dewolf - Survey Results

This document contains the full questionnaires and results of the three conducted user surveys.

I. Survey 1 - Complete Questionnaire and Results

A. Baseline Questions:

- What is your highest level of education?

Answer	Count	%
School Graduation	1	2.7
Bachelor	22	59.5
Master	12	32.4
PhD	2	5.4

- How often do you reverse malware?

Answer	Count	%
Monthly	5	13.5
Weekly	2	5.4
Daily	1	2.7
Less	29	78.4

- For how long are you reversing malware?

Answer	Count	%
>5 years	5	13.5
3 years	1	2.7
2 years	3	8.1
<1 years	28	75.7

- How much experience do you have in .. ?

Reversing	1	2	3	4	5	6	7	8	9	10
Count	4	8	8	3	5	3	2	2	1	1
%	10.8	21.6	21.6	8.1	13.5	8.1	5.4	5.4	2.7	2.7

Malware Analysis	1	2	3	4	5	6	7	8	9	10
Count	14	8	3	1	3	3	3	1	0	1
%	37.8	21.6	8.1	2.7	8.1	8.1	8.1	2.7	0	2.7

Writing C-code	1	2	3	4	5	6	7	8	9	10
Count	1	1	7	7	4	4	3	4	5	1
%	2.7	2.7	18.9	18.9	10.8	10.8	8.1	10.8	13.5	2.7

- Which binary code decompiler have you used before? (multiple answers are possible)

Answer	Count	%
Hex-Rays	25	67.6
Ghidra	21	56.8
BinaryNinja HLIL	3	8.1
Boomerang	0	0
REC	3	8.1
DISC	0	0
Other	6	16.2

Other: JD-GUI, nil, ida, none, OLLYDEBUG, IDA

B. Part A

Part A Our decompiler

- Please consider the following decompiled function and answer the questions below. Feel free to use the editor as you would normally do (e.g. by renaming variables).

```
unsigned long A(int arg1, int arg2) {
  unsigned long var_0;

  if (arg1 == 0) {
    var_0 = arg2;
  }
  else {
    var_0 = arg2;
    while(true) {
      arg2 = (arg1 + -0x1) & 0xffffffff;
      if ((unsigned int)var_0 == 0) {
        if (arg2 == 0) {
          var_0 = 0x1;
          break;
        }
        arg1 = arg2;
        var_0 = 0x1;
        continue;
      }
      var_0 = A(arg1, ((unsigned int) var_0) + 0xffffffff);
      if (arg2 == 0) {
        break;
      }
      arg1 = arg2;
    }
  }
  return ((unsigned int) var_0) + 0x1;
}
```

15 participants count %	strongly disagree	disagree	weakly disagree	unsure	weakly agree	agree	strongly agree
The used control-flow structures are appropriate.	1 6.7	1 6.7	1 6.7	3 20.0	3 20.0	5 33.3	1 6.7
The control-flow is strangely restructured.	0 0.0	4 26.7	2 13.3	2 13.3	3 20.0	2 13.3	2 13.3
It was easy to understand the code.	1 6.7	1 6.7	1 6.7	3 20.0	3 20.0	5 33.3	1 6.7
It took much effort to understand the code.	0 0.0	1 6.7	3 20.0	2 13.3	4 26.7	3 20.0	2 13.3
It seems that there are no unused instructions.	0 0.0	2 13.3	1 6.7	1 6.7	3 20.0	5 33.3	3 20.0
There are too much unused instructions.	3 20.0	1 6.7	8 53.3	0 0.0	1 6.7	1 6.7	1 6.7
I think the code is correctly decompiled.	0 0.0	1 6.7	0 0.0	8 53.3	2 13.3	4 26.7	0 0.0
The decompiled code seems to be incorrect.	0 0.0	3 20.0	3 20.0	7 46.7	0 0.0	1 6.7	1 6.7
The conditions are too complex.	2 13.3	8 53.3	1 6.7	0 0.0	3 20.0	1 6.7	0 0.0
The code contains too many intermediate results.	0 0.0	3 20.0	3 20.0	4 26.7	3 20.0	2 13.3	0 0.0
The line length is too long	8 53.3	6 40.0	1 6.7	0 0.0	0 0.0	0 0.0	0 0.0
Many variables have useless copies.	1 6.7	4 26.7	5 33.3	1 6.7	2 13.3	1 6.7	1 6.7
There are no useless copies of variables.	1 6.7	1 6.7	3 20.0	0 0.0	4 26.7	5 33.3	1 6.7
The variable types seem to be reasonable.	0 0.0	1 6.7	4 26.7	1 6.7	4 26.7	5 33.3	0 0.0
Some variable types seem to be wrong.	0 0.0	3 20.0	3 20.0	3 20.0	4 26.7	2 13.3	0 0.0
There are no variables that only store unnecessary intermediate constants.	2 13.3	2 13.3	0 0.0	1 6.7	4 26.7	5 33.3	1 6.7
There are too many variables that just store intermediate constants.	0 0.0	4 26.7	4 26.7	2 13.3	5 33.3	0 0.0	0 0.0

- What is the output of the function for the parameters

	0	1	2	3	4	5	I do not know
arg_1 = 0 and arg_2 = 1	0 0.0	0 0.0	14 93.3	0 0.0	0 0.0	0 0.0	1 6.7
arg_1 = 1 and arg_2 = 0	0 0.0	0 0.0	13 86.7	0 0.0	0 0.0	0 0.0	2 13.3

- Which computation needs more recursive calls?

Answer	Count	%
arg_1 = 1 and arg_2 = 2	4	26.7
arg_1 = 2 and arg_2 = 1	11	73.3

- What do you like or dislike about the above sample? (optional free text)

I'm not sure what the function was intendend to do, hence, I also don't know if it is decompiled correctly. Also, I'm not sure whether the code could have been simplified. I think so, but I would have to spend more time to think about it. Still, the code was easy and understandable enough to follow it using pen and paper. What I didn't like (major issue): the "+ 0xffffffff" and "& 0xffffffff" parts. I'm not sure if the 0xffffffff in line 9 does anything and it took me really long to figure out that "+ 0xffffffff" is -1 (you got me here). Especially the 0xffffffff in line 21 is totally confusing and should definitely be -1. What I didn't like (minor issue): I'm not a fan of casts cluttering the code. I removed them in the editor to not be distracted anymore. I'm not sure if there are corner cases, but do we really need them in this function? What I didn't like (even less important): I would have preferred an early return in line 5. (Why do I have to answer the each question twice (pos./neg. form)?)

Recursion. Bah.

var_0 seems reused for a lot of things

untidy while loop

Simple computations

no comments

Too many if statement. Not easy to understand what the code trying to do

types are not normalized in stmt

1.) this is clearly obfuscation and written to make it most hard for the reader/reverser in all aspects! 2.) Also, the *author* might *not* have considered that this code would produce quite different result on different OS, such as Windows, to name an important OS. ;-) On Windows, an signed/unsigned int == 64 bit, on a 64 bit OS. Thus, it would NOT lead to overflow with 1 + 0xffffffff. We did assume a Linux system here, where a signed/unsigned int == 32 bit register size, leading to overflow with 1 + 0xffffffff.

There is no recursion

Part A IDA

- Please consider the following decompiled function and answer the questions below. Feel free to use the editor as you would normally do (e.g. by renaming variables).

```
1  __int64 __fastcall A(__int64 arg_1, int arg_2){
2    unsigned int var_0;
3
4    if ( (_DWORD)arg_1 ) {
5      do{
6        while ( 1 ) {
7          var_0 = arg_1 - 1;
8          if ( !arg_2 ) {
9            break;
10         }
```

```
        arg_2 = A(arg_1, (unsigned int)(arg_2 -
↪  1));
        arg_1 = var_0;
        if ( !var_0 ) {
          return (unsigned int)(arg_2 + 1);
        }
      }
      arg_2 = 1;
      arg_1 = var_0;
    }while ( var_0 );
  }
  return (unsigned int)(arg_2 + 1);
}
```

13 participants count %	strongly disagree	disagree	weakly disagree	unsure	weakly agree	agree	strongly agree
The used control-flow structures are appropriate.	1 7.7	0 0.0	2 15.4	6 46.2	2 15.4	2 15.4	0 0.0
The control-flow is strangely restructured.	0 0.0	1 7.7	3 23.1	3 23.1	4 30.8	1 7.7	1 7.7
It was easy to understand the code.	0 0.0	3 23.1	3 23.1	0 0.0	5 38.5	2 15.4	0 0.0
It took much effort to understand the code.	0 0.0	0 0.0	3 23.1	2 15.4	5 38.5	3 23.1	0 0.0
It seems that there are no unused instructions.	0 0.0	0 0.0	1 7.7	5 38.5	3 23.1	2 15.4	2 15.4
There are too much unused instructions.	1 7.7	3 23.1	6 46.2	2 15.4	1 7.7	0 0.0	0 0.0
I think the code is correctly decompiled.	0 0.0	0 0.0	1 7.7	5 38.5	5 38.5	2 15.4	0 0.0
The decompiled code seems to be incorrect.	0 0.0	3 23.1	6 46.2	4 30.8	0 0.0	0 0.0	0 0.0
The conditions are too complex.	4 30.8	2 15.4	4 30.8	1 7.7	1 7.7	1 7.7	0 0.0
The code contains too many intermediate results.	0 0.0	4 30.8	5 38.5	1 7.7	2 15.4	1 7.7	0 0.0
The line length is too long	4 30.8	5 38.5	4 30.8	0 0.0	0 0.0	0 0.0	0 0.0
Many variables have useless copies.	0 0.0	4 30.8	3 23.1	2 15.4	4 30.8	0 0.0	0 0.0
There are no useless copies of variables.	0 0.0	3 23.1	5 38.5	2 15.4	2 15.4	1 7.7	0 0.0
The variable types seem to be reasonable.	0 0.0	1 7.7	2 15.4	4 30.8	4 30.8	2 15.4	0 0.0
Some variable types seem to be wrong.	1 7.7	1 7.7	3 23.1	4 30.8	1 7.7	3 23.1	0 0.0
There are no variables that only store unnecessary intermediate constants.	2 15.4	2 15.4	3 23.1	2 15.4	2 15.4	2 15.4	0 0.0
There are too many variables that just store intermediate constants.	0 0.0	3 23.1	3 23.1	4 30.8	3 23.1	0 0.0	0 0.0

- What is the output of the function for the parameters

	0	1	2	3	4	5	I do not know
arg_1 = 0 and arg_2 = 1	0 0.0	0 0.0	11 84.6	2 15.4	0 0.0	0 0.0	0 0.0
arg_1 = 1 and arg_2 = 0	0 0.0	4 30.8	6 46.2	1 7.7	0 0.0	0 0.0	2 15.4

- Which computation needs more recursive calls?

Answer	Count	%
arg_1 = 1 and arg_2 = 2	5	38.5
arg_1 = 2 and arg_2 = 1	8	61.5

- What do you like or dislike about the above sample?

  var_0 seems redundant arg_2 seems to have type unsigned_int and not int

  Recursive calls

  Like : It compiles, and therefore is easier to test dynamically for programmers. Doesn't use gotos. Seems to be no dead code. Dislike : It is still hard to decipher what the function is attempting to compute.

  from my observations, recursion appears rarely in real-world code.

  Inconsistent datatypes. Mixing windows typedefs (DWORD) with default c types (unsigned int) and pseudo types (__int64) feels unclean. Personally I'd prefer only using the types from cstd.h (uint8_t, int64_t, ...), though int8_t vs. char is debatable.

Part A Ghidra

- Please consider the following decompiled function and answer the questions below. Feel free to use the editor as you would normally do (e.g. by renaming variables).

```
ulong A(ulong arg_1,ulong arg_2){
  int var_0;
  uint var_1;

  arg_2 = arg_2 & 0xffffffff;
  var_0 = (int)arg_2;
  var_1 = (uint)arg_1;
  while (var_1 != 0) {
    while( true ) {
      var_1 = (int)arg_1 - 1;
      if ((int)arg_2 != 0) {
        break;
      }
      arg_2 = 1;
      var_0 = 1;
      arg_1 = (ulong)var_1;
      if (var_1 == 0) {
        goto Label_1;
      }
    }
    arg_2 = A(arg_1,(ulong)((int)arg_2 -
↪  1));
    var_0 = (int)arg_2;
    arg_1 = (ulong)var_1;
```

```
  }
Label_1:
  return (ulong)(var_0 + 1);
}
```

9 participants count %	strongly disagree	disagree	weakly disagree	unsure	weakly agree	agree	strongly agree
The used control-flow structures are appropriate.	0 0.0	0 0.0	0 0.0	7 77.8	2 22.2	0 0.0	0 0.0
The control-flow is strangely restructured.	0 0.0	1 11.1	2 22.2	2 22.2	2 22.2	2 22.2	0 0.0
It was easy to understand the code.	1 11.1	1 11.1	5 55.6	0 0.0	2 22.2	0 0.0	0 0.0
It took much effort to understand the code.	0 0.0	0 0.0	1 11.1	1 11.1	3 33.3	4 44.4	0 0.0
It seems that there are no unused instructions.	0 0.0	1 11.1	1 11.1	2 22.2	2 22.2	2 22.2	1 11.1
There are too much unused instructions.	1 11.1	1 11.1	2 22.2	3 33.3	0 0.0	2 22.2	0 0.0
I think the code is correctly decompiled.	0 0.0	1 11.1	0 0.0	2 22.2	5 55.6	1 11.1	0 0.0
The decompiled code seems to be incorrect.	0 0.0	0 0.0	3 33.3	5 55.6	1 11.1	0 0.0	0 0.0
The conditions are too complex.	0 0.0	5 55.6	1 11.1	2 22.2	1 11.1	0 0.0	0 0.0
The code contains too many intermediate results.	1 11.1	1 11.1	1 11.1	3 33.3	2 22.2	1 11.1	0 0.0
The line length is too long	4 44.4	2 22.2	3 33.3	0 0.0	0 0.0	0 0.0	0 0.0
Many variables have useless copies.	1 11.1	2 22.2	2 22.2	1 11.1	2 22.2	0 0.0	1 11.1
There are no useless copies of variables.	1 11.1	2 22.2	1 11.1	1 11.1	2 22.2	1 11.1	1 11.1
The variable types seem to be reasonable.	0 0.0	0 0.0	0 0.0	1 11.1	5 55.6	3 33.3	0 0.0
Some variable types seem to be wrong.	0 0.0	3 33.3	0 0.0	4 44.4	1 11.1	1 11.1	0 0.0
There are no variables that only store unnecessary intermediate constants.	1 11.1	0 0.0	2 22.2	1 11.1	2 22.2	2 22.2	1 11.1
There are too many variables that just store intermediate constants.	1 11.1	2 22.2	0 0.0	0 0.0	3 33.3	3 33.3	0 0.0

- What is the output of the function for the parameters

	0	1	2	3	4	5	I do not know
arg_1 = 0 and arg_2 = 1	0 0.0	0 0.0	7 77.8	0 0.0	1 11.1	0 0.0	1 11.1
arg_1 = 1 and arg_2 = 0	1 11.1	0 0.0	5 55.6	0 0.0	0 0.0	0 0.0	3 33.3

- Which computation needs more recursive calls?

Answer	Count	%
arg_1 = 1 and arg_2 = 2	3	33.3
arg_1 = 2 and arg_2 = 1	6	66.7

- What do you like or dislike about the above sample?

  the variables name are too similar and have to manually change them, if not very confusing.

  dislike: too many type casts

  Dislike: var_1 = arg_1 and var_2 = arg_2 and vice versa appears to only add to the confusion.

  Dislike: arguments are modified, making it hard to keep track of their values Like: Short statements

C. Part B

Part B - Q1

- Please consider the following three samples and order them from your favourite to least favourite:
  Sample 1 (our):

```
int test1() {
  int var_0;

  var_0 = rand();
  rand();
  switch(var_0) {
  case 1:
    var_0 = 0;
    break;
  case 5:
    var_0 = var_0 + 1;
  case 10:
    var_0 = var_0 << 1;
    break;
  }
  return var_0;
}
```

  Sample 2 (Ghidra):

```
int test1(void) {
  int var_0;

  var_0 = rand();
  rand();
  if (var_0 != 10) {
    if (10 < var_0) {
      return var_0;
    }
    if (var_0 == 1) {
      return 0;
    }
    if (var_0 != 5) {
      return var_0;
    }
    var_0 = 6;
  }
  return var_0 << 1;
}
```

Sample 3 (IDA):

```
1  int test1() {
2    int var_0;
3
4    var_0 = rand();
5    rand();
6    if ( var_0 == 10 ) {
7      goto Label_1;
8    }
9    if ( var_0 <= 10 ) {
10     if ( var_0 == 1 ) {
11       return 0;
12     }
13     if ( var_0 == 5 ) {
14       var_0 = 6;
15   Label_1:
16       var_0 *= 2;
17       return var_0;
18     }
19   }
20   return var_0;
21 }
```

Count	Ranks		
%	1	2	3
Sample 1	28 75.7	4 10.8	5 13.5
Sample 2	6 16.2	26 70.3	5 13.5
Sample 3	3 8.1	7 18.9	27 73.0

- Do you prefer switch or if-else when there are at most 2 or 3 cases?

Answer	Count	%
switch	23	62.2
if-else	14	37.8

Actually, I don't care in this case, but when writing code, I'd probably use if-else. So, this would be the natural choice here.

On fall-through cases (like in the example), I prefer switch. Else I prefer if-else on at most 3 cases.

Depends. If every switch with code has a break, it's fine, but not like this.

But that also depends on the complexity of the conditions.

This depends on the actual cases. For ranged cases, I prefer if-else. For a fixed set of cases (and an "else" / default case), I prefer switch.

Easier to analyse the control flow since we don't have to evaluate the conditions

Switch is easier to understand and follow the program flow.

no real pref., although switch is more readable

depends on complexity. if-else for simple ones, please.

in the given example, switch is more straight forward to read (even with the fall-through).

Dependent on the variable-type and if ranges are involved in the cases.

I usually prefer if-else for 2 or 3 cases, but do not prefer many nested ifs like in the sample above.

- If the value of a variable is known due to a condition, i.e., Branch or Switch, should it be propagated?

Answer	Count	%
yes, the value should be propagated	25	67.6
no, the value should not be propagated	12	32.4

The variable name sometimes hold semantic information that would get lost if only the value gets propagated (like in sample 2, line 11) Even the used operation (like adding one) can contain useful meaning, even if the result is known (to be 6 like in the samples). Best case would be writing known values as comments behind statements where the value is known.

Without a hint, it's confusing. "Why are we setting the variable to 6 here?"

While this should be done in general, in some cases it would be helpful to leave the conditions as they are, to understand the underlying structure. Otherwise there would pop up some random numbers and pruned conditions and it is not clear what they mean.

This obscures future changes in the code as well as the intention of the assigned value, e.g., in line 16, the value 6 cannot be reconstructed as var_0 + 1 but appears as a constant.

It depends. Best would be per default no propagation but the end user should decide

Sometimes we can understand the semantics better when it is known how values are calculated

Operations should be preserved, e.g. the incrementing action in "var_0 + 1" may be more meaningful than the value being 6.

- What did you like most about your favourite sample?

It look very clean and concise. Except for the missing break in case 5 (missing breaks are always confusing), it's possible to immediately see what's going on.

Sample 1 - very easy to follow the control flow

no goto, no code in switch statements without break

clear structure

Clean

easy to understand the control flow conditions

simple and can be easily understood with single look

No gotos, cases are cleary distinguishable (no fall-through), return as soon as the result is known.

No

looks clean

Easiest to understand

Easy to analyse control flow since we just have to compare values with the cases in the switch

easy to read, but ties with switch (sample 1)

clear presentation of codes

Fewer condition checking and code is easy to follow

Easier to understand

Using case easy to follow and understand especially when the code is long

clear conditions to determine var0

Easy to read

neat

Cleaner & easier to understand

the complete switch-case.

clear structure

There were no lines where more than one condition had to be kept in mind.

the switch allows to read the code more or less linearly without the need to potentially re-read above conditions again.

More instantaneous overview of the cases for which additional instructions will be applied.

they are not very different actually, its a matter of syntax familiarity

Sample 3 is easier to interprete

if-else cases

switch

Clean hierarchy of codes

Very flat (not much nesting), no gotos

Switch-Case is super clean

- What did you dislike about your least favourite sample?

I think both, sample 2 and 3, are very cluttered. For sample 3: ifs combined with gotos? Please, no! However, I like that sample 3 has *2 instead of shifts.

Sample 2 - mixing of == and != as if-else conditions thus one has to be alert to not mix them up when following the control flow.

goto

deeper nested structure and gotos

Too complicated

messy, use of gotos makes code more difficult to follow, conditions are harder to understand

need longer time to understand the code

Goto, even worse than a fall-through.

the return statement is only inoked in the end

too many lines

prefer if else to switch

Messy control flow due to goto instruction

goto

too many nested statements

Code is hardest to follow

labels

The use of "!=" can mean there are a lot of possibilities in the variable as compared to the use of "=="

Harder to read

messy

goto label could potentially create some confusion in a bigger sample

lengthy unreadable code

goto and labels

The use of go-tos.

the negative condition in the first if feels less natural than in the other one (sample 3).

Too many if-indent-blocks, the goto instruction; it made the overall control-flow less obvious.

they are not very different actually, its a matter of syntax familiarity

Label

GOTO!

Labels add to the confusion.

Has nested ifs and has a goto

Its long and confusing. One has to keep to many things in mind

---

Part B - Q2

- Please consider the following three samples and order them from your favourite to least favourite:
  Sample 1 (Ghidra):

```
undefined4 test2(void) {
  int var_0 [3];

  printf("Enter any number: ");
  __isoc99_scanf(0x0804a01f,var_0);
  if (var_0[0] < 1) {
    if (var_0[0] < 0) {
      if (var_0[0] < 0) {
        printf("%d is negative.",var_0[0]);
      }
```

```
    }
    else {
      printf("%d is zero.",var_0[0]);
    }
  }
  else {
    if (var_0[0] >= 1) {
      printf("%d is positive.",var_0[0]);
    }
  }
  return 0;
}
```

Sample 2 (our):

```
int test2() {
  int * var_0;
  int * var_1;

  printf("Enter any number: ");
  var_1 = &(var_0);
  __isoc99_scanf(0x804a01f, var_1);
  switch((unsigned int) (byte) (var_0 &
↪  0xffffff00) || (var_0 > 0)) {
    case 0:
      switch((unsigned int) (unsigned byte)
↪  var_0 u>> 31) {
        case 0:
          printf("%d is zero.", var_0);
          break;
        case 1:
          printf("%d is negative.", var_0);
          break;
        }
        break;
    case 1:
      printf("%d is positive.", var_0);
      break;
    }
  return 0;
}
```

Sample 3 (IDA):

```
int test2() {
  int var_0[3];

  printf("Enter any number: ");
  __isoc99_scanf("%d", var_0);
  if ( var_0[0] <= 0 ) {
    if ( var_0[0] >= 0 ) {
      printf("%d is zero.", var_0[0]);
    }
    else if ( var_0[0] < 0 ) {
      printf("%d is negative.", var_0[0]);
    }
  }
  else if ( var_0[0] > 0 ) {
    printf("%d is positive.", var_0[0]);
  }
  return 0;
}
```

Count	**Ranks**		
%	1	2	3
Sample 1	3 8.1	28 75.7	6 16.2
Sample 2	4 10.8	3 8.1	30 81.1
Sample 3	30 81.1	6 16.2	1 2.7

- Do you prefer switch or if-else in a switch-case?

Answer	Count	%
switch	6	16.2
if-else	16	43.2
if-else if switch has at most 2 cases and switch otherwise	15	40.5

Actually, none of the above. In my opinion, the cases of a switch stmt. should be rather short and not contain any other control flow constructs.

depends on the context

Depends on the situation (ranges vs fixed numbers). Here, if-else seems to be advantageous.

If-else is also good if there are not too many if-else inside a if-else

if conditions are simple switch conditions, switch is alright, however if conditions are complex, i am more used to ”parsing” if-else structure as you will need to mentally interpret the switch as if-else.

Actually, if there is a 4th choice where there are only 2 cases, but the switch condition is complex but actually only evaluates to either true or false (like in sample 2), then if-else is better.

for comparison of value ranges (0), opposite to explicit values like in the previous example, switch feels less natural and harder to comprehend (bitmask/shift magic is also harder to understand for beginners).

