zf.1, seg.2, eoff.4
```

The preoptimization step is immediately performed and makes the microcode much shorter:

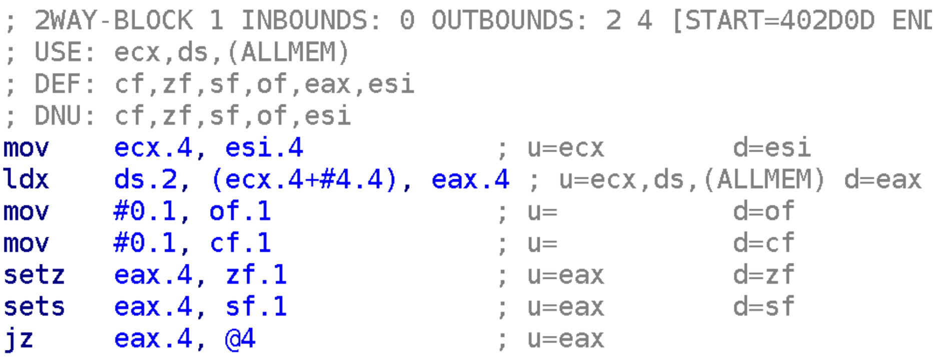


```
; ????-BLOCK 1 COMB [START=
; USE: ecx,ds,(ALLMEM)
; DEF: cf,zf,sf,of,eax,esi
; DNU: cf,sf,of,esi
mov    ecx.4, esi.4
mov    ds.2, seg.2
add    ecx.4, #4.4, eoff.4
ldx    seg.2, eoff.4, et1.4
mov    et1.4, eax.4
mov    #0.1, of.1
mov    #0.1, cf.1
setz   eax.4, zf.1
sets   eax.4, sf.1
jcnd   zf.1, $loc_402D1F
```

The local optimization phase propagates constants and registers within a basic block. It may also propagate a whole instruction into another and replace its operand. All detected dead code gets eliminated. However, the global optimization phase usually destroys more instructions because it has better information on what is used and what is not.

For example, the condition codes (setz/sets instructions) are still present. They will be removed later.

For each instruction, use/def lists are calculated dynamically. We do not store them because it is cheaper to generate them on the fly. However, the use/def lists for the basic block are stored and refreshed when necessary. Parentheses denote “may” access: the **ldx** instruction may modify any memory cell (ALLMEM).



```
; 2WAY-BLOCK 1 INBOUNDS: 0 OUTBOUNDS: 2 4 [START=402D0D ENI
; USE: ecx,ds,(ALLMEM)
; DEF: cf,zf,sf,of,eax,esi
; DNU: cf,zf,sf,of,esi
mov    ecx.4, esi.4             ; u=ecx          d=esi
ldx    ds.2, (ecx.4+#4.4), eax.4 ; u=ecx,ds,(ALLMEM) d=eax
mov    #0.1, of.1               ; u=             d=of
mov    #0.1, cf.1               ; u=             d=cf
setz   eax.4, zf.1              ; u=eax          d=zf
sets   eax.4, sf.1              ; u=eax          d=sf
jz     eax.4, @4                ; u=eax
```

Global optimization

After the global optimization phase, we can observe the following changes:

- Condition codes are gone (because they were not used by any instruction).
- The **ldx** instruction got propagated to jz and all references to **eax** are gone. In fact, instructions may get propagated into other instructions and form really huge expressions. There are some heuristic rules in the decompiler to avoid this. Without them, encryption and hash calculation functions were just dreadful. They are still far from being easily readable but the output is manageable.
- Note that the target of **jz** has changed to **@3** (it is a basic block number) since the global optimization has removed some unused code and blocks.
- We are ready for local variable allocation.