I tend to prefer if-else for ranges, except if the boundaries are few, e.g., 2, and the switch can be well-adapted to these boundaries.

As before, in general I prefer switch, but in this case it cannot properly be represented as switch. Representing it as has been done here is ridiculous. Don’t do switch, if *in*equalities are involved.

The code seems less crowded with switch

- Do you prefer ”else if” as one expression, if possible, (as in sample 3 lines 10 and 14) or separated into two expressions (as in sample 1 lines 16/17)?

Answer	Count	%
one expression	34	91.9
separated expressions	3	8.1

Easier to understand. If the ”else if” is separated into

two expressions, we will still annotate them as one expression during analysis.

visually easier to determine if-else block

even better if can reduce from ”if(A) else if(!A) ” to simply ”if(A) else ”

One expression makes it cleaner, easier to encapsulate the logic

Separated solely if it makes sense to include multiple separations again, e.g., by separating the positive case into 5.

- What did you like most about your favourite sample?

Again, it’s short and concise. No deeply nested control flow structures, clean conditions.

Easy to follow control flow

it’s not my least favourite sample

Easy to read, only one condition dazzled me

Clean

very clear and concise control flow conditions, compared to the other samples.

more compact and easily understandable

Without studying bitshifts or triple nested if statements, the purpose of the function became clear within a second.

That the format string is directly resolved instead seeing the address of the format string

short

easiest to understand

Simple control flow

straight forward inequality checks

concise way of presentation

Easy to follow and conditional checks were easy to understand

Easier to understand

Short and clear

short and neat, making analysing conditions easy

Easier to read.

less nested-if; no redundant nested-if with same condition

Clean and directly easy to see the control flow & conditions & short please

if-else was easy to follow

Seems closest to human-written source-code

among the nesting depth of the given examples and concise logical split of value ranges captured by the ifs

Less unnecessary instructions.

data type conversion is not required

comprehensible flow

The format string is visible.

neat. straightforward.

”else if” is one expression when there is only one block after else

The more python-esque indentation. C/C++ style indentation is ugly in my opinion

---

- What did you dislike about your least favourite sample?

Switch in switch is awful!

Noise from typecasts and switch statements, where if-else would have been more natural.

so many fucking casts

unnecessary if-block where the reason is not clear

Too complicated

nested switch case, conditions that require more time to understand (even if it may be more representative of the asm instructions)

switch case in this scenario become rather confusing.

I usually don’t like nested switches. The switch head is hard to read and understand.

the address of the format string printer and using switch case for this limeted ”cases”

long

more complicated than the other 2 samples

Unnecessary conditions like the nested if

insane switch statements

Complex conditional check

multiple if

The use of hex number and & operation is more difficult to understand

if all the 3 samples are the same, unnecessary opertions

Can be confusing at times.

switch statement makes it not obvious only 2 cases possible i.e. it is an ”if-else”

Bitshifts will occur more mental effort to reverse and understand the same thing

lengthy and harcoded numbers when unneeded

switch in switch is confusing

Use of casting and bit-level operations.

bitmask/shift magic requires more intrinsic knowledge about number representation than straightforward if/else constructs.

Outer switch instructions feels unnecessary complicated to read.

data type conversion is required

switch cases

The switch obscures the semantics!

lots of logical thinking needed.

Unnecessary extra pointer variable (no use of array)

Too many ifs mess with ones head, because you have to keep a lot of stuff in mind

---

Part B - Q3

- Please consider the following three samples and order them from your favourite to least favourite:
  Sample 1 (IDA):

```
int test3() {
  int var_0;
  int var_1;
  int i;

  printf("Please enter a number:");
  __isoc99_scanf("%d", &var_0);
  while ( var_0 <= 99 ) {
    var_1 = rand();
    for ( i = 0; i < var_1; ++i ) {
      var_0 += i;
      if ( var_0 > 100 ) {
        return printf("The result is %d\\n",
↪  var_0);
      }
    }
  }
  return printf("The result is %d\\n",
↪  var_0);
}
```

  Sample 2 (Ghidra):

```
void test3(void) {
  int var_0;
  int var_2;
  int var_1;

  printf("Please enter a number:");
  __isoc99_scanf(0x0804a01f,&var_0);
  while (var_0 < 100) {
    var_2 = rand();
    var_1 = 0;
    while (var_1 < var_2) {
      var_0 = var_1 + var_0;
      if (100 < var_0) {
        goto Label_1;
      }
      var_1 = var_1 + 1;
    }
  }
  Label_1:
  printf("The result is %d\\n",var_0);
  return;
}
```

  Sample 3 (our):

```
int test3() {
  int exit_4;
  int var_0;
  int var_1;
  int * var_2;
  int var_3;
  int var_4;

  printf("Please enter a number:");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a01f, var_2);
  while(var_0 <= 99) {
    var_4 = rand();
    var_3 = 0;
    while(true) {
      if (var_3 >= var_4) {
        exit_4 = 0;
        break;
      }
      var_1 = var_3 + var_0;
      if (var_3 + var_0 > 100) {
        var_0 = var_1;
        exit_4 = 1;
        break;
      }
      var_3 = var_3 + 1;
      var_0 = var_1;
    }
    if (exit_4 != 0) {
      break;
    }
  }
  var_3 = printf("The result is %d\\n",
↪  var_0);
  return var_3;
}
```

Count	Ranks		
%	1	2	3
Sample 1	34 91.9	3 8.1	0 0.0
Sample 2	2 5.4	34 91.9	1 2.7
Sample 3	1 2.7	0 0.0	36 97.3

- Do you think goto is a good choice when breaking out of the inner loop?

Answer	Count	%
yes, goto is a good choice	14	37.8
no, goto should not be used	23	62.2

As long as goto is only used similar to a ”break;” statement, it can help readability. But if mixed with other control flow constructs it often reduces readability.

As you see in Sample 1, it’s just unnecessary

In this particular case, the goto is unnecessary, because the function can terminate. But in general a goto to jump out of an inner loop can improve the readability.

If we are only comparing between sample 2 and 3,

then yes it makes the decompiled code easier to read.

Just do the printf and return statement there.

easy to understand

Having a goto is more intuitive to understand

i don't write C using goto, therefore it should not be used in decompilation

prefer using break

Prefer break than goto statement

if the goto does not lead to another jump, goto is easier to read

goto is bad, but still better then lengthy code!

In this case there was only one goto and it acted like a finally block, and so was easy enough to understand and did help in some ways, too. Where you begin to have multiple gotos and that don't necessarily return, it may become harder to understand. I think the analyst can get used to the goto construct eventually, and it does help with reducing code repetition in some cases, but it adds to the learning curve as it's not something normally used by those coming from a programming background.

It feels like the redundancy of the "return printf()" does not hurt here. Because it's the last statement within the loop(s), it allows directly to "stop" trying to understand the context.

Goto is fine if there are at most two different labels to jump to and if the switch-body is of small-ish size (probably

Gotos obscure the control flow, make it harder to read. Also feels like the decompiler was just lazy.

It is fine here when used singly, but in general multiple gotos and labels make code harder to read.

It messes with my head

- Do you think it is a good idea to duplicate (return-)statements to avoid complicated conditions or gotos?

Answer	Count	%
copy statements to simplify the structure	30	81.1
never copy a statement	7	18.9

If a statement is complicated, you may end up trying to understand it twice instead of once. I also sometimes accidentally overlook return statements in the code when skimming over it. Just one return at the end decreases the risk of that happening.

In general duplication of code is bad and yields convoluted code, but to a certain extend and to simplify the structure, this might help.

In this case, it produced much cleaner decompiled code, so yes.

If it's a single (or reasonable short) statement: copy; Otherwise: Extract as a new method?

for me it makes the analysing a little bit faster

Complicated branch structures are harder to trace

easier to understand the code

Too many return statement complicate the code

Depends, copying could make it simplified in larger functions

don't do that if you can avoid it.

Again, this more similar to how a programmer would do it, I think.

yes, see above.

Copy statements if restructuring/ rewriting them together afterwards is mostly easily possible.

I don't see any downsides to copying statements. In particular if it results in cleaner code, i.e. no goto.

I think this is useful if the destination statement(s) are simple and short. Perhaps it can be combined with some indicator to tell the reader this statement is duplicated (and not a new code block).

Goto is super ugly. I want decompilation to be easy to understand and not adhere to clean code rules

---

- What did you like most about your favourite sample?

Same as always: it's the shortest one and it doesn't contain any unusual constructs.

Easy to follow control flow, the scanf-string was inserted in the code.

It's short

short, intention of the function is very clear

Simple

Much easier to read, looks more like code someone will write.

compact and easy to understand

No gotos, easy to understand, for loop without a break used which further simplifies the method.

format string + the simplicity

simple and short

easy to understand and straightforward

Easy to see when loops break or function returns

looks clean

Least number of lines and easy to understand

short

Short and clear

short and easier to read loops

It copied the return statements so it is easier to

understand.

simplest structure among three. even better if can use break

Easy to see control flow

nice & short, no gotos.

short and readable

It looked closest to something I'd program.

linear readability.

Conciseness, by far most obvious control flow & semantics.

for loop

Clean, sematics are clear on first pass.

compact, neat, nicely structured

The for-loop was recognised, making it simpler to read.

Its clean and easy to read

- What did you dislike about your least favourite sample?

Deep nesting, lots of variables; this looks totally cluttered.

Too many local variables

It's so fucking long and uses a bunch of variables

too many variables to keep track of

Complicated

Too many variables to track.

long length of code

Useless variables, too long, while(true) where not necessary.

too many variables and too complicated for this kind of "task"

too long

long and complicated

Complicated control flow

looks bloated

To many variables and longest of among the 3

long lines of code

Too many variable and operation and the use of !=

unnecessary complexity, can be made simplier

Nested while loops

complex

Too many branches & conditions

too lengthy.

confusing structure

It seemed most complex.

cluttered, too many variables compared to the others.

Way too many unnecessary case-changes, variables; didn't use common counter-variable names, as the 'i' in sample 1. I much prefer the x += y statement over x = x + y. Too many breaks in combination with the 'true' while-condition.

while True statement

The semantics are completely unclear. In particular the while(1) with break should be avoided if possible.

I dislike my least favourite sample because I have my reasons. Some of the reasons are as follow: it is too verbosive with quite a few variables involved. Another such reason is that the additional variables and lines more difficult to read. Following that, the variables, lines and nesting of the conditional variables added to the difficulty.

Long while loop with multiple breaks.

Its long, its ugly and it have to keep too much stuff in mind

---

Part B - Q4

- Please consider the following three samples and order them from your favourite to least favourite:
  Sample 1 (our):

```c
int test4() {
  int * i;
  int var_0;
  int var_1;
  int var_2;
  int * var_3;

  var_2 = rand();
  printf("Enter an number: ");
  var_3 = &(var_0);
  __isoc99_scanf(0x804a01f, var_3);
  var_3 = 0;
  for (i = 0; i < var_0; i = i + 1) {
    while(var_2 + i > var_0) {
      if (var_2 - i > var_0) {
        var_0 = i + var_0;
        continue;
      }
      var_1 = i + var_0;
      if (var_2 == i + var_0) {
        var_0 = var_2;
        break;
      }
      var_0 = var_1;
    }
    if (var_2 == var_0) {
      var_3 = i;
    }
    if (var_3 > 0) {
      printf("The random number is %d\\n",
↪  var_3);
      break;
    }
    var_0 = var_0 - 1;
  }
  return 0;
}
```

Sample 2 (IDA):

```
int test4() {
  int var_0;
  int var_1;
  int var_2;
  int var_3;

  var_1 = rand();
  printf("Enter an number: ");
  __isoc99_scanf("%d", &var_0);
  var_3 = 0;
  var_2 = 0;
  while ( var_3 < var_0 ) {
    while ( var_3 + var_1 > var_0 ) {
      if ( var_1 - var_3 <= var_0 ) {
        var_0 += var_3;
        if ( var_1 == var_0 ) {
          var_0 = var_1;
          break;
        }
      }
      else {
        var_0 += var_3;
      }
    }
    if ( var_1 == var_0 ) {
      var_2 = var_3;
    }
    if ( var_2 > 0 ) {
      printf("The random number is %d\\n",
↪   var_2);
      return 0;
    }
    --var_0;
    ++var_3;
  }
  return 0;
}
```

Sample 3 (Ghidra):

```
undefined4 test4(void) {
  int var_0;
  int var_1;
  int var_2;
  int var_3;

  var_1 = rand();
  printf("Enter an number: ");
  __isoc99_scanf(0x0804a01f,&var_0);
  var_3 = 0;
  var_2 = 0;
  do {
    if (var_0 <= var_3) {
      return 0;
    }
    do {
      while( true ) {
        if (var_1 + var_3 <= var_0) {
          goto Label_1;
        }
        if (var_1 - var_3 <= var_0) {
          break;
        }
        var_0 = var_3 + var_0;
      }
      var_0 = var_3 + var_0;
    } while (var_1 != var_0);
    var_0 = var_1;
  Label_1:
    if (var_1 == var_0) {
      var_2 = var_3;
    }
    if (0 < var_2) {
      printf("The random number is
↪   %d\\n",var_2);
      return 0;
    }
    var_0 = var_0 + -1;
    var_3 = var_3 + 1;
  } while( true );
}
```

Count	Ranks		
%	1	2	3
Sample 1	21 56.8	16 43.2	0 0.0
Sample 2	16 43.2	20 54.1	1 2.7
Sample 3	0 0.0	1 2.7	26 70.3

- Do you prefer while(true) or dowhile(true)?

Answer	Count	%
while	37	100
do-while	0	0

Depends on the situation, eh?

Especially for longer loop-bodies as we see in the examples, while eases the readability. As we only talk about while(true), there is absolutely no reason for dowhile(true) imho

more logical flow

Can see the condition at once

no actual pref. just that if there is a nested while or nested dowhile, it has to be the same (ie, all loops using dowhile or all loops using while)

definitely while

while shows the condition under which execution happens, do-while appears to require an additional if to branch out in case the condition is violated -¿ feels a bit cluttered.

Exceptions are cases for which only one (possibly longer) break-condition exists that can be nicely caught before the main while-body. In case loops are nested, I strongly prefer while over do-while.

When coding I *never* use do-while. I'd rather unroll the loop once. I find it easier to read.

do-while leads to crowded code

- Do you prefer continue-statements and only one if-statement or no continue-statement and an if-else-statement?

Answer	Count	%
continue and if	15	40.5
if-else	22	59.5

But I think this depend on the sample at hand. Might be different for different samples.

Everything that reduces perceived code complexity helps.

Also depends on the situation

But only if the body of the if-statement is short and at the top of the loop

Depends on how long the remaining code (after the continue) is. Continue as a ”jump” is not always nice, but it clearly indicates that I do not have to care about the rest of the body in this iteration. When the else part only spans 3 or 4 lines, I would rate if-else as reasonable as well.

easier to understand

hard choice, but continue seems a bit easier to read

it is easier to follow an if-else block, as the continue-if condition introduces another mental note of a condition

in this example, if-else felt like it achieves better logical segmentation of the while's body.

If the if-condition and body is short, while the else is long (and the corresponding negated if-condition would be worse to read), continue+if can be better.

When coding I like to omit the else in favour of a continue. Thus I prefer reading code that was written that way.

if-else leads to crowded code

- What did you like most about your favourite sample?

Hard to explain in this case. It just looks more ”natural”. Maybe bacause we have one indent-level less.

an outer for loop

the two while-loops made the scope of the values clear, decrement and increment operations

Clean

the exit condition variables are more easy to track compared to the rest.

the loop is easier to understand in sample 1

for-loops, less nesting than the other samples, one can easily follow the flow

Using continue and that we have a for and a while loop

prefer having for loop

easiest to understand amongst the 3

for loop

involves less conditional statements

Easiest to follow and I did not have to deal with pointers

short

While and if is easier to understand compared to for followed by while or do (while true)

structurally easier to visualise the code

Easier to follow the flow.

better structuring, a bit shorter and nicer.

The loop constructs were simplest.

inc/dec operators.

Clearer control-flow, especially in regards to which & how many variables that are part of a control-flow-condition are changed inside the body.

Structured view

Fairly clean. For loop.

Did not have too much redirection that breaks the reading flow (goto, break, continue)

No else branches

- What did you dislike about your least favourite sample?

This looks obscure somehow.

while(true) and goto inside a loop

Why are we using goto again

logic is not clear at all, if one has not seen the two other samples beforehand

Complicated

harder to keep track of when/how the loop exits

dislike 'goto' and 'while loop' being used together as need to jump around the code more often.

Mix of while and do while, goto, nesting-levels

goto statements

hate do-while

do while loop

goto

Most difficult to follow

long lines of code

too many combination, do while true , goto and break

mixture of dowhile and while

they were all relative bad, 2 & 3.

goto

Use of goto to jump to part of an outer loop

compared to the other two, it's harder to follow what is happening why (loops, conditions)

long, nested do-while loops and the label being defined inside the while loop

Label

do-while and goto!

it mixes while with do-while, and it's difficult to keep track of conditions above and below

Long and crowded

Part B - Q5

- Please consider the following three samples and order them from your favourite to least favourite:
  Sample 1 (IDA):

```
int test5() {
  int var_0;
  int var_1[2];
  int var_2;
  int j;
  int i;

  printf("Enter a number: ");
  __isoc99_scanf("%d", var_1);
  printf("Enter a second number: ");
  __isoc99_scanf("%d", &var_0);
  printf("You chosed the numbers %d and
↪  %d\\n", var_1[0], var_0);
  var_2 = 0;
  var_1[1] = 0;
  for ( i = 0; i < var_1[0]; ++i ) {
    for ( j = 0; j < var_0; ++j ) {
      if ( var_1[0] - i == var_0 - j ) {
        var_2 = 1;
      }
    }
    if ( !var_2 ) {
      if ( var_1[0] >= var_0 ) {
        var_0 = 2 * var_0 - var_1[0];
      }
      else{
        var_0 -= var_1[0];
      }
    }
    var_2 = 0;
  }
  return printf("The numbers are %d and
↪  %d\\n", var_1[0], var_0);
}
```

Sample 2 (our):

```
int test5() {
  int i;
  int j;
  int var_0;
  int var_1;
  int * var_2;
  int var_3;

  printf("Enter a number: ");
  var_2 = &(var_1);
  __isoc99_scanf(0x804a01f, var_2);
  printf("Enter a second number: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a01f, var_2);
  printf("You chosed the numbers %d and
↪  %d\\n", var_1, var_0);
  for (i = 0; i < var_1; i = i + 1) {
    var_3 = 0;
    for (j = 0; j < var_0; j = j + 1) {
      if (var_1 - i == var_0 - j) {
        var_3 = 1;
      }
    }
    if (var_3 == 0) {
      if (var_1 < var_0) {
        var_0 = var_0 - var_1;
      }
      else {
        var_0 = (var_0 + var_0) - var_1;
      }
    }
  }
  var_3 = printf("The numbers are %d and
↪  %d\\n", var_1, var_0);
  return var_3;
}
```

Sample 3 (Ghidra):

```
void test5(void) {
  int var_1;
  int var_0;
  undefined4 var_2;
  int var_3;
  int var_4;
  int var_5;

  printf("Enter a number: ");
  __isoc99_scanf(0x0804a01f,&var_0);
  printf("Enter a second number: ");
  __isoc99_scanf(0x0804a01f,&var_1);
  printf("You chosed the numbers %d and
↪  %d\\n",var_0,var_1);
  var_2 = 0;
  var_5 = 0;
  while (var_3 = 0, var_5 < var_0) {
    var_4 = 0;
    while (var_4 < var_1) {
      if (var_0 - var_5 == var_1 - var_4) {
        var_3 = 1;
      }
      var_4 = var_4 + 1;
    }
    if (var_3 == 0) {
      if (var_0 < var_1) {
        var_1 = var_1 - var_0;
      }
      else {
        var_1 = var_1 * 2 - var_0;
      }
    }
    var_5 = var_5 + 1;
  }
  printf("The numbers are %d and
↪  %d\\n",var_0,var_1);
  return;
}
```

Count	**Ranks**		
%	1	2	3
Sample 1	20 54.1	13 35.1	4 10.8
Sample 2	15 40.5	19 51.4	3 8.1
Sample 3	2 5.4	5 13.5	30 81.1

- Do you prefer the condition if(var == 0) or if(!var)?

Answer	Count	%
if(var == 0)	30	81.1
if(!var)	7	18.9

If you want to match the number 0 use 0. Use ! only for booleans or "wrong things".

This is clearly not Python

As we talk about an int here, I like comparison to another int better

doesn't really matter, but would prefer memcmp/strcmp == 0 instead of !memcmp/strcmp

var==0 seems more explicit or more intuitive to understand than !var

condition is simpler to understand

in general (!var). but that really doesn't make a big difference.

In this case it looks like it is meant to be a boolean, so !var makes sense.

Assuming that the knowledge of var being int-castable is found nearby/ easily remembered

I much prefer explicit zero checks "== 0", takes load off my brain. With "if(var)" I always have to take a moment to remind myself what that means.

Undecided. Depends on the code block following. Sometimes it is more natural for a if (!var).

I don't feel strongly about this one. ! is convenient but could be misleading if the variable is used as a counter and not a flag.

- What did you like most about your favourite sample?

The control flow structure looks best to me.

string argument for scanf inserted into the code.

no strange auxiliary variables for the scanf

clear data and pointer types

Cleaner

least disliked

more simplified

i and j as for loop variables ease the understanding as the names clearly show their purpose.

despite var_ also increment vars like i and j are used and the == 0

short and clean

variable naming looks clean and it has for loops

Easiest to follow

short

simple variable

for loops and conditions are clear

a bit better structured than the others.

good and easy variable names

Seemed to recover semantics best, e.g. boolean in line 21 and arithemetic in line 23

"cleanest" for loops

for loop with ++ statements, no totally-unused variables

if == 0 statement

Better local variable detection (no unnecessary arrays) and explicit zero checks.

The for-loops are elegant and easy to read.

Shorter and for-loop

- What did you dislike about your least favourite sample?

Here we have the worst structure of the three samples.

no for loops

unneeded variable, variable initialization in the while-condition

Complicated

while loop when for loop works

dislike the while-loop in this scenario, prefer for-loop in other sample

undefined4 var_2; Many variables, while loops, typo in "chosed"

undefined4 and the while while nesting

uses while instead of for loop

while (var_3 = 0, var_5 ¡ var_0)

Hardest to follow and pointers

long lines of code

mixture of while and if-else

unnecessary complexity; undefined?

undefined type is a no go.

While loops used instead of for loops.

I think for loops are easier to read than while loops.

The x = x+y statements, especially since they are just incremented in such a manner much faster understood by a for-loop.

While instead of for

Used while loops when for-loops were possible, but this was not too bad.

I hate while-loops

Part B - Q6

- Please consider the following three samples and order them from your favourite to least favourite:
  Sample 1 (Ghidra):

```
undefined4 test6(void) {
  int var_4;
  int var_0;
  int var_1;
  int var_2;
  int var_3;

  printf("Find prime numbers between 1 to :
↪  ");
  __isoc99_scanf(0x0804a01f,&var_0);
  printf("All prime numbers between 1 to %d
↪  are:\\n",var_0);
  var_3 = 2;
  do {
    if (var_0 < var_3) {
      return 0;
    }
    var_1 = 1;
    var_2 = 2;
    while (var_2 <= var_3) {
      var_4 = dividable(var_3,var_2);
      if (var_4 != 0) {
        var_1 = 0;
        break;
      }
      var_2 = var_2 + 1;
    }
    if (var_1 == 1) {
      printf("%d, ",var_3);
    }
    var_3 = var_3 + 1;
  } while( true );
}
```

Sample 2 (IDA):

```
int test6() {
  int var_0;
  int var_1;
  int j;
  int i;

  printf("Find prime numbers between 1 to :
↪  ");
  __isoc99_scanf("%d", &var_0);
  printf("All prime numbers between 1 to %d
↪  are:\\n", var_0);
  for ( i = 2; i <= var_0; ++i ) {
    var_1 = 1;
    for ( j = 2; j <= i; ++j ) {
      if ( dividable(i, j) ) {
        var_1 = 0;
        break;
      }
    }
    if ( var_1 == 1 ) {
      printf("%d, ", i);
    }
  }
  return 0;
}
```

Sample 3 (our):

```
int test6() {
  int i;
  int var_0;
  int * var_1;
  int var_3;
  unsigned int var_4;

  printf("Find prime numbers between 1 to :
↪  ");
  var_1 = &(var_0);
  __isoc99_scanf(0x804a01f, var_1);
  printf("All prime numbers between 1 to %d
↪  are:\\n", var_0);
  for (i = 2; i <= var_0; i = i + 1) {
    var_3 = 2;
    while(true) {
      if (var_3 > i) {
        var_3 = 1;
        break;
      }
      var_4 = dividable(i, var_3);
      if (var_4 != 0) {
        var_3 = 0;
        break;
      }
      var_3 = var_3 + 1;
    }
    if (var_3 == 1) {
      printf("%d, ", i);
    }
  }
  return 0;
}
```

Count	Ranks		
%	1	2	3
Sample 1	0 0.0	11 29.7	26 70.3
Sample 2	36 97.3	1 2.7	0 0.0
Sample 3	1 2.7	25 67.6	11 29.7

- Do you prefer for-loops or while-loops?

Answer	Count	%
for-loops	33	89.2
while-loops	4	10.8

In my opinion for loops are always preferable as long as you keep any magic or complex things out of the loop header. Don't do any complex or unexpected computations in your loop header.

It's easier to see how often they loop

depends, but in this case the for-loops are better

Loop behavior is much more clear. Always use for loop if you iterate over a fixed range. Even when you have to use break.

for loops are neater and shorter

Can more explicitly see how conditions are calculated

easier to visualise the condition

for-loops, maybe. not a big difference. I probably don't care much.

When it actually makes sense (not e.g. for(; condition ;))

conditions in the for loops at beginning of their body makes it easy to understand how they'll work (even with the break).

In case the loop-condition variables are changed in more complex ways, a while-loop with a clear break-condition is better.

The semantics are more easy to decipher.

No preference.

while-loops make me sad. The loop body is more crowded

- What did you like most about your favourite sample?

It was the only one where I could understand what's going on whithin the first 5-10 seconds.

concise

It's short

clear logic, function call in the if-condition avoids unnecessary variable assignment

Cleaner

no wasted variables, easy to follow code

for-loop is easier to follow and do the dividable calculation.

For loops, variable namings (i,j), short, easy to follow

no unnecessary copy of vars

short and uses for loop

two clean looking for loops and division check in if condition

clear understanding of the codes

Easiest to follow and had the fewest line of code

short and easy to understand code structure

no while loops

nice & short, even nice variable names, that's unusual, and much better than the other two,

short and clear

It was most concise and the functionality was easy to figure out.

clean and straight forward

Very clear loop-behaviour from the first look on it.

for statement

It's pretty much the same as I would write the code. Two for loops, rather than nasty whiles.

For-loops are elegant and the counters are already named to defaults (i, j)

For-loops

- What did you dislike about your least favourite sample?

The while(true) loops were unexpected here.

do-while

The do-while loop

if-condition with break right after the function start distracts from the intention and should be a do-while-loop

Complicated

do while

the while loop make the code more confusing to do the dividable calculation

Return type, do while, does not follow the algorithms intuition.

copy vars for format string

hate do-whiles

do while loop

has several different conditional statements which may require longer time to understand the codes

Hardest to follow

Mixture of do, while and if-else make it hard to understand and trace

mix of dowihle and while

lengthy, ugly.

do

It used do-while loops and while loop where a for loop would have been more appropriate.

not a fan of the do-while like in the examples before

Combined use of var_3 as incrementing condition-variable and setting it as "return" value combined with break (l. 16)

do while

do while(true). Might as well read the disassembly rather than the decompiled code.

while-true with breaks

Do-while

D. Part C

Part C Q1: expression propagation limits

- We have decompiled the same function with different expression propagation limits. Please choose your favourite version with respect to the most optimal nesting complexity of instructions.

Sample 1:

```
int palindrom() {
  int var_0;
  int var_1;
  int var_2;
  int var_3;
  int var_4;
  int var_5;
  int * var_6;
  long var_7;
  printf("Enter an integer: ");
  var_6 = &(var_0);
  __isoc99_scanf(0x804a028, var_6);
  var_1 = var_0;
  var_2 = 0x0;
  while(true) {
    var_3 = var_0;
    if (var_3 == 0) {
      break;
    }
    var_3 = var_0;
    var_7 = var_3 * 0x66666667;
    var_4 = var_7 >> 0x20;
    var_4 = var_4 >> 0x2;
    var_5 = var_3 >> 0x1f;
    var_5 = var_4 - var_5;
    var_4 = var_5 << 0x2;
    var_4 = var_4 + var_5;
    var_4 = var_4 + var_4;
    var_3 = var_3 - var_4;
    var_4 = var_2 << 0x2;
    var_2 = var_4 + var_2;
    var_2 = var_2 + var_2;
    var_3 = var_3 + var_2;
    var_4 = var_0;
    var_7 = var_4 * 0x66666667;
    var_2 = var_7 >> 0x20;
    var_2 = var_2 >> 0x2;
    var_4 = var_4 >> 0x1f;
    var_2 = var_2 - var_4;
    var_0 = var_2;
    var_2 = var_3;
  }
  if (var_1 == var_2) {
    printf("%d is a palindrome.", var_1);
  }
  else {
    printf("%d is not a palindrome.",
↪   var_1);
  }
  return 0x0;
}
```

Sample 2:

```
int palindrom() {
  int i;
  int var_1;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  int var_6;
  long var_7;
  printf("Enter an integer: ");
  var_2 = &(var_1);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for(i = var_1; i != 0; i = var_4 - var_5){
    var_7 = i * 0x66666667;
    var_4 = var_7 >> 0x20;
    var_4 = var_4 >> 0x2;
    var_6 = i >> 0x1f;
    var_5 = var_4 - var_6;
    var_4 = var_4 - var_6;
    var_4 = var_4 << 0x2;
    var_4 = var_4 + var_5;
    var_4 = var_4 + var_4;
    var_5 = var_3 << 0x2;
    var_3 = var_5 + var_3;
    var_3 = var_3 + var_3;
    var_4 = i - var_4;
    var_3 = var_4 + var_3;
    var_7 = i * 0x66666667;
    var_4 = var_7 >> 0x20;
    var_4 = var_4 >> 0x2;
    var_5 = i >> 0x1f;
  }
  if (var_1 != var_3) {
    printf("%d is not a palindrome.",
↪   var_1);
  }
  else {
    printf("%d is a palindrome.", var_1);
  }
  return 0x0;
}
```

Sample 3:

```
int palindrom() {
  int i;
  int var_1;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  printf("Enter an integer: ");
  var_2 = &(var_1);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_1; i != 0; i = var_4 - (i >>
↪   0x1f)) {
    var_5 = ((i * 0x66666667) >> 0x20) >>
↪   0x2;
    var_4 = (var_5 - (i >> 0x1f)) << 0x2;
    var_4 = var_4 + (var_5 - (i >> 0x1f));
    var_3 = (var_3 << 0x2) + var_3;
    var_3 = var_3 + var_3;
    var_3 = (i - (var_4 + var_4)) + var_3;
    var_4 = ((i * 0x66666667) >> 0x20) >>
↪   0x2;
  }
  if (var_1 == var_3) {
    printf("%d is a palindrome.", var_1);
  }
  else {
    printf("%d is not a palindrome.",
↪   var_1);
  }
  return 0x0;
}
```

Sample 4:

```
int palindrom() {
  int i;
  int var_1;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  printf("Enter an integer: ");
  var_2 = &(var_1);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
```

```
  for (i = var_1; i != 0; i = (((i *
↪  0x66666667) >> 0x20) >> 0x2) - (i >>
↪  0x1f)) {
    var_5 = (((i * 0x66666667) >> 0x20) >>
↪  0x2) - (i >> 0x1f);
    var_4 = (((i * 0x66666667) >> 0x20) >>
↪  0x2) - (i >> 0x1f);
    var_4 = ((var_4 << 0x2) + var_5) +
↪  ((var_4 << 0x2) + var_5);
    var_3 = ((var_3 << 0x2) + var_3) +
↪  ((var_3 << 0x2) + var_3);
    var_3 = (i - var_4) + var_3;
  }
  if (var_1 == var_3) {
    printf("%d is a palindrome.", var_1);
  }
  else {
    printf("%d is not a palindrome.",
↪  var_1);
  }
  return 0x0;
}
```

Sample 5:

```
int palindrom() {
  int i;
  int var_0;
  int * var_2;
  int var_3;
  int var_4;
  printf("Enter an integer: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_0; i != 0; i = (((i *
↪  0x66666667) >> 0x20) >> 0x2) - (i >>
↪  0x1f)) {
    var_4 = (((i * 0x66666667) >> 0x20) >>
↪  0x2) - (i >> 0x1f);
    var_4 = (((((i * 0x66666667) >> 0x20) >>
↪  0x2) - (i >> 0x1f)) << 0x2) + var_4;
    var_3 = (i - (var_4 + var_4)) + (((var_3
↪  << 0x2) + var_3) + ((var_3 << 0x2) +
↪  var_3));
  }
  if (var_0 != var_3) {
    printf("%d is not a palindrome.",
↪  var_0);
  }
  else {
    printf("%d is a palindrome.", var_0);
  }
  return 0x0;
}
```

Answer	Count	%
limit 1 (Sample 1)	4	10.8
limit 3 (Sample 2)	7	18.9
limit 5 (Sample 3)	12	32.4
limit 7 (Sample 4)	4	10.8
limit 10 (Sample 5)	10	35.1

I was balancing between 5 and 7. 1 and 3 have too many lines. In 7 I find the computation in the for loop header too complex, I also don't understand why it doesn't have var_4 = var_5. In 10 the lines are too complex. But that may be because of the arithmetic stuff. Lots of shifts and +- are more complex than calls to functions with meaningful names for instance. I. e. for arithmetic stuff I'd probably opt for shorter lines while I'd accept longer lines for function calls.

Limit 7 looks best here due to statement symmetry, but I assume that this does not generalize.

Only two operands per line in that long block of calculations please

7 and 10 are unnecessary complex, there are too many operations to keep track of; 1 is useless; 3 and 5 are better

Cleaner

Unsure between 5 and 3, but 5 is understandable without looking at each line for a minute or counting parenthesis.

least complex

seems like a sensible tradeoff between number of assingments and their complexity

fewer variables to track

ugh, but still limit 10 makes the overview better I think.

I think I like middleground (limit 5) best. What would be great though: To be able to decide this on a case-by-case, e.g. by having this available as a parameter in a UI.

Rest not familiar with the boolean function

It would never try and understand these without running the individual lines. Limit 10 makes it easiest for me to copy&paste

---

Part C Q2: common subexpression elimination

- Please take a look at the following function and evaluate it with respect to long subexpressions that occur multiple times. Choose the most appropriate one.
  Sample 1:

```
int palindrom() {
  int i;
  int var_0;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  printf("Enter an integer: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_0; i != 0; i = var_4) {
    var_4 = (((i * 0x66666667) >> 0x20) >>
↪  0x2) - (i >> 0x1f);
    var_5 = (var_3 << 0x2) + var_3;
    var_3 = (var_4 << 0x2) + var_4;
    var_3 = (i - (var_3 + var_3)) + (var_5 +
↪  var_5);
  }
  if (var_0 == var_3) {
    printf("%d is a palindrome.", var_0);
  }
  else {
    printf("%d is not a palindrome.",
↪  var_0);
```

```
  }
  return 0x0;
}
```

Sample 2:

```
int palindrom() {
  int i;
  int var_1;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  printf("Enter an integer: ");
  var_2 = &(var_1);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_1; i != 0; i = var_4) {
    var_4 = (((i * 0x66666667) >> 0x20) >>
↪  0x2) - (i >> 0x1f);
    var_5 = (var_4 << 0x2) + var_4;
    var_3 = (i - (var_5 + var_5)) + (((var_3
↪  << 0x2) + var_3) + ((var_3 << 0x2) +
↪  var_3));
  }
  if (var_1 == var_3) {
    printf("%d is a palindrome.", var_1);
  }
  else {
    printf("%d is not a palindrome.",
↪  var_1);
  }
  return 0x0;
}
```

Sample 3:

```
int palindrom() {
  int i;
  int var_0;
  int * var_2;
  int var_3;
  int var_4;
  printf("Enter an integer: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_0; i != 0; i = (((i *
↪  0x66666667) >> 0x20) >> 0x2) - (i >>
↪  0x1f)) {
    var_4 = (((i * 0x66666667) >> 0x20) >>
↪  0x2) - (i >> 0x1f);
    var_4 = (((((i * 0x66666667) >> 0x20) >>
↪  0x2) - (i >> 0x1f)) << 0x2) + var_4;
    var_3 = (i - (var_4 + var_4)) + (((var_3
↪  << 0x2) + var_3) + ((var_3 << 0x2) +
↪  var_3));
  }
  if (var_0 != var_3) {
    printf("%d is not a palindrome.",
↪  var_0);
  }
  else {
    printf("%d is a palindrome.", var_0);
  }
  return 0x0;
}
```

Answer	Count	%
2 (Sample 1)	24	64.9
3 (Sample 2)	10	27.0
4 (Sample 3)	3	8.1

I do not understand what you want from me here

4 is horrible, to much is happening at once and it is already difficult to keep track of opening and closing paranthesis; updating var_3 in two steps seems reasonable

One more line does not hurt but eases readability.

less repetitive expressions

2 feels easiest to follow. But again, might be great to have such things available as a parameter.

It seems useful to replace long subexpressions with variables when the same subexpression is later used 2 or more times.

Again.. Too much calculating in head. So long lines for easy copy&paste

---

Part C Q3: mix of config options

- We have decompiled the same function with different configurations. Please choose up to three samples that are most comprehensible to you.
  Sample 1:

```
int palindrom() {
  int i;
  int var_0;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  int var_6;
  printf("Enter an integer: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_0; i != 0; i = (var_4 >> 0x2)
↪  - var_5) {
    var_4 = (i * 0x66666667) >> 0x20;
    var_5 = i >> 0x1f;
    var_6 = (var_4 >> 0x2) - var_5;
    var_6 = (var_6 << 0x2) + var_6;
    var_3 = (var_3 << 0x2) + var_3;
    var_6 = i - (var_6 + var_6);
    var_3 = var_6 + (var_3 + var_3);
  }
  if (var_0 == var_3) {
    printf("%d is a palindrome.", var_0);
  }
  else {
    printf("%d is not a palindrome.",
↪  var_0);
  }
  return 0x0;
}
```

Sample 2:

```
int palindrom() {
  int i;
  int var_0;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  int var_6;
```

```
  printf("Enter an integer: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for(i = var_0; i != 0; i = var_4 - var_5){
    var_5 = i >> 0x1f;
    var_4 = ((i * 0x66666667) >> 0x20) >> 0x2;
    var_6 = var_4 - var_5;
    var_6 = (var_6 << 0x2) + var_6;
    var_3 = (var_3 << 0x2) + var_3;
    var_3 = (i - (var_6 + var_6)) + (var_3 + var_3);
  }
  if (var_0 == var_3) {
    printf("%d is a palindrome.", var_0);
  }
  else {
    printf("%d is not a palindrome.", var_0);
  }
  return 0x0;
}
```

Sample 3:

```
int palindrom() {
  int i;
  int var_0;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  printf("Enter an integer: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_0; i != 0; i = var_4) {
    var_4 = (((i * 0x66666667) >> 0x20) >> 0x2) - (i >> 0x1f);
    var_5 = (var_4 << 0x2) + var_4;
    var_3 = (var_3 << 0x2) + var_3;
    var_5 = var_5 + var_5;
    var_3 = var_3 + var_3;
    var_3 = (i - var_5) + var_3;
  }
  if (var_0 == var_3) {
    printf("%d is a palindrome.", var_0);
  }
  else {
    printf("%d is not a palindrome.", var_0);
  }
  return 0x0;
}
```

Sample 4:

```
int palindrom() {
  int i;
  int var_0;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  printf("Enter an integer: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_0; i != 0; i = var_4) {
    var_4 = (((i * 0x66666667) >> 0x20) >> 0x2) - (i >> 0x1f);
    var_5 = ((var_4 << 0x2) + var_4) + ((var_4 << 0x2) + var_4);
    var_3 = ((var_3 << 0x2) + var_3) + ((var_3 << 0x2) + var_3);
    var_3 = (i - var_5) + var_3;
  }
  if (var_0 == var_3) {
    printf("%d is a palindrome.", var_0);
  }
  else {
    printf("%d is not a palindrome.", var_0);
  }
  return 0x0;
}
```

Sample 5:

```
int palindrom() {
  int i;
  int var_0;
  int * var_2;
  int var_3;
  int var_4;
  int var_5;
  printf("Enter an integer: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_0; i != 0; i = var_4) {
    var_4 = (((i * 0x66666667) >> 0x20) >> 0x2) - (i >> 0x1f);
    var_5 = (var_4 << 0x2) + var_4;
    var_3 = ((var_3 << 0x2) + var_3) + ((var_3 << 0x2) + var_3);
    var_3 = (i - (var_5 + var_5)) + var_3;
  }
  if (var_0 == var_3) {
    printf("%d is a palindrome.", var_0);
  }
  else {
    printf("%d is not a palindrome.", var_0);
  }
  return 0x0;
}
```

Sample 6:

```
int palindrom() {
  int i;
  int var_0;
  int * var_2;
  int var_3;
  int var_4;
  printf("Enter an integer: ");
  var_2 = &(var_0);
  __isoc99_scanf(0x804a028, var_2);
  var_3 = 0x0;
  for (i = var_0; i != 0; i = (((i * 0x66666667) >> 0x20) >> 0x2) - (i >> 0x1f)) {
    var_4 = (((i * 0x66666667) >> 0x20) >> 0x2) - (i >> 0x1f);
    var_4 = (((((i * 0x66666667) >> 0x20) >> 0x2) - (i >> 0x1f)) << 0x2) + var_4;
    var_3 = ((var_3 << 0x2) + var_3) + ((var_3 << 0x2) + var_3);
    var_3 = (i - (var_4 + var_4)) + var_3;
  }
  if (var_0 == var_3) {
    printf("%d is a palindrome.", var_0);
  }
  else {
    printf("%d is not a palindrome.", var_0);
```

```
  }
  return 0x0;
}
```

Answer	Count	%
Sample 1	13	35.1
Sample 2	16	56.8
Sample 3	21	56.8
Sample 4	18	48.7
Sample 5	12	32.4
Sample 6	6	16.2

- Please enter your comment here:

3rd choice. don't like to for loop condition

They all look the same to me

all bad

In selected samples, it was easiest to identify key state variables and how they were changed across iterations. Sub-expression computations were not repeated unnecessarily, as in sample 6.

shorter lines might be easier to understand, allow to rename/comment more finely.

Anything but samples 1 and 6 is okay to read.

Part C Q4: Complexity of for-loop conditions

- Please take a look at the following function that was decompiled either using a for-loop or using a while-loop. Choose the output that you prefer.
Sample 1:

```
int first_last_digit() {
  int i;
  int var_0;
  int var_1;
  int var_2;
  int * var_3;
  __x86.get_pc_thunk.bx();
  printf("Enter any number to find sum of
↪  first and last digit: ");
  var_3 = &(var_0);
  __isoc99_scanf(0x1693, var_3);
  var_2 = (((var_0 * 0x66666667) >> 0x20) >>
↪  0x2) - (var_0 >> 0x1f);
  var_2 = (((((var_0 * 0x66666667) >> 0x20)
↪  >> 0x2) - (var_0 >> 0x1f)) << 0x2) +
↪  var_2;
  for (i = var_0; i > 9; i = (((i *
↪  0x66666667) >> 0x20) >> 0x2) - (i >>
↪  0x1f)) {} var_1 = i;
  printf("Sum of first and last digit = %d",
↪  (var_0 - (var_2 + var_2)) + var_1);
  return 0x0;
}
```

Sample 2:

```
int fisrt_last_digit() {
  int var_0;
  int var_1;
  int var_2;
  int * var_3;
  __x86.get_pc_thunk.bx();
  printf("Enter any number to find sum of
↪  first and last digit: ");
  var_3 = &(var_0);
  __isoc99_scanf(0x1693, var_3);
  var_2 = (((var_0 * 0x66666667) >> 0x20) >>
↪  0x2) - (var_0 >> 0x1f);
  var_2 = (((((var_0 * 0x66666667) >> 0x20)
↪  >> 0x2) - (var_0 >> 0x1f)) << 0x2) +
↪  var_2;
  var_1 = var_0;
  while(var_1 > 9) {
    var_1 = (((var_1 * 0x66666667) >> 0x20)
↪  >> 0x2) - (var_1 >> 0x1f);
  }
  printf("Sum of first and last digit = %d",
↪  (var_0 - (var_2 + var_2)) + var_1);
  return 0x0;
}
```

Answer	Count	%
for-loop (Sample 1)	8	21.6
while-loop (Sample 2)	29	78.4

As I mentioned earlier: in general for loops are better, but not if you have complex computations in the header. Also, this is a very unintuitive way to use a foor loop (empty, body, using the "counter" variable after the loop).

complicated expressions in loop header of for-loops is a no-go.

Why is that an empty for loop with a big condition, what is this nonsense

the assignment is much to long; the resulting empty body seems weird; also the assignment right after the body throws me of

More for layout than for the loop type.

For loop seems more complicated to understand when there are long expressions involved

I prefer easy to read conditions

while-loop for this code easier to understand compared to the for-loop

prefer the conditions for the loop to be simple

Prefer for key computation to be kept in the loop body rather than header.

while loop feels more appropriate because the loop body / loop variable update looks non-trivial (compared to increments etc).

Because the i = (...) statement in the for-loop is so long, a horizontal deviation as seen in the while-loop feels easier and clutters less with the long statements from the lines above.

In this case the for loop does not make it easier to comprehend the semantics. I feel like anything that does more updates that are more complex than *simple* arithmatics "i += " or "i *= " might be better off being a while loop. I would personally draw the line somewhere around "i = (i+) % ". That's the most complex statement I would still use a for loop for.

for-loop is more suitable when the update statement is short, and the inner block has content.

It does not mattert what I choose here.. This seems too complicated in its details and too simple in its interface. In a real world scenario I would not try to understand this and instead let it run with various inputs and extrapolate the functionality from that.