```
; 2WAY-BLOCK 1 INBOUNDS: 0 OUTBOUNDS: 2 3 [START=402D0D
; USE: ecx,ds,(ALLMEM)
; DEF: esi
; DNU: esi
mov    ecx.4, esi.4             ; u=ecx        d=esi
jz     [ds.2:(ecx.4+#4.4)].4, @3 ; u=ecx,ds,(ALLMEM)
```

Local variable allocation

This phase uses the data flow analysis to connect registers from different basic blocks in order to convert them into local variables. If a register is defined by a block and used by another, then we will create a local variable covering both the definition and the use. In other words, a local variable consists of all definitions and all uses that can be connected together. While the basic idea is simple, things get complicated because of byte/word/dword registers.

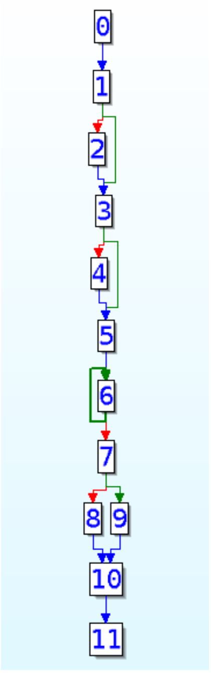
Some registers (defined and immediately used within a basic block) are not visible to the data flow analyzer. For them, we arrange an extra pass and create new local variables. Since everything is within a basic block, the algorithm is very simple.

For the time being, we do not analyze live ranges of stack variables (this requires first a good alias analysis: we have to be able to prove that a stack variable is not modified between two locations). I doubt that a full fledged live range analysis will be available for stack variables in the near future.

After this phase we have a nice microcode. We do not modify it anymore and turn our attention to the control graph and structural analysis.

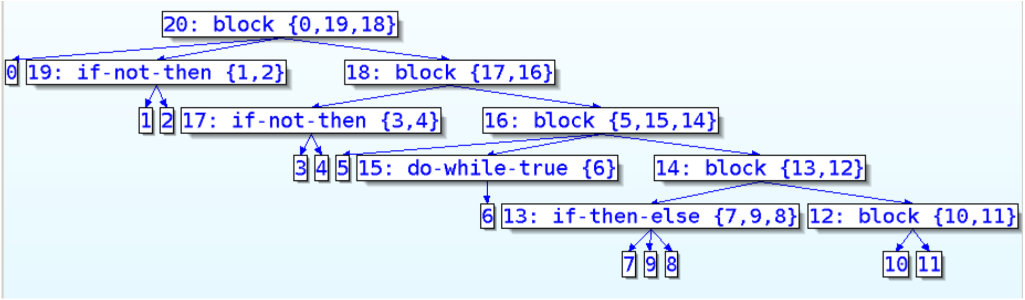
```
; 2WAY-BLOCK 1 INBOUNDS: 0 OUTBOUNDS: 2 3 [START=402D0D
; USE: ecx,ds,(ALLMEM)
; DEF: esi
; DNU: esi
mov    ecx0.4, esi1.4            ; u=
jz     [ds.2:(ecx0.4+#4.4)].4, @3 ; u=ds,(ALLMEM)
```

Structural analysis



This phase extracts control structures from the CFG (control flow graph) and represents them as a tree. The structural analysis algorithm is robust and can handle any graphs, including irreducible ones. If it encounters an irreducible graph, it removes an edge and adds a *goto* node to the control tree. The choice of the edge to remove remains a black art; currently the decompiler tries to remove the edges whose destination node has multiple incoming edges but if it fails do to so, it removes just any edge (given that the graph stays connected). The structural analysis algorithm has been tested on random graphs – I let my computer running for days and nights in the self test mode.

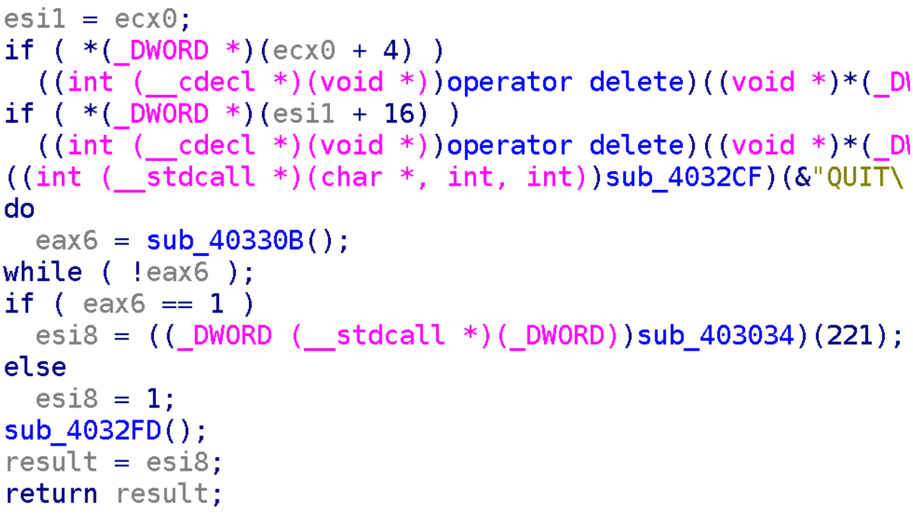
Below is the control tree. It will be used for generating the initial pseudocode.



Initial pseudocode

The generated pseudocode is almost unreadable. The decompiler generates numerous cast operators because the variable types are not correct yet, and it over cautiously generates casts when needed or not. Later most of these casts will go away. For example, if **ecx0** becomes a pointer to an integer, the first cast operator will disappear.

Please also note that the **return** statement could be combined with the previous assignment. This is a task for the next phase.