Part C Q5: byte constants representation

- The output contains various of byte constant (from printable ASCII range) representation in the decompiler output. Rank them from most to least favourite.
  Sample 1:

```
int is_consonant_or_vowel() {
  byte var_0;
  byte * var_1;
  __x86.get_pc_thunk.bx();
  printf("Enter any character: ");
  var_1 = &(var_0);
  __isoc99_scanf(0xe8b, var_1);
  if ((var_0 != 'a') && (var_0 != 'e') &&
  ↪  (var_0 != 'i') && (var_0 != 'o') &&
  ↪  (var_0 != 'u') && (var_0 != 'A') &&
  ↪  (var_0 != 'E') && (var_0 != 'I') &&
  ↪  (var_0 != 'O') && (var_0 != 'U')) {
    if (((var_0 <= 'z') && (var_0 > '`')) ||
  ↪  ((var_0 <= 'Z') && (var_0 > '@'))) {
      printf("\'%c\' is Consonant.", (int)
  ↪  var_0);
    }
    else {
      printf("\'%c\' is not an alphabet.",
  ↪  (int) var_0);
    }
  }
  else {
    printf("\'%c\' is Vowel.", (int) var_0);
  }
  return 0x0;
}
```

Sample 2:

```
int is_consonant_or_vowel() {
  byte var_0;
  byte * var_1;
  __x86.get_pc_thunk.bx();
  printf("Enter any character: ");
  var_1 = &(var_0);
  __isoc99_scanf(0xe8b, var_1);
  if ((var_0 != 'a'/*97*/) && (var_0 !=
  ↪  'e'/*101*/) && (var_0 != 'i'/*105*/) &&
  ↪  (var_0 != 'o'/*111*/) && (var_0 !=
  ↪  'u'/*117*/) && (var_0 != 'A'/*65*/) &&
  ↪  (var_0 != 'E'/*69*/) && (var_0 !=
  ↪  'I'/*73*/) && (var_0 != 'O'/*79*/) &&
  ↪  (var_0 != 'U'/*85*/)) {
    if (((var_0 <= 'z'/*122*/) && (var_0 >
  ↪  '`'/*96*/)) || ((var_0 <= 'Z'/*90*/) &&
  ↪  (var_0 > '@'/*64*/))) {
      printf("\'%c\' is Consonant.", (int)
  ↪  var_0);
    }
    else {
      printf("\'%c\' is not an alphabet.",
  ↪  (int) var_0);
    }
  }
  else {
    printf("\'%c\' is Vowel.", (int) var_0);
  }
  return 0x0;
}
```

Sample 3:

```
int is_consonant_or_vowel() {
  byte var_0;
  byte * var_1;
  __x86.get_pc_thunk.bx();
  printf("Enter any character: ");
  var_1 = &(var_0);
  __isoc99_scanf(0xe8b, var_1);
  if ((var_0 != 97) && (var_0 != 101) &&
  ↪  (var_0 != 105) && (var_0 != 111) &&
  ↪  (var_0 != 117) && (var_0 != 65) &&
  ↪  (var_0 != 69) && (var_0 != 73) && (var_0
  ↪  != 79) && (var_0 != 85)) {
    if (((var_0 <= 122) && (var_0 > 96)) ||
  ↪  ((var_0 <= 90) && (var_0 > 64))) {
      printf("\'%c\' is Consonant.", (int)
  ↪  var_0);
    }
    else {
      printf("\'%c\' is not an alphabet.",
  ↪  (int) var_0);
    }
  }
  else {
    printf("\'%c\' is Vowel.", (int) var_0);
  }
  return 0x0;
}
```

Sample 4:

```
int is_consonant_or_vowel() {
  byte var_0;
  byte * var_1;
  __x86.get_pc_thunk.bx();
  printf("Enter any character: ");
  var_1 = &(var_0);
  __isoc99_scanf(0xe8b, var_1);
  if ((var_0 != 97/*'a'*/) && (var_0 !=
  ↪  101/*'e'*/) && (var_0 != 105/*'i'*/) &&
  ↪  (var_0 != 111/*'o'*/) && (var_0 !=
  ↪  117/*'u'*/) && (var_0 != 65/*'A'*/) &&
  ↪  (var_0 != 69/*'E'*/) && (var_0 !=
  ↪  73/*'I'*/) && (var_0 != 79/*'O'*/) &&
  ↪  (var_0 != 85/*'U'*/)) {
    if (((var_0 <= 122/*'z'*/) && (var_0 >
  ↪  96/*'`'*/)) || ((var_0 <= 90/*'Z'*/) &&
  ↪  (var_0 > 64/*'@'*/))) {
      printf("\'%c\' is Consonant.", (int)
  ↪  var_0);
    }
    else {
      printf("\'%c\' is not an alphabet.",
  ↪  (int) var_0);
    }
  }
  else {
    printf("\'%c\' is Vowel.", (int) var_0);
  }
  return 0x0;
}
```

Count	Ranks			
%	1	2	3	4
char (Sample 1)	14 37.8	8 21.6	12 32.4	3 8.1
char + decimal as comment (Sample 2)	15 40.5	13 35.1	6 16.2	3 8.1
decimal (Sample 3)	1 2.7	4 10.8	6 16.2	26 70.3
decimal + char as comment (Sample 4)	7 18.9	12 32.4	13 35.1	5 13.5

- Please enter your comment here:

Maybe not directly related, but: if it's possible to detect that the code checks for capital letters e.g. then I'd prefer to have the condition altered to var ¿= 'A'.

Come on, either this is text, then I want to see a character, or it is a number, then I want to see a number, and if the decompiler is wrong, I want to easily see that by looking at what chars the numbers are

only the decimal is hard to understand without a ascii table right next to it...

As the method really works on chars and does not calculate with them, the single char is sufficient and the only relevant information for this method.

would be good if it's toggle-able

Would rather have a more simple view of char or decimal at one time and choose their representation manually (like in IDA)

Char make more sense than decimal for code; the comment make the code hard to read (prefer mouse over to get the comments or comments at the end of line)

humans prefer reading what the condition means, but essential to know the decimal value

Could be unicode and i might not want it to display as ascii printable char at first. Would prefer decimal + char as comment

please hexadecimal instead of decimal. Who needs decimal anyway?

first choice should be concise (either dec or char), comments maybe as an optionally feature (via button).

The range-tests for characters (line 9) would be great as an individual if at the start.

In this particular case it is obvious (for a human) that it's actual characters rather than numbers, but I'd still like the decimal values in case the decompiler misinterpreted the use as characters.

if the byte variable is always compared to a printable-ascii value, char alone is great.

As many information as possible

Part C Q6

- Please consider the following variable naming schemes and rate them from favourite to least favourite:

```
1  // naming scheme 1
2  int var_0, var_1, var_2, var_3, var_4;
3
4  // naming scheme 2
5  int v1, v2, v3, v4, v5;
6
7  // naming scheme 3
8  int dog, cat, wolf, bear;
9
10 // naming scheme 4
11 int a, b, c, d, e, f;
12
13 // naming scheme 5
14 int loc_1, loc_2, loc_3;
```

Count	Ranks				
%	1	2	3	4	5
Scheme 1 (var_0,...) (Sample 1)	13 35.1	11 29.7	8 21.6	2 5.4	3 8.1
Scheme 2 (v1,...) (Sample 2)	12 32.4	11 29.7	10 27.0	3 8.1	1 2.7
Scheme 3 (dog,...) (Sample 3)	6 16.2	4 10.8	2 5.4	9 24.3	16 43.2
Scheme 4 (a,...) (Sample 4)	6 16.2	1 2.7	6 16.2	15 40.5	9 24.3
Scheme 5 (loc_1,...) (Sample 5)	0 0.0	10 27.0	11 29.7	8 21.6	8 21.6

- Do you have any better ideas for variable naming schemes?

Sorry, unfortunately not. What would be important would be that function arguments are distinguishable from local variables. While I ranked the animal scheme rather low, I'm actually not sure if it would be a bad idea. Unconventionally, but short words might be harder to confuse and easier to remember than the short v-things. I would have to see a larger code sample with animals to evaluate this. Also, I'm undecided if it makes sense to name counter variables "counter_1" or something (can you could detect this?). Not sure if consistency would be preferable here.

Guess the function of the variable and pick a suitable name :3

No, but I don't like the naming schemes where only an integer is incremented in the scheme. This makes the variables visually hard to distinguish. When (or if) the exact meaning of the variable is clear, it can be renamed after all.

meaningful word that can help understand more about the variable data

dog, ... suggests a meaning of the name. Same does the usage of a,b,... (don't let that count up to i or j!).

The other options make it clear that the name has no meaning, here I like v1, v2, ... most as; they are short and do not contain (useless) underscores. One could always include the type (or an indicator to the type), e.g. int_1 or i1, i2

allow user to rename var name

var_ and loc_ might actually be easier to click since it's 5+ chars. "v1", and "a" make the output more concise; using non-numbered variable (a, ...) may run out of symbols for huge functions.

In *some* cases with few variables, names such as 'tmp' can be okay, although not nice either. In case the variable is used as a counter/ incrementer/ decrementer, change it to a common counter-name, such as 'i', etc. .

nouns with varying lengths

Since the order of the variables does not matter the scheme v1, v2, ... does not provide any information beyond "this is a variable". Personally I like to look at the disassembly and the decompilate at the same time. In this scenario something like (v8, vF, ...) could be interesting, referring to the variable offsets ([ebp-8], [ebp-0xF], ...) since it adds information. However this quickly gets impossible when registers are involved.

inputs and outputs can be named specially (e.g. scanf input, file input, return values). anything that increments can be named from i, j, k...

It really doesn't matter except for cheme 3 :)

Part C Q7

- Please consider the following parameter naming schemes and rate them from favourite to least favourite:

```
// naming scheme 1
int param_1, param_2, param_3;

// naming scheme 2
int a1, a2, a3, a4, a5;

// naming scheme 3
// <animal_emojis>

// naming scheme 4
int arg_1, arg_2, arg_3;
```

Count	Ranks			
%	1	2	3	4
Scheme 1 (Sample 1)	11 29.7	10 27.0	13 35.1	3 8.1
Scheme 2 (Sample 2)	10 27.0	10 27.0	15 40.5	2 5.4
Scheme 3 (Sample 3)	2 5.4	1 2.7	3 8.1	31 83.8
Scheme 4 (Sample 4)	14 37.8	16 43.2	6 16.2	1 2.7

- Do you have any better ideas for parameter naming schemes?

See my answer from the previous questions. Maybe I'd like arg1 more than arg_1, but that's a minor issue. Again, I'm not sure about the animals. Might actually be a good idea (I'm more the visual type of person), but you should evaluate this separately with more samples.

param_dog, param_cat, param_wolf, ...

Guess the function of the parameters and pick a suitable name :3

Again, no, but the parameters should stand out from the local variables.

p1, p2, p3...; Secretly, I like the emoji scheme most.

allow user to rename var name

I cannot even read scheme 3, it's rectangles only.

same reasons as before

nouns with varying length, i.e., dog, cruise, mangosteen, etc

No, but the person who came up with the emoji naming should consider calling a therapist.

Maybe the police be involved as well. if the type is boolean, flag1, flag2...

It does not matter :)

Part C Q8

- Please consider the following two code snippets. Do you prefer reusing variable names for successive for-loops?
Sample 1:

```
int foo(int n, int m){
  for (i = 0; i < n; i++){
    do_important_stuff(i);
  }
  for (i = 0; i < m; i++){
    do_other_important_stuff(i);
  }
  return 0;
}
```

Sample 2:

```
int foo(int n, int m){
  for (i = 0; i < n; i++){
    do_important_stuff(i);
  }
  for (j = 0; j < m; j++){
    do_other_important_stuff(j);
  }
  return 0;
}
```

Answer	Count	%
I prefer Reuse (Sample 1)	20	54.0
I prefer No Reuse (Sample 2)	17	46.0

But really only if the variable is exclusively used as a

simple counter in the loop. If you have stunts like in one of the previous samples (the one with the empty body), I'd prefer different names.

Other variable names ('jklmn') should be reserved for nested for-loops.

Reusing variable names such as i and j is fine. If you do not stick to that naming scheme (why would'nt you?), do not reuse the name. i,j,k can be helpful for nested loops.

to have a better clarity on different for loops

not a strong preference but reuse helps with scope

clearer for analysis

Reason: if No Reuse, I have to look carefully to know values of variable i from first loop is not used again. If Reuse, it is obvious.

That is not about my preference. Do how it was written in the code. If the author did reuse it, I want reuse. Other, no.

Reuse if the first loop-body is not too long.

This is probably the only case in which I like variable reuse. Also I would like the loop variables to be scoped in the loop (for(*int* i = 0; ...)). That would make it obvious that its a variable reuse without side-effects.

Undecided. Either is fine.

Reuse is fine as long as i is never used outside the scope of these loops

It does not matter :)

E. Part D

- What other optimization/de-optimization features would you like to see in your decompiled code? (for example, string/ip address does not appear correctly, thus hard to trace origin, etc.)

Hard to say, I'd have to spend more time decompiling code to actually give an answer here. IP adresses would be great, though! It's hard to convey, but I'd like the output to look like a human could have written the code. In the previous answers this is mostly what I called short and concise. I don't thinks that any human (with a sane mind) would write switches in switches in a do-while loop for instance. So, I'd expect this also not to appear in the output of a decompiler. I'd like to have clean code ;-)

sensible names from the start

Array-detection; String-detection

Solving the C++ decompiled code mess, ability to generate vtables and jumping to indirect function calls

Nil

Minimization of bitwise operations when operating on numbers. Better resolution of function names.

dead code elimnation

Allow analysts to specify whether to reuse a variable or not (at the variable level, not a global setting). Sometimes reusing variable makes the decompiled cleaner but other times would make it confusing.

the decompile string results and the possible values presented in the codes

Remove unused variables, dead branches, unused branches

Structures.

For strings, able to recognize /suggests the language use via the type of character used as string and their appearance frequency; Able to import signature to recognized more function

to be able to distinguish unicode and ascii through visual emphasis rather than printing out the "raw char/bytes"

Resolving API calls correctly is very very very important. Any decompiler not able to resolve; ordinal API calls or any API call is useless. Resolve API calls always. Correctly resolving types is also very important. It would be cool if there was struct support, e.g., you can load in custom header files with typedefs, and apply them on your own to variables, and use their fields andsoon.

resolution of stack and xmm-based string definitions.

IP Address in clear; URL in clear

String literals are very important! Not all examples showed them in the decompilate.

- Prefer arguments to never change value within a function; - Perhaps leaving some decompilation preferences configurable, e.g. threshold for preferring switch-case or if-else, threshold for subexpression length

I don't know

- What quality of life features (for example, GUI interactivity, etc.) do you think would help you use a decompiler much more effectively?

An interactive GUI would be gold! I'd like to be able to rename variables and replace statements with different statements which I find easier to understand. So, basically some simple refactoring features like IDEs provide. Also, other features an IDE provides would be great (code (un-)folding, highlighting a seleced variable etc.). It would also be great to change the representation for single variables, e.g. have one variable as hex and a different one as char (selectable by the user).

highlighting (like highlighting all occurences of one variable), additional information on mouse-over (like known variable values?).

sensible names from the start

Scriptability and thus customizeability; integration of my favourite text editor

Live group collaboration on the same file, better annotations/note taking, better structure reconstruction

for beginner like myself, having GUI or comment as a hint to the data will be beneficial.

When some magic numbers accour a hint field could help

- Emitting memcpy/memset like what IDA Pro does for snippet of code that perform manual looping (e.g. rep stos on x86); - provide access to decompiler options at function or even more fine-grain level rather than at a global level.; - Probably not feasible, but would be nice to allow analyst to modify decompiled code directly so that it remains semantically the same but syntactically different.

having a GUI presentation

Function tracing

Graph representation of the flow of code. Built in "Help" menu and suggestions if user hit a problem

allow the user to toggle between the various decompiler parameters that will give rise to the different level of decompiling optimisation to suit the user's feel as different type of codes may be presented better using different params

It would be cool if there was struct support, e.g., you can load in custom header files with typedefs, and apply them on your own to variables, and use their fields andsoon.

Synchronizing with disassembly view, mouse-over for corresponding register/mem-slot, variable re-naming, copy-paste, concrete and symbolic execution, multiple decompilation candidates.

as mentioned earlier, options to control the aggressiveness of optimization/de-optimization might allow an analyst to find a configuration best suited for a given function on a case-by-case basis.

Automatically added comments regarding additional parameter-information (as to, in which range they should be) when extracting a method. If in doubt, whether a variable is used as character to int, add the character as comment.

GUI - easy with wizard guide

In general: Scripting capability. Exposing the AST to a scripting language would be a nice start, but in particular high-level stuff such as listing all local variables with inferred types and where in the AST they are accessed (written or read). And it would be *really* amazing if you could actually execute (parts of) the code from your scripts without having to copy-paste the code into gcc. In particular I'm thinking about string decryption/deobfuscation. Deobfucating strings with inlined decryption routines tends to be painful to automatically decrypt. Admittedly this is not trivial as you would have to allow the script to feed the code with any outside variables (such as globals, parameters and local variables not part of the executed code).

Code-editor-like features - Zoom, code-block folding, keyboard shortcuts; Perhaps colour-coding of arguments (e.g. a1, a2, a3 having different colours in a palette); Allow some aesthetic preferences to be configurable, e.g. variable naming conventions, highlighting/font palette...

Jetbrains-style refactorings. Like being able to extract variables or even functionblocks. For example, malware tend to have really long functions. I would love to be able to extract functions from that

---

- Is there any particular malware sample (or even a specific function), family, or functionality you would like to see as a decompiled output for the next survey?

I don't want a specific sample here, but I'd like to see more "real-world" code. E.g. network connections and file system interactions and stuff like that.

no

Nil

mirai

i've seen assembly code emitted by compiler that tries to align the stack to 16 bytes on every external calls, e.g.: push eax; push eax; push address_of_str1; push address_of_str2; call strcmp; Even though strcmp only takes two argument. This can cause decompiler to show redundant arguments or worse, emitting many redundant variables. Would like to see how this can be effectively handled.

calling functions within another functions

nil

nil

nope

since it's an absolute classic you could go for variants of RC4 implementations. re-using some of the functions/code from the DREAM survey could be interesting to see how things evolved since then.

(None I can think of, can I get back to you if I find one?)

Mirai's C&C command parsing is fun!

II. Survey 2 - Complete Questionnaire and Results

A. Self Assessment

- Did you take part in the decompiler survey last year?

Answer	Count	%
Yes	17	38.6
No	27	61.4

- How much time (approx.) did you spend working with C code?

Answer	Count	%
None	2	4.6
A few hours	6	13.6
Several days	12	27.3
More than a year	10	22.7
On a regular basis	14	31.8

- How much time did you spend reversing executables before?

Answer	Count	%
None	3	6.8
A few hours	10	22.7
Several days	15	34.1
More than a year	8	18.2
On a regular basis	8	18.2

B. Comprehension

- Please consider the following decompiled function:

```
int sub_401000() {
  char i;
  char * var_0;
  char var_1;
  char var_3;
  int var_4;
  var_0 = malloc(16);
  var_1 = GetTickCount();
  GetSystemTime(lpSystemTime: var_0);
  for (i = 0; i < 8; i++) {
    var_3 = var_0[i];
    var_0[i] = ((unsigned int)(var_3 ^
↪   var_1) % 24 & 0xff) + 'a';
  }
  var_4 = var_0 + 8;
  *var_4 = 0x6d6f632e;
  *(var_4 + 4) = 0x0;
  switch((((int) var_1) % 0x8) - 0x1) {
  case 0:
  case 5:
    *var_4 = *var_4 ^ 0x6d001700;
    break;
  case 1:
    *var_4 = *var_4 ^ 0x190a0d00;
    break;
  case 4:
  case 6:
    *var_4 = *var_4 ^ 0x6d1a1100;
    break;
  }
  return var_0;
}
```

- What does this function return? (*Please note that you cannot change the answer after selecting it.*)

Answer	Count	%
No Idea	6	13.6
Manipulates the system time	8	18.2
Allows Virtual Machine Detection	1	2.3
Generates random domains	27	61.4
Encryptes memory regions	2	4.6

Obviuously does not encrypt or manipulate system time. Function does not obtain enough information to do virtual machine detection. ASCII printable text in result most likely points to generation of random domains.

it does something with the system time apparently. Maybe a repeatedly called generator function?

for loop changes first 8 bytes of output, switch statement last 8 bytes. Both changes depend on system time.

It returns some random (not really) values generated from system time and tick count. Probably uses it to generate random domains.

DGA

I don't know how you can dereferance 'var_4', but I guess it's x86?

Missing cast when var4 is initialized.

Returns a string with eight chars + tld and memory ends with 0x00; e.g.: ? = random printable char; ????????.com; ????????.to; ????????.net; ????????.ru

To replace the code with RUST since it's far better to use RUST.

First I thought it might be used to generate "random" looking domains in order to communicate with a c&c center of a botnet or something similar, but since the generator results depend on the system up time it can not be in sync with another instance of that generator on another machine without communicating the tickcount which would defeat the purpose.

var_0 contains the result. Translates first 8 characters of var_0 from GetSystemTime to presumably an alphabetic string using GetTickCount(). Sets 9th character to some kind of checksum value based on GetTickCount(). Sets 10th character to null byte (presumably string terminator).

seems to generate a random string of characters using the time

getting system time and performing multiple and and or operation

Compile the code and run

Buffer Overflow; var_0[i]: 0-23; 0x00 - 0xE; Switch(0-6)

Look like always return "a"

The char * seems to be a ‘SYSTEMTIME‘struct. The loop manipulates the first four WORDS: year, month, week, day byte wise. The var_4 int points to hour and minute and changes this values in the switch statement. The allocated pointer to the time struct is returned. Since this does not really change the system time and the values are out of valid time values, I have no idea.

- Please type all utilized top-level-domains (TLDs)

Answer	Count	%
.com, .to, .net, .ru	29	65.9
.com, .to, .net	1	2.3
.idk	1	2.3
Rust	1	2.3
.com, .de, .sg, .org,.edu, .int, .gov, .ney, .mil, .my, .ae, .af, .ag	1	2.3
.to, .ten, .ur, .moc	1	2.3
.com, .org, .gov, .net	1	2.3
.ru, .net, .com	1	2.3
.a	2	4.6
.com, .ru, .neu	1	2.3
no answer	5	11.4

- Which Top-Level domain will be utilized the most?

Answer	Count	%
.com	32	72.7
.to	1	2.3
.moc	2	4.6
.ru	1	2.3
.a	1	2.3
no answer	7	15.9

- Which of these second-level domains can potentially be generated by the function?

count %	No	Yes	no answer
simpmpfp	3 6.8	31 70.5	10 22.7
xfbcbcic	3 6.8	31 70.5	10 22.7
facebook	10 22.7	25 56.8	9 20.5
squzuzfz	21 47.7	13 29.5	10 22.7
rlpmpmgmjdh	33 75.0	2 4.5	9 20.5

- How many domains can potentially be generated by this function per day?

Answer	Count	%
0	3	6.8
1	3	6.8
4	1	2.3
6	1	2.3
16	1	2.3
24	5	11.4
256	3	6.8
768	1	2.3
3600	1	2.3
86400	2	4.6
345600	1	2.3
2941884	1	2.3
43200000	1	2.3
86400000	10	22.7
259200000	1	2.3
463416696	1	2.3
764411904	1	2.3
2073600000	1	2.3
4294967296	1	2.3
110075314176	1	2.3
152587890625	0	0.0
440301256704	3	6.8
no answer	1	2.3

C. Comparison

- In this part, we are going to show you how the decompiled function from before is decompiled by other popular decompilers.
Hex-Rays:

```
LPSYSTEMTIME sub_401000()
{
LPSYSTEMTIME lpSystemTime; // [esp+4h]
  [ebp-Ch]
WORD *p_wHour; // [esp+8h] [ebp-8h]
char TickCount; // [esp+Eh] [ebp-2h]
char i; // [esp+Fh] [ebp-1h]

lpSystemTime = (LPSYSTEMTIME)malloc(16);
TickCount = GetTickCount();
GetSystemTime(lpSystemTime);
for ( i = 0; i < 8; ++i )
    *((_BYTE *)&lpSystemTime->wYear + i) =
  (unsigned __int8)(TickCount ^ *((_BYTE
  *)&lpSystemTime->wYear + i)) % 24 + 97;
p_wHour = &lpSystemTime->wHour;
*(_DWORD *)&lpSystemTime->wHour=1836016430;
*(_DWORD *)&lpSystemTime->wSecond = 0;
switch ( TickCount % 8 )
{
    case 1:
    case 6:
    *(_DWORD *)p_wHour ^= 0x6D001700u;
    break;
    case 2:
    *(_DWORD *)p_wHour ^= 0x190A0D00u;
    break;
    case 5:
    case 7:
    *(_DWORD *)p_wHour ^= 0x6D1A1100u;
    break;
    default:
    return lpSystemTime;
```

```
}
return lpSystemTime;
}
```

Ghidra:

```
LPSYSTEMTIME FUN_00401000(void)
{
LPSYSTEMTIME lpSystemTime;
DWORD DVar1;
uint *puVar2;
uint uVar3;
char local_5;

lpSystemTime = (LPSYSTEMTIME)malloc(0x10);
DVar1 = GetTickCount();
GetSystemTime(lpSystemTime);
local_5 = '\0';
while (local_5 < '\b') {
    *(byte *)((int)&lpSystemTime->wYear +
    (int)local_5) =
    (byte)(*(byte
    *)((int)&lpSystemTime->wYear +
    (int)local_5) ^ (byte)DVar1) % 0x18 +
    0x61;
    local_5 = local_5 + '\x01';
}
puVar2 = (uint *)&lpSystemTime->wHour;
*puVar2 = 0x6d6f632e;
*(undefined4 *)&lpSystemTime->wSecond = 0;
uVar3 = (int)(char)(byte)DVar1 & 0x80000007;
if ((int)uVar3 < 0) {
    uVar3 = (uVar3 - 1 | 0xfffffff8) + 1;
}
switch(uVar3) {
case 1:
case 6:
    *puVar2 = *puVar2 ^ 0x6d001700;
    break;
case 2:
    *puVar2 = *puVar2 ^ 0x190a0d00;
    break;
case 5:
case 7:
    *puVar2 = *puVar2 ^ 0x6d1a1100;
}
return lpSystemTime;
}
```

Binary Ninja:

```
int32_t sub_401000()

  int32_t eax = malloc(0x10)
  char eax_1 = GetTickCount()
  GetSystemTime(lpSystemTime: eax)
  char var_5 = 0
  while (sx.d(var_5) s< 8)
  int32_t eax_6
  int32_t edx_4
  edx_4:eax_6 = sx.q(zx.d(*(eax +
  sx.d(var_5)) ^ eax_1))
  *(eax + sx.d(var_5)) =
  (mods.dp.d(edx_4:eax_6, 0x18)).b + 0x61
  int32_t ecx_1
  ecx_1.b = var_5
  ecx_1.b = ecx_1.b + 1
  var_5 = ecx_1.b
  void* edx_8 = eax + 8
  *(edx_8 + 0) = 0x6d6f632e
  *(edx_8 + 4) = 0
  int32_t edx_11 = sx.d(eax_1) & 0x80000007
  if (edx_11 s< 0)
  edx_11 = ((edx_11 - 1) | 0xfffffff8) + 1
  int32_t eax_9 = edx_11 - 1
  if (eax_9 u<= 6)
  if (eax_9 == 4 || eax_9 == 6)
      *(edx_8 + 0) = *(edx_8 + 0) ^
  0x6d1a1100
  if (eax_9 == 0 || eax_9 == 5)
      *(edx_8 + 0) = *(edx_8 + 0) ^
  0x6d001700
  if (eax_9 == 1)
      *(edx_8 + 0) = *(edx_8 + 0) ^
  0x190a0d00
  return eax
```

dewolf:

```
int sub_401000() {
char i;
char * var_0;
char var_1;
char var_3;
int var_4;
var_0 = malloc(16);
var_1 = GetTickCount();
GetSystemTime(lpSystemTime: var_0);
for (i = 0; i < 8; i++) {
    var_3 = var_0[i];
    var_0[i] = ((unsigned int)(var_3 ^
    var_1) % 24 & 0xff) + 'a';
}
var_4 = var_0 + 8;
*var_4 = 0x6d6f632e;
*(var_4 + 4) = 0x0;
switch((((int) var_1) % 0x8) - 0x1) {
case 0:
case 5:
    *var_4 = *var_4 ^ 0x6d001700;
    break;
case 1:
    *var_4 = *var_4 ^ 0x190a0d00;
    break;
case 4:
case 6:
    *var_4 = *var_4 ^ 0x6d1a1100;
    break;
}
return var_0;
}
```

Snowman:

```
struct s0* fun_401000() {
struct s0* eax1;
struct s0* v2;
signed char al3;
signed char v4;
signed char v5;
uint32_t edx6;
int32_t edx7;
uint32_t edx8;
eax1 = reinterpret_cast<struct
    s0*>(malloc(16));
v2 = eax1;
al3 = reinterpret_cast<signed
    char>(GetTickCount());
v4 = al3;
GetSystemTime(v2);
v5 = 0;
while (static_cast<int32_t>(v5) < 8) {
    edx6 = static_cast<uint32_t>(
    *reinterpret_cast<unsigned
    char*>(reinterpret_cast<int32_t>(v2) +
    v5)) ^ reinterpret_cast<uint32_t>(
    static_cast<int32_t>(v4));
```

```
    __asm__("cdq ");
    edx7 = reinterpret_cast<int32_t>(
    static_cast<uint32_t>(
    *reinterpret_cast<unsigned
    char*>(&edx6))) % 24 + 97;
    *reinterpret_cast<signed
    char*>(reinterpret_cast<int32_t>(v2) +
    v5) = *reinterpret_cast<signed
    char*>(&edx7);
    v5 = reinterpret_cast<signed char>(v5 +
    1);
}
v2->f8 = 0x6d6f632e;
(&v2->f8)[1] = 0;
edx8 = reinterpret_cast<uint32_t>(
    static_cast<int32_t>(v4)) & 0x80000007;
if (__intrinsic()) {
    edx8 = (edx8 - 1 | 0xfffffff8) + 1;
}
switch (edx8 - 1) {
case 0:
case 5:
    v2->f8 = v2->f8 ^ 0x6d001700;
    break;
case 1:
    v2->f8 = v2->f8 ^ 0x190a0d00;
    break;
case 4:
case 6:
    v2->f8 = v2->f8 ^ 0x6d1a1100;
case 2:
case 3:
    goto 0x40112d;
}
return v2;
}
```

Retdec:

```
int32_t function_401000(int32_t a1) {
int32_t * mem = malloc(16); // 0x401008
int32_t result = (int32_t)mem; // 0x401008
int32_t v1 = GetTickCount(); // 0x401014
GetSystemTime((struct _SYSTEMTIME *)mem);
for (int32_t i = 0; i < 8; i++) {
    char * v2 = (char *)(i + result); //
    0x401046
    *v2 = (*v2 ^ (char)v1) % 24 + 97;
}
int32_t * v3 = (int32_t *)(result + 8); //
    0x40107e
*v3 = 0x6d6f632e;
*(int32_t *)(result + 12) = 0;
int32_t v4 = 0x1000000 * v1 / 0x1000000; //
    0x401097
int32_t v5 = v4 & -0x7ffffff9; // 0x40109b
int32_t v6 = v5; // 0x4010a1
if (v5 < 0) {
    // 0x4010a3
    v6 = (v4 + 7 | -8) + 1;
}
// 0x4010a8
g18 = v6 - 1;
switch (v6) {
    case 1: {
    }
    case 6: {
    // 0x4010e8
    *v3 = *v3 ^ 0x6d001700;
    // break -> 0x40112d
    break;
    }
    case 2: {
    // 0x40110c
    *v3 = *v3 ^ 0x190a0d00;
    // break -> 0x40112d
    break;
    }
    case 5: {
    }
    case 7: {
    // 0x4010c4
    *v3 = *v3 ^ 0x6d1a1100;
    // break -> 0x40112d
    break;
    }
}
// 0x40112d
return result;
}
```

JEB:

```
LPSYSTEMTIME sub_401000() {
LPSYSTEMTIME lpSystemTime =
    (LPSYSTEMTIME)malloc(16);
DWORD v0 = GetTickCount();
char v1 = (unsigned char)v0;
GetSystemTime(lpSystemTime);
for(char i = 0; i < 8; ++i) {
    *(char*)((int)i + (int)lpSystemTime) =
    (unsigned char)((unsigned
    int)*(char*)((int)i + (int)lpSystemTime)
    ^ (int)v1) % 24 + 97;
}
int* ptr0 = (int*)(lpSystemTime + 4);
*ptr0 = 1836016430;
*(ptr0 + 1) = 0;
int v2 = (int)v1 & 0x80000007;
if(v2 < 0) {
    v2 = ((v2 - 1) | 0xfffffff8) + 1;
}
unsigned int v3 = (unsigned int)(v2 - 1);
switch((unsigned int)(v2 - 1)) {
    case 1: {
    *ptr0 = *ptr0 ^ 0x190a0d00;
    return lpSystemTime;
    }
    case 0:
    case 5: {
    *ptr0 = *ptr0 ^ 0x6d001700;
    return lpSystemTime;
    }
    case 4:
    case 6: {
    *ptr0 = *ptr0 ^ 0x6d1a1100;
    return lpSystemTime;
    }
    default: {
    return lpSystemTime;
    }
}
}
```

- Which aspects in the decompiled codes above are especially favorable to you?

---

hexrays: Automatic renaming of some variables, creation of structs. ghira: As above. dewolf: Seems to be the cleanest. Likely not as ”aggressive” in analysis, but that is favourable in many situations. snowman: Useful information, but with drawbacks. retdec: likewise.

hexrays: type info; ghidra: type info; binaryninja: -; dewolf: looks simple and uncluttered; snowman: -; retdec: -; jeb: type info

binaryninja: the switch-statement is represented as if-statement. Values, especially numbers, are shown as hexdecimals. hexrays: de-obfuscation of variables, that's neat. ghidra: type-casting is shown. dewolf: proper for-loop. type-casting shown. Values are well very represented and only then shown as characters where appropriate. snowman: -; retdec: usage of indentation; jeb: the best switch-statement of all shown decompilations: no unnecessary brackets, a default statement, ... $->$ allows a clear track of flow here. Also, good type deduction for the systemTime.

hexray, snowman, retdec: addresses, registers as variable names, or in comments (guess I prefer it as comment as I can rename vars); dewolf, retdec: short lines

hex-rays, cleanest code with the min amt of variables used. variable names makes more sense.

Using strings resolved e.g: lpSystemTime$->$wYear in line 16

hexrays, ghidra: variable name guessing, including struct members; jeb: variable name guessing

Hexrays: structure aware (e.g. LPSYSTEMTIME), renames variables based on what is known from the windows APIs (e.g. GetTickCount()); * default case for switch statement: hexrays, jeb; * struct resolution: e.g. &lpSystemTime-¿wSecond: hexrays, ghidra, (jeb); * correct variable type cast, e.g. LPSYSTEMTIME lpSystemTime: hexrays, ghidra, jeb; * DGA algorithm easily readable in: dewolf; * switch case possibilities: TickCount % 8 in hexrays,

Best one is not surprisingly IDA it seems both very accurate and quite comprehensible. Only thing is that both '1836016430' and '97' should probably be in hex, but there probably is an option to default to hex. Ghidra struggles with the '%' alot but otherwise it seems pretty much on par. Dewolf and jeb have rather basic but pretty clean code. They are not trying to be as smart (not trying to infer the LPSYSTEMTIME) which kinda helps in this case as there is an intentional type confusion.

hexrays, ghidra, retdec, jeb: - recognize LPSYSTEMTIME type $->$ also struct fields recognizable (e.g. lpSystemTime$->$wYear) (had to look this up via docs.microsoft.com); generally: viewer lines to read is (mostly) better

Heyrays: Automated annotation of the stack layout; Heyrays, Ghidra, binaryninja, dewolf, jeb: Annoate name of GetSystemTime parameter; dewolf: Very clean decompilation. Proper placement of var_0[i] instead of pointer castings

HexRays: Reasonable variable names and datatype inference, accurate decompilation, and no redundant typecasts. Optimization to detect modulo arithmetic. Looks the 'cleanest'; Ghidra: Correctly optimizes away the default case in switch statement. Few redundant typecasts. Quite clean; Dewolf: Similar to hexrays, slightly less accurate, but optimizes away the redundant default case in the switch statement. Clean look.; JEB: Casts the malloc to LPSYSTEMTIME correctly. Relatively clean look. retdec: The comments with addresses is unnecessary bloat

hexrays: 1) Automatically inferring of the return type of windows API. 2) Renaming of the variables based on the inferred types. dewolf: 1) Representing the 0x61 constant as 'a'. 2) The for loop that sets the 2nd level domain is much more comprehensible compared to all other because it has less noisy type casting.

hexrays is the cleanest.

In HexRays, it detects the LPSYSTEMTIME struct, and the rest of the code interacting with it is in struct form. Ghidra does the same, albeit in a more standard C way.

hexrays

dewolf: - nice and clean switch case structure; - good variable definition

dewolf

dewolf looks very clean and is easy to read

hexrays looks also very clean, you get an idea of the stack trough the comments of esp and ebp; if I would ignore the comments in retdec, its also good to read; jeb is ok, nothing too bad, but would not prefer about bn I like, that the variable names enumerate the used registers

It really helps to see the last part of the code as a switch case instead of nested if-statements. While it might be useful to see all the typecasts / types explicitly I like it more to see more "clean" or shorter statements. Therefore I liked the way the while/for loop was presented by all the decompilers except for snowman. I liked the most: dewolf / retdec / jeb (no particular order)

Hexrays: The ability to break down the structure of lpSystemTime without requiring me to search for the documentation -$->$ good vraiable naming.

hexrays, ghidra: meaningful variable and field names; dewolf: appears closest to actual written code

- useful comments (hexrays); - recognized mod 24 (as decimal int) and the "a" as char (dewolf)

dewolf has the most accurate decompiled codes.

looks more like C code

Hexrays: Naming of the buffer variable with parameter and applying type is usually helpful, although not so much in this case. Code is concise. Switch statement is the simplest amongst the options. Ghidra: Naming of the buffer variable with parameter and applying type is usually helpful. Reasonably concise decompilation. Binary Ninja: n/a; Dewolf: Basic but fairly clean and concise decompilation. Not applying the lpSystemTime struct happens to be helpful here, making the loop simpler to read. Switch parameter is simpler than some of the others. Snowman: n/a; Retdec: Loop is easy to read, not applying the struct

happens to be helpful as well. Jeb: Mostly simple and clean decompilation.

You can compile hexrays, ghidra and dewolf code easily to dynamically debug them .

hexrays / ghidra: + STRUCT TYPE; + var naming; + easy switch

hexrays: types and struckts, variable names

At first glance, dewolf will be most favourable because the previous exercise has required the need to go through the same set of information. However if an unbiased mind is to review the codes from the start again, Ghidra would be especially favourable due to the consistency of the values being primarily resolved to hex values.

hexrays: naming of variables, type hints, clean switch case, nice for loop; dewolf: very nice for loop, somewhat clean switch case, not resolving lpSystemTime fields (since they are misused anyway)

hexrays has better variable naming

dewolf decompiled the codes in a cleaner and more understandable manner compared to the other decompilers.

Hexrays managed to provide meaningful name(s) for variables, e.g. p_wHour, TickCount, wHour, wSecond

hexrays, retdec

using switch and case for options

Switch and choice options

I like the variable naming of hexrays as it already contains some meaning. I like that binaryninja, retdec, and jeb combine variable declaration and definition where possible. Especially jeb has only a short variable declaration/definition section at the beginning and moves the definition of loop variables into the loop. This helps keeping the code compact and doesn't force me to remember lots of variables. In general, I like that hexrays and ghidra seem to know the structure of LPSYSTEMTIME and use its field accessors ($->$wYear, $->$wHour). Although in this particular example this doesn't really help or make sense, in general I think this would be useful. This is my favorite loop header "for(char i = 0; i $<$ 8; ++i) {" (jeb). The for loop is definitely better that all while loops. From an overall perspective, dewolfs output looks "most natural" to me. I. e. this could be code as written by a human. The output is rather uncluttered (no cast chains, magic numbers, or special notations).

hexrays: struct resolution implies reuse of fields for other purpose (wHour,wMin for TLD), inconsistent number display (dec, hex); ghidra: unlucky comparions (local_5 $<$'\b'), but somewhat consistent use of hex numbers. logic expressions are a bit harder to understand (bitmasking, ...); binja: explicit registers instead of abstraction makes it harder to comprehend. dewolf: looks clean, but inconsistent number display (dec, hex); snowman: too much casting, no simplification through resolved aliasing, inconsistent number display (dec, hex); retdec: VAs as comments clutter the code, this code be solved better with highlighting in a UI; jeb: assignment GetTickCount$->$DWORD$->$char makes it somewhat more obvious there are only 256 values possible

The correct typing of the LPSYSTEMTIME (hexrays, ghidra, jeb) variable in this case is not helpful, in other cases it may be though. hexrays: switch statement seems clearer; dewolf, retdec: clean layout

ease of understanding, jeb.

dwolf code make it easy to analyze

- Which aspects of the presented decompiled functions do you deem unhelpful or hamper your understanding of the code?

---

ghidra: Not as good at handling loops. (e.g, line 14) binary ninja: Nasty way of naming variables. snowman: type-casting is useful when deep diving, but for quick glances, it only adds to clutter the viewport. (hotkey to toggle?) retdec: Addresses on comments only add to the clutter. jeb: Type-casting seems off

hexrays: I don't care about registers ghidra: numbers represented as char strings, all those () and * and shit binaryninja: I don't care about registers, not even in variable names dewolf: - snowman: are we using registers as variable names again? also, what is this casting nonsense retdec: What are those comments about jeb: all those () and *

binaryninja: the for-loop is shown as while loop. Casts are sometimes difficult to read. hexrays: the default in the switch-statement bugs me and it is redundant. Usage of decimal values seems inappropriate here. ghidra: strange naming of variables. local_5 is shown as character, which confuses more than it helps. dewolf: no indentation in switch-statement snowman: not clear what intrinsic does. Casts clutter the lines. Very bad naming convention (variable names should be longer) retdec: those comments... really? Fall-through cases are less clear with these brackets jeb: too many casts (line 8).

I lack the experience to say what I see as unhelpful other than the opposit of question 1

binja's (and snowman) variables being named after their registers is a huge pain to read, I also have no idea what is sx.d etc.

binaryninja, snowman: very basic disassembly, looking at their assembly codes might be more useful.

In line 19, &lpSystemTime$->$wHour replaces the address calculation. May be confusing.

ghidra: Converts the variable to a char (local_5) in

the while loop. Not structure-aware as well (e.g. &lpSystemTime− >wYear + (int)local_5): have to manually calculate the offset)

* weird type castings that clutter the code: ghidra, snowman (!), jeb * IR: Binja IR is okayish but doesnt even come close to fully decompiled code like in hexrays or jeb * retdec code formatting and comments (??) * weird ass loop condition: ghidra (while local_5 < '\b', the heck is that..

1) The crazy C++ cast Situation in snowman, also not understanding a cdq before a mod is kinda weird. 2) Binja is not really a decompiler its more of a visualization of their bytecode. 3) Retdec is (I think) a command line tool so the comments at the end of the line make sense, but are really verbose. They would probably be less annoying if they were aligned and way to the right. 4) In Jeb it seems like some of the Casts should be able to get folded. '(unsigned int *)(char *) = (unsinged int *)' and '(char *)(int + (int)ptr) = (char *)ptr + int'

snowman: - uses C++ style casts which is a bit strange - inline assembly (e.g. cdq) should be avoided − > use casts for that binaryninja: - poor type recognition, also sx.q, zx.d etc. not nice to read if you don't know what it means hexrays vs. dewolf: - TickCount % 8 vs (((int) var_1) % 0x8) - 0x1) − > this is IMHO not the same condition... but +1 for the modulo vs OR representation ghidra: - loop recognition and variable usage while (local_5 < '\b') { *(byte *)((int)&lpSystemTime− >wYear + (int)local_5) = (byte)(*(byte *)((int)&lpSystemTime− >wYear + (int)local_5) ^(byte)DVar1) % 0x18 + 0x61; local_5 = local_5 + '\x01'; } pretty messed up for a simple for loop that counts from 0 to 7 :D binaryninja: - if vs. switch... switch is better to read hexrays: - has a redundant default: return lpSystemTime − > +1 for dewolf snowman: - goto 0x40112d; − > no idea where that is ^^- C++ casts clutter everything and have no real "advantage" if this isn't a C++ program dewolf: - i don't like the GetSystemTime(lpSystemTime: var_0); lpSystemTime jeb: - inconsistent use of hex representations e.g. *ptr0 = 1836016430; // decimal *(ptr0 + 1) = 0; // decimal int v2 = (int)v1 & 0x80000007; // hex ghidra and others that do not use the % operator: - this is way to complicated to be easily recognized: uVar3 = (int)(char)(byte)DVar1 & 0x80000007; if ((int)uVar3 < 0) { uVar3 = (uVar3 - 1 |0xfffffff8) + 1; } switch(uVar3) {

Heyrays: Integers "randomly" being in decimal Heyrays, Ghidra: The default type of the 16 byte array being LPSYSTEMTIME and all access then being treated like it is that type Snowman: The casts hinder readability Ghidra, binaryninja, snowman : No for loop detection Ghidra, binaryninja, snowman, retdec, jeb: Don't understand "TickCount % 8" and produce hard to comprehend code for it snowman:__asm__("cdq "); and gotos binaryninja: Too similar naming scheme and things like sx.q and mods.dp.d(edx_4:eax_6, 0x18) dewolf: "- 0x1" on the switch which causes the -1 case which is a bit more confusing compared to IDA Ghidra: Random usage of '\b' and '\x01' instead of hex numbers

binja: Things like "*(eax + sx.d(var_5)) = (mods.dp.d(edx_4:eax_6, 0x18)).b + 0x61" is not C and so it's hard to understand

Ghidra: Redundant casts calculating uVar3. No mod 8 optimization. Binary Ninja: sx.q, zx.d, mods.dp.d etc. are alien syntax to people used to C. No mod 8 optimization. Dewolf: Missing typecast on var4 made the surrounding code very confusing Snowman: reinterpret_cast / static_cast bloat the screen and hide the other more important code. Unimplemented instruction cdq. goto without a matching label. No mod 8 optimization. JEB: A few weird typecasts on the TickCount

Too much casting, and no API type information.

In BinaryNinja, the variable names are chosen by the register they were found in, which makes reading the code far harder than with other decompilers. In Dewolf, the type decompilation was far off from other decompilers, making analysis difficult. In Snowman, the use of C++ type casts makes the block of code very cluttered and difficult to understand.

the comments beside the local variable declaration sometimes are not helpful

dewolf: − > + 'a'; irritated me at first − > switch((((int) var_1) % 0x8) - 0x1) < − why "- 0x1" if it is only modulo?

.

snowman has this string "reinterpret_cast<signed char" that reduces my attention on the important stuff I dont know exactly why there are addresses in the comments of retdec and could not find anything helpful

It was quite easy to determine which domain is used most due to the way the switch case was presented. BinaryNinja does not present that part of the code as a switch case but rather nested if statements which is also fine but - at least to me - harder to wrap my head around it. The while loop from the snowman generated code looks so awful that it encurages me to ignore it the very second I see it.

retdec: comments corresponding to address of equivalent instruction bytes make it harder to process the main logic visually. snowman: excessive display of static_cast and reinterpret_cast

binaryninja: bad variable names snowman: too many casting functions, incomplete decompilation with asm snippets remaining

- for loop not recognized (ghidra, binaryninja, snowman) - comparison with char where it does not make sense (ghidra) - chained type cast (ghidra) - reinterpret_cast ??? (snowman) - switch statement not recognized (binaryninja) - goto (snowman) - unnecessary many returns (jeb)

In snowman, overuse of reinterpret_cast, static_cast can be confusing

Hexrays: Applying the struct type happens to make the loop harder to read, but it is not a huge issue. Ghidra: The excess casts can be slightly annoying. Bit manipulation of the switch argument is a bit difficult to parse. Binary Ninja: The IL is not particularly helpful in my opinion. Would prefer to read the assembly instead. Dewolf: No particular complaints. Switch could be slightly simplified (see Hexrays version) but is not a huge issue. Snowman: Excessive casting along with verbose C++-like casting syntax is off-putting. The bit of inline assembly appearing in the loop is not helpful. Bit manipulation of switch argument is a bit difficult to parse. I have no idea what __intrinsic() is supposed to mean. Retdec: Decompiled code could be more concise. Arithmetic and bit manipulation of switch argument is difficult to parse. Jeb: Casting in loop makes it slightly harder to read. Bit manipulation of switch argument is a bit difficult to parse.

binaryninja decompiler seems to be very convoluted

snowman: casts are confusing ghidra/binaryninja: everything

snowman decompiler presented a decompiled code that was too messy to be picked up at first glance, binarywolf decomipler would have been useful for memory diving, but not when determining the use of said code in example The additional "*(undefined4 *)" presented by Ghidra would have thrown me off the table if a prior exercise had not been done.

binaryninja: while loop with horrible body, no switch case snowman: verbose type casting retdec: many lines with useless arithmatic expressions

snowman has many casts

binaryninja and snowman seems to be too complicated to understand what has been decompiled

Going towards assembly (binaryninja), e.g. eax, edx does not seem to be helpful to understand the intent of the code

binaryninja, dewolf

Using cpu register name in the code

using cpu register name in the code

Basically, the things that make the code feel "unnatural": I don't like long cast chains. In jeb, there is the part "(unsigned char)((unsigned int)*(char*)((int)i + (int)lpSystemTime)". Here it's quite easy to miss the i in the middle. This gets worse when we use C++ casts as snowman does. This makes it really hard to find actual code in between the cast operations. I also don't like special notation as binaryninja uses. There we have this .b and .d and an sx function which are not really C. You need special knowledge to find out what they mean and I'd prefer not having to know this when reading the output of a debugger. Especially, since they don't help in understanding the code in this example. The same holds for some magic constants which are present in retdec for example (lines 13-14). I'd have a hard time understanding why they are there and what they do. Finally, I don't like that ghidra and binaryninja use hex notation also for small integers. In the malloc call I definitely prefer 16 over 0x10.

while probably most accurate, too fine-grained output (binja) is a bit contraproductive for fast analysis (might as well look at assembly then). too much casting clutters the output (snowman). automated application of structs and fields is usually very helpful, here it's potentially slightly confusing because of the "unintended" use (hexrays,ghidra).

hexray: switching between hex und dec representation of numbers. ghidra: while loop with char limit mixed with hex increment hexrays, ghidra, jeb: in this case, the correct type LPSYSTEMTIME and the resulting byte casting in the loop binaryninja: weired style snowman: cluttering c++ casts dewolf, retdec: incorrect types sometimes retdec, jeb: brackets in switch cases may lead to false assumptions

require a longer time to understand the codes, snowman

Snowman and gihidra code can be hard to read

D. Feedback

- Thank you for getting this far! If you would like to leave us any feedback about the survey, please use the lines below to help us improve ourselves.
  Reminder: All replies in the complete study are going to be anonymized.