```
esi1 = ecx0;
if ( *(_DWORD *)(ecx0 + 4) )
  ((int (__cdecl *)(void *))operator delete)((void *)*(_D
if ( *(_DWORD *)(esi1 + 16) )
  ((int (__cdecl *)(void *))operator delete)((void *)*(_D
((int (__stdcall *)(char *, int, int))sub_4032CF)(&"QUIT\
do
  eax6 = sub_40330B();
while ( !eax6 );
if ( eax6 == 1 )
  esi8 = ((_DWORD (__stdcall *)(_DWORD))sub_403034)(221);
else
  esi8 = 1;
sub_4032FD();
result = esi8;
return result;
```

Pseudocode transformations

The decompiler has over 50 pseudocode transformation rules. Without them, the output would be much inferior to what we have. Below is an example of such a rule:

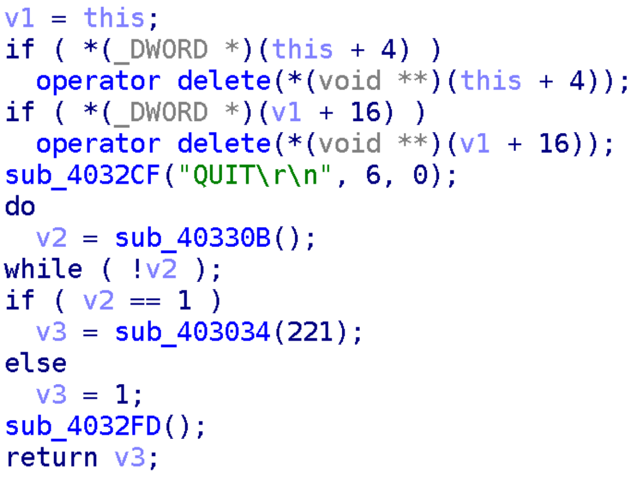
```
IF-8              CONDITIONS: x is a comma operator
----

if ( (x1, x2) || y )       x1
  z;                       if ( x2 || y )
                             z;

|| can be replaced by &&
```

Moving x1 out of the **if** condition makes the output more readable. In fact, these rules are quite easy to write and the decompiler SDK allows any user to create a plugin for additional improvements.

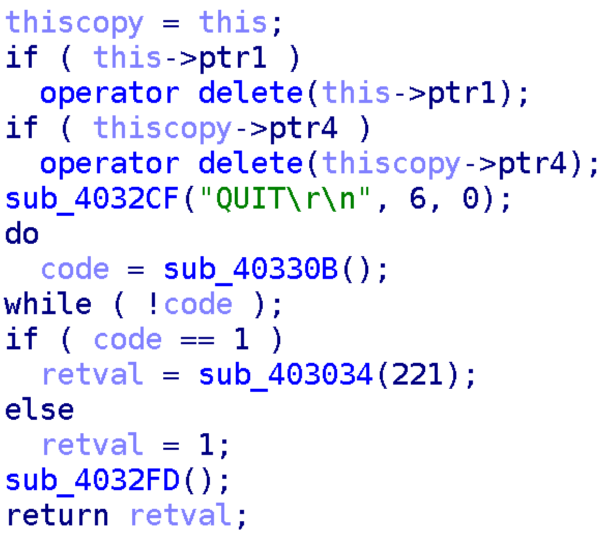
Here is our code after the transformations. Please note that variables have better names and many types were guessed correctly. But there are some remaining problems and the user can fix them interactively.