More code decompilation examples would be helpful.

NIL

all gucci

This is the first time I actually saw a DGA in C/Pseudo C :D So my results will not be biased be former experience of reversing DGAs. The usage of available API information (data types, names, struct defs) is one of the most important things to quickly understand what's going on. Complex expression with low level shifting, bitwise operations should be converted to conditions that are easier to understand (e.g., modulo vs ... |...) dewolf + API information would be pretty close to hexrays IMHO The length of the survey was very manageable and thus was quite entertaining ;)

Give me more Rust

After taking a break and resuming the survey I was able to change my first answer because it was not selected anymore.

very interesting survey

The code editor did not let me select-to-copy, that was anoying.

It is not fair to the other decompiles if one of the contestant's output is used as the precursor exercise.

nil

No feedback at the moment

Not at the moment

This was more fun that the last one! ;-) When asked again which aspect of compiler output I like, it would be cool to have the ability to comment inline in the output as e. g. in a code review. But I'm not sure if this is technically possible. Also, it would be nice to have the different decompiler outputs side-by-side so that I don't have to click through the tabs. Can we get the solutions?

Nil

Thank you very much for your participation! List of utilized decompiler versions:

- JEB: 4.0-beta.3.202103090424
- Snowman: 0.1.3
- retdec: 4.0
- IDA: 7.6
- Ghidra: 9.2.2
- binaryninja: 2.3.2720-dev

III. Survey 3 - Complete Questionnaire and Results

A. Self Assessment

The questions of this group are dedicated to self assessment to allow us to better understand trends in the answers of the users.

- Did you take part in one of the previous decompiler surveys?

count %	Yes	No	no answer
Decompiler Survey 1 - October 2020	18 33.3	31 57.4	5 9.3
Decompiler Survey 2 - April 2021	24 44.4	26 48.1	5 9.3

- How much time (approx.) did you spend working with C code?

Answer	Count	%
None	0	0.0
A few hours	2	3.7
Several days	14	25.9
More than a year	25	46.3
On a regular basis	13	24.1

- How much time did you spend reversing executables before?

Answer	Count	%
None	2	3.7
A few hours	1	20.4
Several days	18	33.2
More than a year	12	22.2
On a regular basis	11	20.4

B. Decompiler Comparison

- In this part, we will show you code snippets that are decompiled with different decompilers.
  Clove (Hex-Rays):

```
_int64 sub_401C3D()
{
  char s[8]; // [rsp+0h] [rbp-60h] BYREF
  __int64 v2; // [rsp+8h] [rbp-58h]
  __int64 v3; // [rsp+10h] [rbp-50h]
  __int64 v4; // [rsp+18h] [rbp-48h]
  __int64 v5; // [rsp+20h] [rbp-40h]
  __int64 v6; // [rsp+28h] [rbp-38h]
  __int64 v7; // [rsp+30h] [rbp-30h]
  __int64 v8; // [rsp+38h] [rbp-28h]
  char v9; // [rsp+40h] [rbp-20h]
  __int64 v10; // [rsp+50h] [rbp-10h] BYREF
  int v11; // [rsp+58h] [rbp-8h]
  unsigned int v12; // [rsp+5Ch] [rbp-4h]

  *(_QWORD *)s = 0LL;
  v2 = 0LL;
  v3 = 0LL;
  v4 = 0LL;
  v5 = 0LL;
```

```
  v6 = 0LL;
  v7 = 0LL;
  v8 = 0LL;
  v9 = 0;
  printf("Enter any binary number: ");
  __isoc99_scanf("%lld", &v10);
  v12 = v10;
  while ( v10 > 0 )
  {
  v11 = v10 % 10000;
  if ( v11 == 1111 )
  {
    *(_WORD *)&s[strlen(s)] = 70;
  }
  else if ( v11 <= 1111 )
  {
    if ( v11 == 1110 )
    {
    *(_WORD *)&s[strlen(s)] = 69;
    }
    else if ( v11 == 1101 )
    {
    *(_WORD *)&s[strlen(s)] = 68;
    }
    else if ( v11 <= 1101 )
    {
    if ( v11 == 1100 )
    {
      *(_WORD *)&s[strlen(s)] = 67;
    }
    else if ( v11 == 1011 )
    {
      *(_WORD *)&s[strlen(s)] = 66;
    }
    else if ( v11 <= 1011 )
    {
      if ( v11 == 1010 )
      {
      *(_WORD *)&s[strlen(s)] = 65;
      }
      else if ( v11 == 1001 )
      {
      *(_WORD *)&s[strlen(s)] = 57;
      }
      else if ( v11 <= 1001 )
      {
      if ( v11 == 1000 )
      {
        *(_WORD *)&s[strlen(s)] = 56;
      }
      else if ( v11 == 111 )
      {
        *(_WORD *)&s[strlen(s)] = 55;
      }
      else if ( v11 <= 111 )
      {
        if ( v11 == 110 )
        {
        *(_WORD *)&s[strlen(s)] = 54;
        }
        else if ( v11 == 101 )
        {
        *(_WORD *)&s[strlen(s)] = 53;
        }
        else if ( v11 <= 101 )
        {
        if ( v11 == 100 )
        {
          *(_WORD *)&s[strlen(s)] = 52;
        }
        else if ( v11 == 11 )
        {
          *(_WORD *)&s[strlen(s)] = 51;
        }
        else if ( v11 <= 11 )
        {
          if ( v11 == 10 )
          {
          *(_WORD *)&s[strlen(s)] = 50;
          }
          else if ( v11 )
          {
          if ( v11 == 1 )
            *(_WORD *)&s[strlen(s)] = 49;
          }
          else
          {
          *(_WORD *)&s[strlen(s)] = 48;
          }
        }
        }
      }
      }
    }
    }
  }
  v10 /= 10000LL;
  }
  printf("Binary number: %lld\\n", v12);
  printf("Hexadecimal number: %s", s);
  return 0LL;
}
```

Cumin (Ghidra):

```
undefined8 FUN_00401c3d(void)

{
size_t sVar1;
undefined8 local_68;
undefined8 local_60;
undefined8 local_58;
undefined8 local_50;
undefined8 local_48;
undefined8 local_40;
undefined8 local_38;
undefined8 local_30;
undefined local_28;
long local_18;
int local_10;
uint local_c;

local_68 = 0;
local_60 = 0;
local_58 = 0;
local_50 = 0;
local_48 = 0;
local_40 = 0;
local_38 = 0;
local_30 = 0;
local_28 = 0;
printf("Enter any binary number: ");
__isoc99_scanf(&DAT_00403259,&local_18);
local_c = (uint)local_18;
for(; 0 < local_18; local_18 = local_18 / 10000){
  local_10 = (int)local_18 + (int)(local_18 /
↪  10000) * -10000;
  if (local_10 == 0x457) {
    sVar1 = strlen((char *)&local_68);
    *(undefined2 *)((long)&local_68 + sVar1) =
↪  0x46;
  }
  else {
    if (local_10 < 0x458) {
      if (local_10 == 0x456) {
        sVar1 = strlen((char *)&local_68);
        *(undefined2 *)((long)&local_68 + sVar1)
↪  = 0x45;
      }
      else {
        if (local_10 < 0x457) {
          if (local_10 == 0x44d) {
            sVar1 = strlen((char *)&local_68);
            *(undefined2 *)((long)&local_68 +
↪  sVar1) = 0x44;
          }
          else {
            if (local_10 < 0x44e) {
              if (local_10 == 0x44c) {
                sVar1 = strlen((char
↪  *)&local_68);
                *(undefined2 *)((long)&local_68 +
↪  sVar1) = 0x43;
              }
              else {
                if (local_10 < 0x44d) {
                  if (local_10 == 0x3f3) {
```

```
                    sVar1 = strlen((char
↪ *)&local_68);
                    *(undefined2
↪ *)((long)&local_68 + sVar1) = 0x42;
                  }
                  else {
                    if (local_10 < 0x3f4) {
                      if (local_10 == 0x3f2) {
                        sVar1 = strlen((char
↪ *)&local_68);
                        *(undefined2
↪ *)((long)&local_68 + sVar1) = 0x41;
                      }
                      else {
                        if (local_10 < 0x3f3) {
                          if (local_10 == 0x3e9)
↪ {
                            sVar1 = strlen((char
↪ *)&local_68);
                            *(undefined2
↪ *)((long)&local_68 + sVar1) = 0x39;
                          }
                          else {
                            if (local_10 < 0x3ea)
↪ {
                              if (local_10 ==
↪ 1000) {
                                sVar1 =
↪ strlen((char *)&local_68);
                                *(undefined2
↪ *)((long)&local_68 + sVar1) = 0x38;
                              }
                              else {
                                if (local_10 <
↪ 0x3e9) {
                                  if (local_10 ==
↪ 0x6f) {
                                    sVar1 =
↪ strlen((char *)&local_68);
                                    *(undefined2
↪ *)((long)&local_68 + sVar1) = 0x37;
                                  }
                                  else {
                                    if (local_10
↪ < 0x70) {
                                      if
↪ (local_10 == 0x6e) {
                                        sVar1 =
↪ strlen((char *)&local_68);

↪ *(undefined2 *)((long)&local_68 + sVar1) =
↪ 0x36;
                                      }
                                      else {
                                        if
↪ (local_10 < 0x6f) {
                                          if
↪ (local_10 == 0x65) {
                                            sVar1
↪ = strlen((char *)&local_68);

↪ *(undefined2 *)((long)&local_68 + sVar1) =
↪ 0x35;
                                          }
                                          else {
                                            if
↪ (local_10 < 0x66) {
                                              if
↪ (local_10 == 100) {

↪ sVar1 = strlen((char *)&local_68);

↪ *(undefined2 *)((long)&local_68 + sVar1) =
↪ 0x34;
                                              }

↪ else {

↪ if (local_10 < 0x65) {

↪ if (local_10 == 0xb) {

↪ sVar1 = strlen((char *)&local_68);

↪ *(undefined2 *)((long)&local_68 + sVar1) =

↪ 0x33;

↪ }

↪ else {

↪ if (local_10 < 0xc) {

↪ if (local_10 == 10) {

↪ sVar1 = strlen((char *)&local_68);

↪ *(undefined2 *)((long)&local_68 + sVar1) =

↪ 0x32;

↪ }

↪ else {

↪ if (local_10 < 0xb) {

↪ if (local_10 == 0) {

↪ sVar1 = strlen((char *)&local_68);

↪ *(undefined2 *)

↪ ((long)&local_68 + sVar1) = 0x30;

↪ }

↪ else {

↪ if (local_10 == 1) {

↪ sVar1 = strlen((char *)&local_68);

↪ *(undefined2 *)

↪ ((long)&local_68 + sVar1) = 0x31;

↪ }

↪ }

↪ }

↪ }

↪ }

↪ }
                                                                                                    }
                                                                                                  }
                                                                                                }
                                                                                              }
                                                                                            }
                                                                                          }
                                                                                        }
                                                                                      }
                                                                                    }
                                                                                  }
                                                                                }
                                                                              }
                                                                            }
                                                                          }
                                                                        }
                                                                      }
                                                                    }
                                                                  }
                                                                }
                                                              }
                                                            }
                                                          }
                                                        }
                                                      }
                                                    }
                                                  }
                                                }
                                              }
                                            }
                                          }
                                        }
                                      }
                                    }
                                  }
                                }
                              }
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
  printf("Binary number: %lld\\n",(ulong)local_c);
  printf("Hexadecimal number: %s",&local_68);
  return 0;
}
```

Chilli (dewolf):

```
long sub_401c3d() {
  size_t var_5;
  long i;
  long var_0;
  long var_2;
  long var_4;
  long * var_3;
  printf(/* format */ "Enter any binary number:
↪  ");
  var_3 = &var_0;
  __isoc99_scanf(/* format */ "%lld", var_3);
  var_2 = 0L;
  for (i = var_0; i > 0L; i /= 0x2710) {
    var_4 = i % 0x2710;
    switch(var_4) {
    case 0:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x30;
      break;
    case 1:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x31;
      break;
    case 10:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x32;
      break;
    case 11:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x33;
      break;
    case 100:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x34;
      break;
    case 101:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x35;
      break;
    case 110:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x36;
      break;
    case 111:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x37;
      break;
    case 0x3e8:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x38;
      break;
    case 0x3e9:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x39;
      break;
    case 0x3f2:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x41;
      break;
    case 0x3f3:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x42;
      break;
    case 0x44c:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x43;
      break;
    case 0x44d:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x44;
      break;
    case 0x456:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x45;
      break;
    case 0x457:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x46;
      break;
    }
  }
  printf(/* format */ "Binary number: %lld\\n",
↪  var_0 & 0xffffffff);
  var_3 = &var_2;
  printf(/* format */ "Hexadecimal number: %s",
↪  var_3);
  return 0L;
}
```

- Please rank the decompiled code presented.

Count	Ranks		
%	1	2	3
Clove	7 18.9	37 100.0	34 91.9
Cumin	3 8.1	8 21.6	10 27.0
Chilli	44 118.9	9 24.3	1 2.7

- Which aspects in the decompiled codes above is especially favorable to you?

---

switch case helps understanding conditions more easily

The switch-statement of Chilli is nice especially vs the ” indentation of Cumin. Clove is the only one that recoginizes that ’s’ is used as an array and not a single value.

Assuming they are logically equivalent (not heavily scrutinized), elimination of deep nesting of if/else is most favorable. Use of X[i] notation as opposed to *(X + i) might be preferable too. Identification of a for loop to replace a while loop is useful too.

1. Chilli: I prefer the constructed switch-case, as it reduces code complexity and improves readability 2. Clove: The RSP/RBP annotations for variable declarations are useful

overview: the less indentation the clearer it gets + cleverly reconstructing common code

Switch statements in chilli.

Note: Chilli/Clove are pretty close. Clove - It’s quite immediately obvious what each branch is doing, because of a) identifying s as being accessed as an array and b) directly indexing with strlen(s) makes it clear what is going on. That made it quite easy to make a good guess at what the function is doing quickly. Chilli - The switch statement makes the control flow much more obvious. Also, the lifting of

the loop conditions into the head of the for loop also helps make the purpose of the loop more immediately apparent. Cumin - The lift of the loop condition into the loop head is useful.

The close relation to the (probably) original C Code, with a switch statement instead of nested if-else blocks in ”Chilli”.

Switch cases in Chilli makes code much more readable.

readability: resolved types, shortness, levels of nesting

* Low if-nesting * Proper datatypes

variable name numbering

Chilli had cleaner and more human readable codes. This makes analysis much easier as compared the one seen in Cumin.

Propper switch-case extraction

The use of switch case to simplify the code.

Easier control-flow analysis (e.g. prefer switch statements rather than convoluted if-else statements)

Chilli: Use of switch statement. Use of for loop instead of while loop makes the condition more clear as well. The lack of type casting also makes it looks cleaner (although it would be useful for user to enable typecasting to be shown). Clove: Deduction of the ”s” as char array and then indexing it automatically is helpful.

Chilli is able to identify and decompile as a case-switch. Clove however provides good detail on declaration of variables which may be useful It would be good to have the option to display constants either as hex or decimal depending on preference.

Switch makes things easier to read for me, compared to repeatedly nested if-else.

- Appropriate use of switch-case - Omission of redundant checks - Omission of implied typecasts - Use of /=, which is quite succinct

readability of chilli, using switch case instead of if else resulting in less indentations and also easy to read which conditions will trigger which code path.

type information and avoiding deeply nested ifs

Data types, sufficient verbosity e.g. switch case statement not expanded into if-else

- using a switch statement - using decimal values - using a while loop - indexing arrays instead of using offsets - using less type casts

Use of switch-case instead of endless if/else if

The decompiled switch-case, and i find ”var_2” easier to read than v2. Also the local vars in chilli help the understanding I like that clove displays the stack offsets of the variables

the switch statement ist easier to understan

Chilli has the least amount of nesting. For this example less nesting means much better readability. Clove contains less typecasting than Cumin.

switch-case instead of nested if-else for loop instead of while loop

switch case instead of nested if else statements

Modulo detection in Clove/Chilli Concise loop expression in Clove/Cumin Inlined writes to s in Chilli

Easy to understand at a look

Using switch-case instead of nested if statements, as it can get very messy.

Readability, switch case statement instead of large nested conditional statements

Switch statement Array recognition Comments mapping variables back to memory locations

I like how chilli looks like a proper source code. However, i do trust a decompiler that does less analysis more.

I liked how Chilli was able to recover the switch statement. Clove, by not assigning the result of strlen(s) to an intermediate variable, was able to produce a decompilation with less visual clutter.

Neatness; able to quickly deduce there are switch statements and the values for each case.

Shorter expressions tend to be more easy to read. The switch-case statement appears to be more a lot clearer than the nested branches.

switches in place of a huge and nested if-else block makes the decompiled code more readable.

Chilli

switch case in chilli

Especially the ’Chilli’ decompiler, produced a clear and easy readable code, which wasn’t too nested due to the use of switch-case statements.

Chilli: The switch case is much easier to read than the nested if/ else/ else if structures. Chilli and Clove: The instructions before entering the loop where optically clearly distinguishable from the loop.

Clear structure based on the switch allows better comprehension than nested if statements.

easy to follow along structure

The use of switch case from Chili makes the code cleaner and readable.

deeply nested code is hard to read, hence I enjoyed the one with switch. Apart from that arithmetic was displayed a litter bit better in Chilli than Clove.

presenting as case, instead of massive loops. Showing the logic of the loop condition.

more human readable, closer to high-level programming code

Switch statements are easier to comprehend compared

to if-else

switch case

- Which aspects of the presented decompiled functions do you deem unhelpful?

---

hard to follow the heavily nested if/else, cumin with many unnecessary variables

Chilli could have realized 0x2710 is a power of 10. Which the other decompiler did, *wink wink*. Chilli is the only one that does not produce valid c. Cumin for some reason does not display the string in scanf and has redundant else branches increasing indentation.

THe use of undefined8 is not as helpful.

1. /* format */ annotations 2. Cumin did not resolve the constant format string used in the __isoc99_scanf call 3. Cumin introduces WAY too much cyclomatic complexity. I think it's hard to follow the control flow there

n/a

insanely deep nesting. weird for loop (cumin)

Clove - if/else branching is messy to read, it is not immediately obvious how or when the loop terminates until reading the end of the loop Chilli - The branch of each switch statement is a bit harder to understand, and might be considered somewhat misleading - consider the following: var_3 = &var_2; var_5 = strlen(var_3); *(&var_2 + var_5) = 0x30; break; The var_2 is typed as long, so it looks like &var_2 + var_5 is indexing &var_2 as a long * using the strlen character count, and then dereferencing it as a long, which doesn't seem to make sense in this context. Cumin: I am not sure what 'undefinedn' types are supposed to be (I assume it is a n-byte type), not printing in the short constant string in scanf is a little annoying, the modulus operation on local_18 was not simplified, the control flow within the loop is harder to understand without the switch statement, branches of the switch statement are slightly less concise.

Many variables and nested if-else blocks.

Nested if-else conditions Some parts of the code would be easier to understand when the integers were in decimal instead of being converting to hex

levels of nesting, unresolved types, pointer arithmetic

* Chilli: Pointer arithmetic and dereference instead of indexing operator * Cumin: Deep nesting and unhelpful datatypes * Clove: Nesting

nil

While loops as seen in Clove were unhelpful and impedes analysis when determining the flow of the program.

to many local variables, deeply nested control structures

The use of &local_68 + sVar1 in Cumin seems largely unhelpful.

Referencing static data through global variable (Cumin: __isoc99_scanf(&DAT_00403259,&local_18);). Would rather not click on the variable to find out what it actually is.

Cumin: Heavily nested if statement makes it hard to read. Showing DAT_00403259 instead of the string literal "%lld" means user have to waste time and check the variable and fix it (presumably to make it read-only).

Large nested if-else

- Omission of redundant checks must be done carefully, in case there may be some vulnerabilities in the checks, e.g. overflows

if else instead of switch case

deeply nested ifs

Data types as WORD/QWORD, switch case expanded into if-else

- using long nested else-if constructs - using hex values (when not optimal for readability like e.g. for addresses) - using many type casts - array offset arithmetics - no or undefined variable types

Refer to this: https://i.imgur.com/BtjZedW.jpg

in clove the statements in the form of "*(_WORD *)&s[strlen(s)] = 56;", so as a one-liner, are very hard to read. The long else-if chain is inferior to the switch case.

the exessive type casts in Cumin are confusing, especially with casts to "undefined"

Chilli mixes base-10 and hex-representations of integers for their cases. I prefer a consistent usage of only one of the two. Chillis representation does not contain information about stack positions of variables, which can be quite useful in cases where the decompiler output is not completely correct.

clove uses instructions like *(_WORD *)&s[strlen(s)] = 70; which are not readable at all big amount of undefined local variables (in contrast to chilli)

nested if else statements

The /* format */ annotation in Clove's output Messy chain of if/else in Cumin/Chilli due to lack of switch detection

Too many nested else and if

Naming of variables like "local_10".

Unncessary nested "<" or ">" conditions just to check for another if condition for "==" , such as the one below else if (v11 ¡= 101) { if (v11 == 100) { *(_WORD *)&s[strlen(s)] = 52; } else if (v11 == 11)

undefined8 Having to add offset to pointer and dereference (instead of array access)

Nil. I do understand why all the aspects are needed and can be helpful even though they are unintuitive.

Cumin's decompilation is probably closest to what it looks like in disassembly - a lot of branches. Not very useful though. Likewise with Clove, but Clove does it much better with a cleaner by using lesser intermediate variables.

Too many unnecessary casting information

Long expressions (containing cast operations) are more difficult to read. The nested branching is somewhat cluttered and more difficult to follow.

the decompiled functions here are generally okay, except for the nested if-else blocks.

Cumin

too many indentations

Declaration of numerous local variables, which unclear usage. E.g., 'Cumin' defined the following variables: undefined8 local_68 undefined8 local_60, undefined8 local_58, undefined8 local_50, undefined8 local_48, undefined8 local_40, undefined8 local_38, undefined8 local_30, undefined , cal_28, long local_18, int local_10 uint local_c. But from these variables, only a few (= local_68, local_18, local_10, local_c) were actually used in the code above. This results in cluttered code, with more hard to understand.

Clove: The nested else if statements are hard to read. Cumin: The nested if statements were not as easy to read as the switch case structure in Chilli. The many variables were a bit confusing.

given nested if statements beyond degree 2-3, any alternative cases (especially on the same level but lines further down in the code) become hard to follow quickly.

too many if else conditions

Too many if else statements from Cumin which makes the whole code messy and harder to read & understand.

lots of redundant local variables

Cumin view, logic flow difficult to understand at first glance.

unnecessary casting and convoluted computation

Too many layers of nested blocks of code.

C. Measurement Part

In this part, we will show you short code snippets that represent the same code but are restructured differently. The goal is to find a way to establish a baseline on user preference between the presented approaches.

Copy Instructions If-Else

- Please consider the following two code snippets, which both represent the same semantics with a different structure. In Snippet 1, `/* Block #10 */` is copied to achieve a simplified structure, while in Snippet 2 it occurs only once.
  Snippet 1:

```
/* Block #1 */
while(var_1 > 0){
if(var_2 == 2){
    /* Block #4 */
    if(var_4 == 4){
    /* Block #8 */
    continue;
    }
    /* Block #9 */
}else{
    /* Block #5 */
}
/* Block #10 */ // <----- occurs two times
}
if(var_3 > 3){
/* Block #6 */
}else{
/* Block #7 */
}
/* Block #11 */
/* Block #10 */ // <----- occurs two times
```

  Snippet 2:

```
/* Block #0 */
while(true){
if(var_1 <= 0){
    if(var_2 != 2){
    /* Block #5 */
    break;
    }
    /* Block #4 */
    if(var_4 == 4){
    /* Block #8 */
    continue;
    }
    /* Block #9 */
    break;
}else{
    if(var_3 > 3){
    /* Block #6 */
    }else{
    /* Block #7 */
    }
    /* Block #11 */
    break;
}
}
/* Block #10 */ // <----- occurs only once
```

- Which structure do you prefer in general?

Answer	Count	%
Snippet 1 (Copying /* Block #10 */)	14	25.9
Snippet 2 (Not copying /* Block #10 */)	31	57.4
No preference	8	14.8
No answer	1	1.9

- Now, please consider the following possible instruction sequences as replacement for `/* Block #10 */`:
  Option 1:

```
return var;
```

  Option 2:

```
y = x + 5;
z = bar(y);
printf("Some text to print");
y += z;
return y;
```

  Option 3:

```
y = x + 5;
z = bar(y);
printf("Enter a number larger than %d: \\n",
↪  z);
scanf("%d", &numb1);
printf("Enter a number smaller than %d:
↪  \\n", z);
scanf("%d", &numb2);
y += z;
printf("Enter a number larger than %d: \\n",
↪  y);
scanf("%d", &numb3);
printf("Enter a number smaller than %d:
↪  \\n", y);
scanf("%d", &numb4);
diff_1 = numb1 - numb2;
diff_2 = numb3 - numb4;
printf("The two differences are %d and %d:
↪  \\n", diff_1, diff_2);
return diff_1 + diff_2;
```

	Snippet 1	Snippet 2	No preference
Option 1 (1 Instruction)	42 77.8	4 7.4	8 14.8
Option 2 (5 Instructions)	10 18.5	36 66.7	8 14.8
Option 3 (15 Instructions)	3 5.6	49 90.7	2 3.7

- Please explain your choices. Which criteria guide your decision?

---

the length of the structure matters, but the main reason to include it not twice is the extra step it adds to understanding which configurations use the specified paths

At some scoping levels duplicating a ret, or a jump to the epilog seems worthit, otherwise it depends on how much code is around it. A return statement inside of a while-statement is often confusing and in the given example the scope-depth is manageable, so i'd vote for no duplication.

Am assuming code duplication helps in writing a "cleaner-looking" while loop. The overall guiding principle is to prevent analysis of additional code that appears elsewhere too.

I'd prefer snippet 1 for relatively small code blocks. . However, the larger the block, the harder it would be to visually distinguish between identical code and code with slight differences.

overview: the less code one sees, the easier it gets to reason about the code

for short snippets context is clear even if they are copied, leading to cleaner code. copying longer sequences might lead to confusion

Primary factor is how close it it to code that I think a human might write. I think except for very short blocks like Option 1 it feels more natural to not duplicate the code, even if it makes the control flow a little bit more complex. Having less code to read is generally good as well, as now I do not have to figure out whether two blocks of code separated by large amounts of other code actually do the same thing or not. At the cost of slightly more complex flow I think this makes reversing easier.

To copy one return statement is just fine, but if it's more than one or two lines, I think it is confusing, and it bloats the decompiled code.

With only 1 return instruction, it's more readable to understand that the function terminates given certain conditions. However, when there's additional logic, probably a shorter code (with no copying) is more readable.

If the code gets too long, it's more difficult to see, that it's the same block of code, if it is copied. It's shorter and better to overview, if it's not copied as well. But it depends on the code itself. If the code itself gets more complicated and nested, if the block is not copied, the copied option may be better.

Preventing duplicated code, but a single return statement is not worth the more complex control flow.

Even if Snippet 1 has a simplified structure, a lengthy and repeated "Block #10" would not make it easier to read as compared to Snippet 2.

Single line codes will take less effort in terms of analysis, in terms of the example, a return shows the end of a pathway quickly, allowing for (targeted) quick dives of code snippets as compared to full code analysis. Longer codes (Option 2 and 15) will require more tactful approach as this functions occurring twice in a single function may throw off the analysis and waste valuable time re-analyzing a function that has already been dived through.

single statements are tolerable

Is there is too many instructions then it will over complicate things if we repeat it.

A simple return statement will be good in guiding control-flow analysis. However when there are additional n operations performed before the return, it may complicate/make analysis less intuitive. For example if there are huge chunks of operations done before

the program return, I may forget what happens during a return and analyse the n operations again. In this case, having the /* Block #10 */ in a single place helps simplify my thought process.

Copying is fine as long as it is not too long. Forcing it to be a single block can make other code structure become harder to read. E.g. snippet 2's while loop contains several break statement which makes it harder to follow.

Once we've identified a unique block of code, repetition is fine since we can easily mark and understand the function. It's more important to have a simplified structure to better understand the overall flow.

For shorter snippets I prefer it to be copied since it helps me parse the flow better; forcing a break out of a while(true) just to handle it without copying seems a bit strange to me. I put no preference for longer snippet because it really depends on how long the other code blocks are as well; i think there's merit to keeping code short of the rest of the code is already super long.

- I prefer while-condition loops to while-true loops with many inner checks, as they are easier to interpret. - It is hard to compare longer snippets and notice that they are identical. Possible solutions: - Inline function - Goto label

If the same block of code can be presented without copying (so there is only one block to process) it is much simpler to understand the code flow and logic. Having two (or more) exact copies of a block of code in a function makes the logic generally harder to understand, takes up more screen real-estate and is just generally annoying because it needs to be abstracted out by your mind when processing the logic of the decompiled code.

the sequence of conditions that have to be checked is clearer

Repetition is fine for short instruction sequences else the readability will be affected

The structure of snippet 1 is much better in general, because blocks 6, 7 and 11 don't need to be in the loop. If the code duplication is too much, i would prefer the worse structured code, though.

Just the length of the code

As soon as multiple lines are duplicate i am caught up checking if they are actually duplicate, or if there are small nuances/differences that i might overlook

to me its about the visual impact of the block more than its true length or complexity, option 3 has a huge visual impact because it is both long (number of lines) as well as wide (lots of long-ish text literals). option 2 is right at the borderline for me. i'd be okay with either copying or not copying in this case

For small snippets I only care about how easy it is to follow and understand the control flow. For larger snippets (5 lines or more?) less repetition means less time spent on trying to understand the same code twice.

1. Keep your code DRY, always! 2. Snippet 2 is easier to read because the code is better structured

An easy structure makes it much quicker to understand the code. So I always prefer an easiert to read structure

Not copying longer snippets feels more natural and likely closer to the original source.

Length

Code length

When decompiled code is repeated, i tend to refer to the disassembly for the memory address for the instructions. If the code is repeated in the decompiled code, it may get quite confusing if they are pointing to the same memory address. For a single instruction such as Option 1, it is still manageable, but for Options 2 and 3, not repeating the block may seem tidier and easier to understand

Amount of things to read, only to realise it's a copy Even if realised, it's still visually irritating, and tedious to manually refactor With Snippet 1 it's okay but still convenient to have just one

No preference for return instructions. However, i do like decompiled codes to look like proper source code and i feel that most programmers will not have multiple blocks of similar instructions in their source code.

If Block 10 contained interesting operations or needed a more in-depth look, having it in one location lets me focus better. Usually I like to comment/highlight, having multiple copies just means that I'd need to copy my highlights and comments to duplicate blocks.

The number of lines of code. Prefer larger code blocks to not be copied/repeated as it can make it harder/take longer for users to understand that such code blocks are actually the same.

If the instruction sequence in Block #10 is too long, the overall code appears cluttered and becomes more difficult to read, even though it only contains the same instruction sequence multiple times.

Unless the block is a one-liner that branches, ("return, break, continue, call"), I prefer to not have a block of code copied out.

number of instructions. snippet 2 is preferred if there are many instructions

I think that in snippet 2 the general structure of the program and how/ when we reach Block #10 is easier to see.

if repetition is avoidable, that's output I would prefer as it allows to code paths to merge into one point eventually. If the repetition is comprised of 1,2,3 instructions, that may be helpful, especially if the code path ends there (return) - means potential code paths do not have to be followed mentally anymore.

option 3 is too lengthy

I feel that a longer instruction if possible should not be repeated but only once while a shorter instruction can be repeated to simplify the code structure

duplicate code can lead to reading the same things twice even though it's doing the same. So if a decompiled decides to duplicate code, it should only do so for portions that one can be understood very fast.

Length of code.

preference to avoid repeated codes blocks, even if it is 5 lines.

More complex the code block should not be duplicated.

- Now, please assume that `/* Block #10 */` contains a more complex structure, for example the following:

```
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
    var /= numb;
    }else{
    var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
}
return var;
```

- Snippet 1 and Snippet 2 are structured as before. In Snippet 3, `/* Block #10 */` is extracted as a function.
Snippet 1:

```
/* Block #0 */
while(var_1 > 0){
if(var_2 == 2){
    /* Block #4 */
    if(var_4 == 4){
    /* Block #8 */
    continue;
    }
    /* Block #9 */
}else{
    /* Block #5 */
}
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
    var /= numb;
    }else{
    var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
}
return var;
}
if(var_3 > 3){
/* Block #6 */
}else{
/* Block #7 */
}
/* Block #11 */
while(var > 1){
printf("Enter a number \\n");
scanf("%d", &numb1);
if(var % numb == 0){
    var /= numb;
}else{
    var -= numb;
}
printf("The new number is %d, \\n", var);
}
return var;
```

Snippet 2:

```
/* Block #0 */
while(true){
if(var_1 < 0){
    if(var_2 != 2){
    /* Block #5 */
    break;
    }
    /* Block #4 */
    if(var_4 == 4){
    /* Block #8 */
    continue;
    }
    /* Block #9 */
    break;
}else{
    if(var_3 > 3){
    /* Block #6 */
    }else{
    /* Block #7 */
    }
    /* Block #11 */
    break;
}
}
while(var > 1){
printf("Enter a number \\n");
scanf("%d", &numb1);
if(var % numb == 0){
    var /= numb;
}else{
    var -= numb;
}
printf("The new number is %d, \\n", var);
}
return var;
```

Snippet 3:

```
/* Block #0 */
while(var_1 > 0){
if(var_2 == 2){
    /* Block #4 */
    if(var_4 == 4){
    /* Block #8 */
    continue;
    }
    /* Block #9 */
}else{
    /* Block #5 */
}
return call_sub(var);
}
if(var_3 > 3){
/* Block #6 */
}else{
/* Block #7 */
}
/* Block #11 */
```

```
return call_sub(var);


int call_sub(int var){
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
    var /= numb;
    }else{
    var -= numb;
    }
    printf("The new number is %d, \\n",
 ↪  var);
}
return var;
}
```

- Which Snippets do you like? (Multiple choices are possible)

Answer	Count	%
Snippet 1 (Copying /* Block #10 */) (Sample 1)	1	1.9
Snippet 2 (Not copying /* Block #10 */) (Sample 2)	27	50.0
Snippet 3 (Extracting /* Block #10 */ as a function and calling it twice)	46	85.2

- Please explain your choice.

  by declaring a function the multiple calls are easier to read than just including the code multiple times

  Introducing _inline_ functions complicates the programm which i am not a fan of. In this case, as block#10 containes a return-statement, I feel that it should always be at the bottom, so that the control flow is easier to read. Ghidra often uses the 4th option of using a goto, more or less combining the 'simpler' scoping without copying. But gotos can also be confusing.

  Minimal duplication of instructions would be great. Abstracting logic to a seaprate function to improve the readability for while loops is welcome too.

  Snippet 1 creates visuals that cause the code to appear more complex than it must be. My favorite would be Snippet 2, because it provides more "sequential readability". Snippet 3 is fine, too, but would require me to read the code less sequentially. This MAY be more beneficial depending on the actual code block and actual function length, because I can analyze the "outsourced" code more isolated. Afterward, I know what comes out of this function and know directly what's going on when I see the function call.

  I do like this approach since it simplifies and shortens the information on the first look. I'd also suggest to have distinct function names specifically for the decompiler-generated functions.

  the reduced complexity in the remaining code due to copying, seems to be balanced with the added complexity of the duplicated block. hiding complexity behind function calls seems much cleaner

  Not #1, for the reasons I've explained before. #2 and #3 are acceptable, especially if I can label code lines to quickly know where the break ends up. #3 would be preferred if there is some way to indicate that the call is inlined (just to indicate that there is no actual call happening), and if the extracted routine is complex enough and used often enough to warrant extraction. Extracting out too many of these into functions that don't get reused often enough might hinder understanding.

  Snippet 3 make sit very clear in which case we run that code. Snippet 1 does this also, but bloats the logic in the while loop, causing it to be harder to read. Snippet 2 is still favorable over Snippet 2, but the cases in which the block runs is more implicit and hence also harder to read as Snippet 3.

  Snippet 2 will probably be more accurate as it doesn't add additional function calls, but snippet 3 will probably be more readable

  Functions are "clean code" and it's easy to see, that this is a "functional" block, which does exactly one thing and which variables are neccessary for that thing. On the other hand, too much function jumping may hinder the analysis as well, if you get lost and don't know, where you came from.

  Same as before; code duplication is avoided in snippet 2 and 3. I prefer 3 over 2.

  It reduces the number of code lines and is very readable with an appropriate function name.

  Snippet 3 is cleaner and prevent un-necessary re-analysis of code snippets (as seen in snippet 1).

  Makes things look simpler as long as it is clearly mentioned that it is a decompiler only function

  Would rather have the block in a single location as explained previously. Also would rather the block not be extracted out as a function as I would prefer seeing everything in one place (rather than clicking on the call_sub function and to see what's going on).

  Snippet 3 retains the clarity of the while loop without introducing break statements. The copying of the "call_sub" is fine as it is only a single line. Furthermore, user can rename the "call_sub" function to make it descriptive.

  Extracting it out as a function further helps us in identifying a unique block of code across different sections, and is naturally more helpful for interpretation

  Helps to meet the criteria I stated for my choices in the previous question; flows better, and helps to shorten the code.

  It avoids both problems of the while true loop and comparing long snippets. I would prefer the function to be marked "inline", and would not mind a goto-

label structure if used once per function.

Snippet 3 is able to achieve a simplified structure (ie. easier to understand the control flow compared to snippet 2) but still keep the decompiled code neat and not introduce too many lines of repeated code which takes up more screen space.

refactoring is lit

Length of instruction sequence is too long for snippet 1. It is fine to be inlined in snippet 2 or as a function call

For such a large section, function extraction is a very good choice.

Extracting functions is the way to go, at least for anything larger than a few lines in my opinion

Now i dont have to worry if the snippets are actually identical, as i described before

the block is again on the borderline of visual impact for me, but I dislike excessive break-ing. snippet 3 is a really elegant solution, both minimizing excessive breaking as well as having to re-extract the same semantics from a block of code

As above, less repetition means less time spent on trying to understand the same code twice. Extracting the code snippet into its own function yields a very readable control flow but also implies adherence to calling conventions! Depending on the analysis questions this can actually be counterproductive, especially if the analyst blindly assumes that the original assembly code also adheres to a calling convention here.

1. Again the DRY argument 2. extracting code to a separate function (separation of concerns) is good for readability

It looks like its the most readable version

Snippet 2 feels closest to what original source would be and is more 'readable' than Snippet 1. Extracting blocks as functions can get confusing if they reference local variables.

More clean

If the code block is to be duplicated, I would prefer it to be either short, or made into a function like Snippet 3.

Code is more presentable and understandable, compared to snippet 2 where there is a while(true) loop waiting for break conditions. Snippet 3 has a clear exit condition for the while loop.

At least it's still one visual block to read

Like i said, snippet 3 looks more like a proper source code. Snippet 2 might be more complex but it is still acceptable. Snippet 1 does not look like how a programmer will write his source code.

Similar reasons as above, but no real preference between snippet 2 or 3.

Both choices allows me to understand more quickly when a code block will be executed. Better preference for snippet 3 if the extracted code block is relatively large, making the code more modular and easier to understand.

Even though Block #10 appears in both Snippet 2 and Snippet 3 only once, Snippet 3 appears to be more readable as it's branching seems to be less complex compared to Snippet 2. Also extracting code which is used at multiple locations into a named function makes it easier to remember what the code sequence actually does.

Unless the block is a one-liner that branches, ("return, break, continue, call"), I prefer to not have a block of code copied out. Snippet 3 here simplifies the block into a one-liner, which falls into the ("return, break, continue, call") category. In such cases where a block is "extracted" into a function, I think it'll be good to also have a hint that the function does not actually exist.

I like snippet 3 the best because it removes the code duplication from snippet 1. Although the function is called twice, it is more readable than snippet 2 because the structure is easier to follow.

As above I think not duplicating the snippet makes the program structure easier to see. Extracting it as a function has a similar effect to just using the snippet once.

Here, snippet 2 merges potential code paths at a point where something relevant is happening (user interaction). Function extraction (snippet 3) looks just as good to me - as long as it is clearly stated that the output has been optimized for comprehension and is not as close to its disassembly source/structure.

cleaner structure and easier to understand. can simply go through the function once to understand the operations involved.

I would prefer snippet 2 more because of non repeated codes but snippet 3 is also fine if repeat is necessary as making it into a function tidies up the structure.

Especially for a very simple case like this (only called ones, instead of '/*logic*/ return var;' it is just 'return foo(); int foo() /* logic */'), hat the decompiler should imho not create artificial functions.

Fewer lines of code.

neater and avoids repeated code blocks

The criteria to enter the Block #10 only occurs at 1 place..

first snippet increases cognitive load while trying to match both blocks third snippet could distract me from code comprehension in terms of existent functions (which are present in real in binary) and virtual (which introduced by decompiler)

Copy Instructions Multiple Exit for Loops

- Please consider the following two code snippets, which both represent the same semantics with a different structure. In Snippet 1, `/* Block #10 */` is copied to achieve a simplified structure, while in Snippet 2 it occurs only once.
  Snippet 1:

```
/* Block #0 */
if(var_0 > 10){
while(var_1 > 0){
    if(var_2 == 2){
    /* Block #4 */
    if(var_4 == 4){
        /* Block #8 */
        continue;
    }
    /* Block #9 */
    }else{
    /* Block #5 */
    }
    /* Block #10 */ // <----- occurs two
↪   times
}
}
/* Block #3 */
if(var_3 > 3){
/* Block #6 */
}else{
/* Block #7 */
}
/* Block #11 */
/* Block #10 */ // <----- occurs two times
```

  Snippet 2:

```
/* Block #0 */
if(var_0 > 10){
while(true){
    if(var_1 > 0){
    exit_1 = 0;
    break;
    }
    if(var_2 == 2){
    /* Block #4 */
    if(var_4 == 4){
        /* Block #8 */
        continue;
    }
    /* Block #9 */
    }else{
    /* Block #5 */
    }
    exit = 1;
    break;
    }
}
if(var_0 <= 10 || exit_1 == 0){
if(var_3 > 3){
    /* Block #6 */
}else{
    /* Block #7 */
}
/* Block #11 */
}
/* Block #10 */ // <----- occurs only once
```

- Which structure do you prefer in general?

Answer	Count	%
Snippet 1 (Copying /* Block #10 */)	19	35.2
Snippet 2 (Not copying /* Block #10 */)	22	40.7
No preference	10	18.5
No answer	3	5.6

- Now, please consider the following possible instruction sequences as replacement for `/* Block #10 */`:
  Option 1:

```
return var;
```

  Option 2:

```
y = x + 5;
z = bar(y);
printf("Some text to print");
y += z;
return y;
```

  Option 3:

```
y = x + 5;
z = bar(y);
printf("Enter a number larger than %d: \\n",
↪   z);
scanf("%d", &numb1);
printf("Enter a number smaller than %d:
↪   \\n", z);
scanf("%d", &numb2);
y += z;
printf("Enter a number larger than %d: \\n",
↪   y);
scanf("%d", &numb3);
printf("Enter a number smaller than %d:
↪   \\n", y);
scanf("%d", &numb4);
diff_1 = numb1 - numb2;
diff_2 = numb3 - numb4;
printf("The two differences are %d and %d:
↪   \\n", diff_1, diff_2);
return diff_1 + diff_2;
```

	Snippet 1	Snippet 2	No preference	no answer
Option 1 (1 Instruction)	40 74.1	7 13.0	6 11.1	1 1.9
Option 2 (5 Instructions)	17 31.5	27 50.0	9 16.7	1 1.9
Option 3 (15 Instructions)	3 5.6	46 85.2	3 5.6	1 1.9

- Please explain your choices. Which criteria guide your decision?

only one return statements helps to understand the cfg better

A return statement inside of a while-statement I generally find confusing. In this case the scope-depth seems to be quite low and thus I would avoid copying in all cases. If the scoping depth is really high, or there is a lot of complex code around it, ghidra sometimes introduces a goto in cases like this.

Reducing duplication is still preferred, as per the previous question.

The code's expressiveness, cumulative length, and block size. The longer the block, the more dangerous it is to overview slight differences when it would be "copied".

in this case, it's smarter to just break the while loop.

longer sequences that are duplicated add to complexity

Similar reasoning as previous question, but now because of additional changes of introducing the temp variable and increased control flow complexity, I feel that I would be more willing to tolerate a few extra lines.

To copy one or two lines is preferable for clarity, but more than that can bloat the code and makes it harder to read.

I find the additional logic of the exit variable more confusing to follow.

If the block gets longer, it is more difficult to see, if it's the same or not. Therefore not copying the block seems more convenient. If depends on the complexity of the overall code though. If the code becomes more difficult to read and more nested, if the block is not copied, the other option may be better.

The control flow in snippet 2 is too complex to easily understand.

it feels less complicated to read when the code is more segregated.

As Previously stated, shorter codes that needs not be re-analyzed can be repeated, and repeats of longer code snippets should not.

If its too long then copying will overcomplicate things

Easier to analyze control-flow if there is just a simple return statement. However when there are operations done before the return, I would rather have them all in one place (not copying) since it would simplify my analysis. For example, if there are a huge chunk of operations done before the return block, I may have forgotten what was being performed in the return and attempt to analyze the return block again. I rather have them in one place so I can add comments on their meaning in a single location where I can refer to.

Copying is fine for short instruction sequences. Not copying makes sense for long instruction sequence. But the multiple exits in the while loop makes it harder to read.

Same as earlier

Same as previously, I like having the flow in Snippet 1, and the "exit" flag and immediate check feels strange to me. But I can see the argument for shortening it if the code blocks are too long.

- A while-true loop with many breaks and continues is confusing, so I prefer the while-condition loop with the copied block. - It is hard to compare longer snippets and notice that they are identical.

I can see the value of simplifying the structure like in Snippet 1 but in this case, the control flow is easy enough to understand and IMO is not worth the trade-off of having to repeat the same code block.

isn't this the same thing as the last part

Option 1 is short enough to be repeated but not option 2 and 3

As in the last step, I would prefer duplication as long as its not too much. I like function function extraction even more, though. P.S.: What happened to Block #3 in the first example?

Readability mostly

Same reason as in the last question (hard to see if actually identical or different)

i dislike excessive break-ing and the branch conditions are more complex in snippet 2, but the visual impact of option 3 is too large for copying

Less repetition means less time spent on trying to understand the same code twice. But if I have to pay for it via more complex control flow conditions (like that exit_1 variable), I lean on preferring easier control flow most of the time.

If block 10 would be moved to a separate function, I would always prefer snippet 1

I prefer an easy structure instead of less code

Following control flow in Snippet 2 is difficult and only tolerable in Snippet 1 for the simple return statement.

Length

Code length. Prefer to have less lines of code if the code block is going to be duplicated.

Multiple exit conditions with break seems more confusing compared to a while loop with a fixed exit condition. But if the repeated block gets too large, not repeating it seems more reasonable

IMO Option 1 and Option 2 is not worth the effort of having to follow Snippet 2?

I prefer the simplicity of snippet 2. It is also more understandable.

Similar explaination as before - Not Copying makes it easier for analysis.

For larger code blocks that are copied, it takes more time to realize that both blocks are actually identical and hinders program understanding.

If the instruction sequence is relatively short, copying the sequence into multiple locations appears to be a good trade-off between overall code length and a more complex. The instruction sequence in Option 2 seems to be a good boundary, whereas the seuquence in Option 3 would make Snippet 1 appear more cluttered.

Similar to the section before.