```
v1 = this;
if ( *(_DWORD *)(this + 4) )
  operator delete(*(void **)(this + 4));
if ( *(_DWORD *)(v1 + 16) )
  operator delete(*(void **)(v1 + 16));
sub_4032CF("QUIT\r\n", 6, 0);
do
  v2 = sub_40330B();
while ( !v2 );
if ( v2 == 1 )
  v3 = sub_403034(221);
else
  v3 = 1;
sub_4032FD();
return v3;
```

Interactive operation

Here we renamed some variables and changed their types. The **this** variable is now a pointer. We could also rename the called functions, add comments, and replace 221 with something more meaningful.



```
thiscopy = this;
if ( this->ptr1 )
  operator delete(this->ptr1);
if ( thiscopy->ptr4 )
  operator delete(thiscopy->ptr4);
sub_4032CF("QUIT\r\n", 6, 0);
do
  code = sub_40330B();
while ( !code );
if ( code == 1 )
  retval = sub_403034(221);
else
  retval = 1;
sub_4032FD();
return retval;
```

Beyond decompilers

A robust decompiler is a great time saver. It also allows the user to focus on the interesting stuff and diminishes distraction and user mistakes, all this is obvious. What other new things and horizons does it open? As we move away from low instruction level to higher abstractions, there must be something more than simply time saving.

The decompiler can (and should) be considered as a platform to build other things on. For that, we need an API. It already exists but microcode is not accessible yet. We will disclose its format later – before that we need to port it to at least one more platform and possibly add floating point support. I suspect that these additions may lead to some modifications in the microcode (floating point arithmetic will require floating point operand types and instructions; retargeting the decompiler to a new processor will reveal all places that are too x86-centric).

Examining and understanding binary code was the primary goal of the decompiler. I was thinking of the next steps:

- program verification
- better disassembler
- recovery of application domain
- more abstract representations
- binary code comparison

Program verification

Decompilation based vulnerability scanner, is anyone interested? ;) Well, we have to keep in mind that the decompiler is not a precise tool and that it will eventually generate a wrong output. However, some bug classes can be detected quite easily:

- Missing return value validations (e.g. for NULL pointers)
- Missing input value validations
- Taint analysis
- Insecure code patterns
- Uninitialized variables
- etc..

I don't claim that this is something easy but this is certainly feasible and will bring some practical results.

Better disassembler

The idea is simple: use the decompilation results to improve the disassembly listing. Even the most hardcore assembler fans will appreciate an unintrusive hints, for example hovering the mouse over a register or data could display:

- its possible values or value ranges
- locations where is is defined
- locations where it is used

There are many possible improvements there...

Recovery of application domain

Another possibility would be a tool to recover types from the application domain. It is about leveraging from **DWORD** (assembler) to **signed int** (standard C/C++) and then to even higher **customer_id_t** (application defined type). We need to be able to recover scalar and non-scalar (structure) types. This is not a new direction in itself but another step forward. Type recovery will make possible more analyses. One of the biggest benefits is that we will have more information on the object boundaries and this will be extremely useful.

More abstractions

Again, just some ideas: a tool to cluster functions into modules or libraries would be useful. This can be done using the type information (functions dealing with the same type can be grouped together).

Global data flow analysis can be used for better taint analysis. Questions like
“is there a possible execution path between two program locations?” or
“what locations can the given value reach?” require building global data flow equations.

Statistical analysis of pseudocode may give some insight.

To move to the next level, inline functions and templates could be detected.

Binary code comparison

You know better than me the possible applications...

- To find code plagiarisms
- To detect changes between program versions
- To find library functions (high-gear FLIRT)

Since you are the practitioners and we are mere developers, your ideas are welcome!