Option 3 is the codes, which shows most explicit what exactly it does. Although it is more lengthy than the other options, it is the code I best understand / can read. Because, option3 would be for snippet 2 called outside the while statement, it is best, if the intention of the instructions are clear, without the context of the loop. I chose option1 for the first snipped, because of its simplicity. Especially if the code is called twice and is not wrapped by a function, it is preferable if the code is then simple.

When not copying the snippet, the structure is easier to see.

I think I already had this question? Anyhow, short code copied (1,2,3 instructions) is fine, for longer instruction sequences, merging flow is preferred.

too lengthy

Would prefer codes not to be repeated but if necessary then the shorter instructions could be repeated

same as before

preference for not reading code blocks that are identical.

complex code should only be evaluated once or minimally.

- Now, please assume that `/* Block #10 */` contains a more complex structure, like e.g. the following:

```
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
    var /= numb;
    }else{
    var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
}
return var;
```

- Snippet 1 and Snippet 2 are structured as before. In Snippet 3, `/* Block #10 */` is extracted as a function.
Snippet 1:

```
/* Block #0 */
if(var_0 > 10){
while(var_1 > 0){
    if(var_2 == 2){
    /* Block #4 */
    if(var_4 == 4){
        /* Block #8 */
        continue;
    }
    /* Block #9 */
    }else{
    /* Block #5 */
    }
    while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
        var /= numb;
    }else{
        var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
    }
    return var;
}
}
/* Block #3 */
if(var_3 > 3){
/* Block #6 */
}else{
/* Block #7 */
}
/* Block #11 */
while(var > 1){
printf("Enter a number \\n");
scanf("%d", &numb1);
if(var % numb == 0){
    var /= numb;
}else{
    var -= numb;
}
printf("The new number is %d, \\n", var);
}
return var;
```

Snippet 2:

```
/* Block #0 */
if(var_0 > 10){
while(true){
    if(var_1 > 0){
    exit_1 = 0;
    break;
    }
    if(var_2 == 2){
    /* Block #4 */
    if(var_4 == 4){
        /* Block #8 */
        continue;
    }
    /* Block #9 */
    }else{
    /* Block #5 */
    }
    exit = 1;
    break;
}
}
if(var_0 <= 10 || exit_1 == 0){
if(var_3 > 3){
    /* Block #6 */
}else{
    /* Block #7 */
}
/* Block #11 */
}
while(var > 1){
printf("Enter a number \\n");
scanf("%d", &numb1);
```

```
if(var % numb == 0){
    var /= numb;
}else{
    var -= numb;
}
printf("The new number is %d, \\n", var);
}
return var;
```

Snippet 3:

```
/* Block #0 */
if(var_0 > 10){
while(var_1 > 0){
    if(var_2 == 2){
    /* Block #4 */
    if(var_4 == 4){
        /* Block #8 */
        continue;
    }
    /* Block #9 */
    }else{
    /* Block #5 */
    }
    return call_sub(var);
}
}
/* Block #3 */
if(var_3 > 3){
/* Block #6 */
}else{
/* Block #7 */
}
/* Block #11 */
return call_sub(var);


int call_sub(int var){
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
        var /= numb;
    }else{
        var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
}
return var;
}
```

- Which Snippets do you like? (Multiple choices are possible)

Answer	Count	%
Snippet 1 (Copying /* Block #10 */) (Sample 1)	4	7.4
Snippet 2 (Not copying /* Block #10 */) (Sample 2)	26	48.2
Snippet 3 (Extracting /* Block #10 */ as a function and calling it twice)	47	87.0

- Please explain your choice.

declaring the duplicate code in a function looks cleaner. multiple return statements require an extra look

Introducing a function to me seems like introducing complexity, which seems bad, but as ghidra does not do that I don't really now how it would work out. It might be worth a test. In this case (the same reason as above) I would vote to not copy the block.

Reducing duplication is still preferred, as per the previous question.

Again: It would be too convenient for my brain to see the redundancies of a complex copied block. I'd scan the code real quick and then just skip it. However, what if there are slight nuances in code that differ? That would be too dangerous for me.

When encountering somewhat big repeated code-block, I'd prefer extracting the block as a function.

complexity between 1 and 2 seems to be balanced. snippet 3 looks clean

Same reasoning as before, but here I am leaning more towards call_sub because of the increased control flow complexity in #2.

The function call in Snippet 3 makes it very clear when the block is executed. In Snippet 2, this is more implicit and hence harder to read. Snippet 1 makes this very clear, but bloats the code logic, causing it also to be harder to read as Snippet 3.

Same as above, I find the additional logic of the exit variable more confusing to follow. And same as previous question, extracting as a function can potentially make it more human-readable but may not be the truest representation of the program.

It seems more clean, like "clean code" and it is easy to see, that the function is a block on its own, which does one thing and to see, which variables are neccessary for it. To much function jumping on the other hand may not be beneficial too, because it's easy to loose track, where you came from.

Snippet 3 is easy to understand with a simple control flow.

substituting function calls with makes it easier to read.

Less analysis required and no repeated functions. Functions are read 1-pass and in the case of snippet 3, code is separated in different function and can be analyzed seperately

Looks similar to how you would do in C code

Would prefer the return block in one place as explained previously. Also would prefer the block not be extracted out as a function as I would rather have everything in one place for easier analysis (I don't have to click the call_sub function to analyze it).

While loop is more readable without multiple exits. User can also rename the "call_sub" to make it more descriptive.

Same as earlier

Like the previous question, I like to extract as function to allow me to maintain the flow while reducing code length.

It avoids both problems of the while-true loop and comparing long snippets. I would prefer the function to be marked ”inline”, and would not mind a goto-label structure if used once per function.

Repeated code blocks is quite annoying and requires more focus to abstract out when understanding the decompiled code logic.

refactoring is still lit

Snippet 2 and 3 are cleaner and easier to read

I think function extraction makes sense here (because of the length of the block)

Refactor anything that is longer than a few lines and occurs multiple times

again same reason as in the last question (easier to see that its identical)

again, excessive breaking in snippet 2 and more complex branch conditions visual impact of the block is near the borderline for copying

As mentioned in a previous question, I do not like the implied adherence to a calling convention in snippet 3 if it may not be true.

Has the advantage of having the simpler/more readable structure with benefit of DRY code

just the easiest to read and understand

Removing gotos in this case causes nothing but pain

More clean

Prefer not to have duplicated code blocks if there are many lines of code. If there is, would be good to have it in a function like Snippet 3.

Code looks easier to understand with while loop exit condition clearly defined

IMO feels the cleanest and easiest to follow?

Each of the functions are smaller and much easier to understand.

As above.

For more complicated blocks I prefer if it is not copied again or it is extracted into a function for quicker understanding of the code.

Extracting Block #10 into a separate function as in Snippet 3 is the most favorable solution as it enables to decrease the overall code length and the complexity of the code structure. Snippet 2’s structure is more complex than the structure of Snippet 1, but as Block #10 contains a more complex structure, copying the code block into multiple locations would increase the overall complexity of Snippet 1.

Similar to the section before. Note that the ”extracted” function should have a hint that it is a real function.

Snippet 1 is the snippet which I like the least, because of the code duplication. I prefer the structure of snippet 2 over the wrapping function of snippet 3 because its instructions are less difficult to understand for me.

When not copying the snippet, the structure is easier to see. Replacing the snippet with a function call also makes this easier, although having the snippet only once is still the easiest.

same arguments as before, relevant functionality happening after code flow is merged is nice and function extraction is fine as well.

easier to follow as the code is structured

Snippet 2 & 3 makes the code looks tidier as there are no repeats of code compared to snippet 1.

Snippet 1 duplicates a lot of code, so absolutely not an option. Snipped 2 needs to use the helper variables to avoid the duplication, which adds cognitive load, hence Snippet 3 is preferable.

Fewer line of code.

neater and easier to understand.

Criteria to enter into Block #10 is only evaluated once.

Copy Instructions Multiple Entry for Loops

- Please consider the following two code snippets, which both represent the same semantics with a different structure. In Snippet 1, `/* Block #4 */` is copied to achieve a simplified structure, while in Snippet 2 it is not.
  Snippet 1:

```
/* Block #0 */
if(var_0 > 10){
/* Block #1 */
/* Block #4 */ // <----- occurs two times
}else{
/* Block #2 */
}
while(var_3 > 0){
/* Block #4 */ // <----- occurs two times
}
/* Block #5 */
```

  Snippet 2:

```
/* Block #0 */
if(var_0 > 10){
/* Block #1 */
entry_1 = 1;
}else{
/* Block #2 */
entry_1 = 0;
}
while(true){
if(entry_1 == 0 && var_3 > 0){
    break;
}
/* Block #4 */ // <----- occurs only once
entry_1 = 0;
```

```
}
/* Block #5 */
```

- Which structure do you prefer in general?

Answer	Count	%
Snippet 1 (Copying /* Block #4 */)	27	50.0
Snippet 2 (Not copying /* Block #4 */)	17	31.5
No preference	9	16.7
No answer	1	1.9

- Now, please consider the following possible instruction sequences as replacement for `/* Block #4 */`:
  Option 1:

```
return var;
```

  Option 2:

```
y = x + 5;
z = bar(y);
printf("Some text to print");
y += z;
return y;
```

  Option 3:

```
y = x + 5;
z = bar(y);
printf("Enter a number larger than %d: \\n",
 ↪  z);
scanf("%d", &numb1);
printf("Enter a number smaller than %d:
 ↪  \\n", z);
scanf("%d", &numb2);
y += z;
printf("Enter a number larger than %d: \\n",
 ↪  y);
scanf("%d", &numb3);
printf("Enter a number smaller than %d:
 ↪  \\n", y);
scanf("%d", &numb4);
diff_1 = numb1 - numb2;
diff_2 = numb3 - numb4;
printf("The two differences are %d and %d:
 ↪  \\n", diff_1, diff_2);
return diff_1 + diff_2;
```

	Snippet 1	Snippet 2	No preference	no answer
Option 1 (1 Instruction)	47 87.0	1 1.9	5 9.3	1 1.9
Option 2 (5 Instructions)	22 40.7	21 38.9	10 18.5	1 1.9
Option 3 (15 Instructions)	9 16.7	39 72.2	3 5.6	3 5.6

- Please explain your choices. Which criteria guide your decision?

---

it reduces the while loop to one block which helps understanding it faster

This example is a little bit weird. As Block #4 returns snipped 2 could be drastically simplified. Further, I can't really tell what the underlying machine code is. If there is an actual variable 'entry_1' (either stack or register) then eliminating the code that sets and reads the value, is not acceptable. But probably the 'if(var_0 > 10)' just jumps into the while, in which case the c-scoping semantics are not really obeyed and the cleanest solution would to be a goto.

While it might be personally more cumbersome to read, the use of entry flags can help to signpost possible code flows.

in this case, I'm hesitant to choose the copy feature because sometimes the logic is repeated for a reason, but here you would make it less obvious.

breaking condition in snippet 2 is less complex than sequences in option 2 and 3 that would be duplicated

Similar to previous question, here I think I would lean more towards copying for similar reasons as before.

Because the nesting is so shallow, copying short blocks is preferable for clarity in this case.

Same as previously, I find the additional logic of the entry variable more confusing to follow.

Since the code gets more complicated, if not copyied, but is very easy, if copied, option 1 and 2 are more convenient for snippet 1. But in option 3 the block is very large and snippet 2 becomes more convenient.

Only entering a loop to break it in the first iteration can be hard to follow, so I would prefer to avoid snippet 2.

Snippet 1 has a significantly more simplified structure to tolerate the copying.

single line and small (simple) code snippets can be repeated, but long snippets should not be to prevent unnecessary re-analysis of code.

Length of copied code

In snippet 2, I would only have to analyze the if statement in line 10 to determine if the block will be executed at least once. However for snippet 1, I have to analyse the if statement in line 4 and the while statement in line 9 to determine if the block will be executed.

Length of the copy affects readability.

Same as earlier

Same as previous.

- I prefer while-condition loops to while-true loops - The entry_1 flag is hard to keep track of - It is hard to compare longer snippets and notice that they are

identical, so I prefer not copying them.

Copying code here does make the conditions for block #4 to be executed clearer. However, if block #4 is large, it is going to be annoying to read.

while(true) is annoying with the exit conditions

Shorter instruction sequence for readability. The logic flow is not lost in snippet 2.

The structure in Snippet 2 is really ugly and hard to unwind (when the actual blocks are filled in).

the ”copying” version is for more readable for me, but if there are too many instructions it becomes messy again

while the visual impact of option 3 is significant, in this case the increase in complexity of snippet 2 compared to snippet 1 outweighs the visual impact

Side note: All options end in return instruction, which seems somewhat wrong considering the code of the two snippets. Except for repetition of large snippets I prefer less variables that need to be followed to understand the control flow.

I dislike the use of extra variables to denote control flow in Snippet 2, and copying large blocks is undesirable in Snippet 1. while() return is also terrible decompilation

Length

Code length for Option 1. Option 3 is too long with many function calls.

If there are multiple entry within loops, would prefer snippet 1 to show the possible entries into the block. However, if the block is huge, I would not prefer either snippets, as the block should be modularized into a function on its own.

Option 1 and Option 2: IMO copying is easier than having to follow the entry variable? Not sure for Option 3

Simplicity, ease of understanding

Same reasoning as before. Less duplicated code means less bloat.

Size of code block. For larger/more complicated blocks I would prefer for it to not be repeated as it makes it harder to realize two code blocks at different parts of the code are actually the same.

This time the structure of Snippet 1 and Snippet 2 appear to be equally complex. Thus, copying Option 2 could be avoided as it does necessarily bring a less complex structure. Option 3 appears to be too large to be copied into multiple locations compared to the reduced structure complexity that is achieved.

Similar to the section before.

I choose option 1 for snippet 1 to reduce the code duplication. Option 3 is in my opinion better suitable for snippet 2 because the code is more ’expressive’, meaning it shows in a meaningful, explicit way what is exactly does, whereas the simple return statement might be unclear, what exactly being returned.

The criteria in line 10 and 11 of the second snippet were a bit hard to read, but for a longer block 4, this would be worth not duplicating the long code block.

overall, I like snippet one more because it has simpler conditional expressions. but once again, if the copied code portion becomes too long, I’d rather have it once during a time of mostly non-conditional, linear code flow.

length of the block

As the codes in snippet 1 is shorter than snippet 2, option 1 being repeated is fine but if the instructions are long, then snippet 2 will be easier to read

same as before

for 1-liners, it is ok to duplicate the instructions. otherwise, it is preferred for identical code blocks (functionality) to be called once.

Complex code should be evaluated once or minimally.

- Now, please assume that `/* Block #4 */` contains a more complex structure, like e.g. the following:

```
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
    var /= numb;
    }else{
    var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
}
return var;
```

- Snippet 1 and Snippet 2 are structured as before. In Snippet 3, `/* Block #4 */` cis extracted as a function.
Snippet 1:

```
/* Block #0 */
if(var_0 > 10){
/* Block #1 */
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
    var /= numb;
    }else{
    var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
}
}else{
/* Block #2 */
}
while(var_3 > 0){
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
```

```
    if(var % numb == 0){
    var /= numb;
    }else{
    var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
}
}
/* Block #5 */
```

Snippet 2:

```
/* Block #0 */
if(var_0 > 10){
/* Block #1 */
entry_1 = 1;
}else{
/* Block #2 */
entry_1 = 0;
}
while(true){
if(entry_1 == 0 && var_3 > 0){
    break;
}
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
    var /= numb;
    }else{
    var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
}
entry_1 = 0;
}
/* Block #5 */
```

Snippet 3:

```
/* Block #0 */
if(var_0 > 10){
/* Block #1 */
var = call_sub(var);
}else{
/* Block #2 */
}
while(var_3 > 0){
var = call_sub(var);
}
/* Block #5 */

int call_sub(int var){
while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){
    var /= numb;
    }else{
    var -= numb;
    }
    printf("The new number is %d, \\n",
↪   var);
}
return var;
}
```

- Which Snippets do you like? (Multiple choices are possible)

Answer	Count	%
Snippet 1 (Copying /* Block #4 */) (Sample 1)	8	14.8
Snippet 2 (Not copying /* Block #4 */) (Sample 2)	21	38.9
Snippet 3 (Extracting /* Block #4 */ as a function and calling it twice)	48	88.9

- Please explain your choice.

function identifies the cide block as a consistent piece that can be understood on its own without the need of identifying duplicate code

Minimal analysis of potentially long code still takes priority.

the same explanation as before

again: easy breaking condition with complex duplicated code. having code duplication inside function looks nice

Similar reasoning as before, but it is a closer choice that I might pick 1.

The function call makes it very clear when the block is executed and don't obstruct the logic. Snippet 1 also achieve this, but is more bloated than Snippet 3. Snippet 2 does this more implicit, but is still harder to read.

Same as previously, I find the additional logic of the entry variable more confusing to follow. And same as previous questions, extracting as a function can potentially make it more human-readable but may not be the truest representation of the program.

Since the examples are not that complex and long, all 3 options seem to be fine.

Snippet 3 is the best solution due to the simple control flow.

substitute with functions calls makes it easier to read

single line and small (simple) code snippets can be repeated, but long snippets should not be to prevent unnecessary re-analysis of code. Having the code separated out as a function also allowed for cleaner analysis as the function can be analysed as a separate unit

easiest to understand

Same reasoning as the previous part (would prefer not extracting the block as a function)

It's more concise to have everything viewable from a single function. Furthermore, it doesn't create any artificial function calls. Perhaps it would be useful to allow user to collapse the while statement.

Same as earlier

Same as previous

It avoids all the problems of the while-true loop, flags, and comparing long snippets. I would prefer the function to be marked ”inline”, and would not mind a goto-label structure if used once per function.

As mentioned earlier, having to read repeated code blocks in 1 function (Snippet 1) is annoying.

once again, refactoring is lit

Easier to understand a more complex set of instructions when they are extracted out into a separate function

Function extraction is the best choice here, IMHO.

I find the copying version easier to read, and now there is no duplicate code.

snippet 3 is by far my favourite but I'd be okay with analyzing snippet 1

Snippet is not long enough for me to mind the repetition. I would actually prefer snippet 3 if it were not for the implied adherence to a calling convention that may not actually be true.

Like in the previous section, removing gotos leads to worse decompilation results here.

Clean

Snippet 3 will be very clear if the decompiler supports clicking of call_sub to go to the decompiled code of the call_sub function.

Snippet 3 is the clearest and easiest to understand. The repeated block of code is modularized into a function where multiple entries to the block are labelled function calls.

Easiest to follow

It looks like a proper source code. Plus, individual functions are smaller which are easier to analyze.

As above.

Prefer larger/complicated code blocks to not be repeated across the code for quicker understanding of program behaviour.

Extracting the code into a separate function still is the most readable and least complex solution. However, as the overall complexity of Snippet 2 is somewhat equal to the complexity of Snippet 1, Snippet 2 seems also OK.

Similar to the section before.

I chose snippet 3 because it has the easiest structure without any code duplication. Hence, it is easy to read.

Here in snippet 3 makes getting an overview about the functionality of the code the easiest.

given my previous arguments, snippet 3 is best of both worlds.

shorter code and easy to follow structure

Snippet 2 & 3 looks cleaner & readable than snippet 1

same as before

neater and easier to read.

The conditions leading to Block#4 is processed prior to the call.

Switch

- Please consider the following two code snippets.
  Snippet 1:

```
/* ... */
switch(var){
    case 1:
        /* Block #1 */
        break;
    case 2:
        /* Block #2 */
        break;
}
/* ... */
```

  Snippet 2:

```
/* ... */
if(var == 1){
    /* Block #1 */
}
if(var == 2){
    /* Block #2 */
}
/* ... */
```

- Which snippet do you prefer?

Answer	Count	%
Snippet 1	24	44.4
Snippet 2	15	27.8
No preference	15	27.8

- Please explain your choice.

---

for snippet 2 it's not clear whether it's possible to enter multiple if conditions

I assume the compiler did not emit a jump-table and in this case, as there are not even any 'else'-statement it is likely that Snippet 2 is more accurate and has less scoping.

Personal preference (even from a software engineering perspective). Less Indentation too.

snippet 2 reduces one depth of indentation. Feels more natural for me to read.

simple condition blocks are rather lengthy when implemented as switch-case block

snippet 2 is shorter and clear to read

There are not enough cases for me to consider a switch an upgrade yet.

I think the logic in a switch statement is more obvious and nicer to read than if-else blocks.

Both are still quite readable.

Just don't like switch statements.

Snippet 1 highlights the exclusiveness of the two arms which is not immediately visible in snippet 2 due to the missing "else if". Also, it's easier to spot that we are comparing the same variable against multiple constants in snippet 1.

nil

The readability of Switch Cases and Ifs are dependent on the code snippet prior, and which determines the flow of the code. The readability of switch cases/if statements can drastically change based on the value that undergoing the logic statements.

switch statement seems unnecessary. Also not sure if they have the same meaning as snippet 2 can possibly execute both blocks 1 and 2 but snipper 1 cannot.

If statements are more intuitive to me if there are only a few cases.

If statement is more concise when it's just 2 cases.

Case switch presents a more intuitive understanding of the flow of the code.

Two choices I don't think calls for a switch, just a personal preference. I prefer when switch is used for numerous options, but for two I prefer simple ifs.

Both are fine, as they are short and uniform. - The switch-case structure has no weird fall-throughs - The if conditions have no overlaps

neater, less braces :) and if it's a long statement, you'll immediately know that the control flow is determined by 1 variable (var).

I immediately see that I only need to think of the value of var for all of this switch statement

Depends on the length of the switch case/if-else. Switch case might be more appropriate for a greater number of conditions while multiple if-else has better readability when representing a small number of conditional statements

When there are no nested 'else if ()' constructs and only two options a switch isn't really worth the effort

For just two cases, if-elif is better in my opinion.

switch cases make it easy to directly see the structure of the code

the switch statement adds inherent semantics compared to the if conditions, making it easier to glance the purpose of the code

Snippet 1 makes it clearer that the code cannot execute both block 1 and 2 but only one of them.

Both snippets are easy to read. Anything more complex than that should be a switch

Both appear to be plausible, but with only 2 possibilities, Snippet 2 is readable enough.

Easier to read

Easier to understand if there is not too much nested if-else statements.

Both snippets are straight forward and easy to understand.

Easier to read and list the important values of var?

No preference. They look the same to me.

I find switch statements slightly easier to read here, but I'm fine with either.

Easier to determine that the case statements depend on the same variable, especially when there are a lot of conditional (case/if-else) statements.

The Switch-Statement and the If-Statements appear to be equally complex. Thus, no real preference.

No preference if the number of differentiating conditions (2 in this case) are small.

Although switch case statements might be a little lengthier than the if statements from snippet 2, they have the better and cleaner structure. In specific, any case refers to the same variable, whereas in the multiple if statements this can be potentially mixed up and therefore difficult to read. Furthermore, the switch statements encapsulate all cases so that it's obvious, when the parsing ends.

Here both are equally readable and the if construction is more familiar. But in the first example in the survey with the multiple nesting layers for the (else) if structures, these became quite confusing and the switch case construction would be preferable.

I'd say this depends on the length of block #1 and #2, which is not shown. Otherwise I probably like snippet 1 more because the switch can be understood as one big coherent block, while I would have to check in snippet 2 if var is modified in a way that would lead to execution of block #2.

similar in length and both snippets are still easy to comprehend

Looks cleaner & condition only validate once

switch cases are a great way to express the connectedness of branches, hence I prefer it.

if..else for switch case is easily readable, no difference.

Switch statement contains less visual artefacts (it's neater)

in short switch case while 'switch(var)' is visible it's good to have snippet one

---

- Please consider the following two code snippets.
  Snippet 1:

```
/* ... */
switch(var){
    case 1:
        /* Block #1 */
        break;
    case 2:
        /* Block #2 */
        break;
    case 3:
        /* Block #3 */
        break;
    case 4:
        /* Block #4 */
        break;
    case 5:
        /* Block #5 */
        break;
}
/* ... */
```

  Snippet 2:

```
/* ... */
if(var == 1){
    /* Block #1 */
}
if(var == 2){
    /* Block #2 */
}
if(var == 3){
    /* Block #3 */
}
if(var == 4){
    /* Block #4 */
}
if(var == 5){
    /* Block #5 */
}
/* ... */
```

- Which snippet do you prefer?

Answer	Count	%
Snippet 1	37	68.5
Snippet 2	6	11.1
No preference	11	20.4

- Please explain your choice.

  for snippet 2 it's not clear whether it's possible to enter multiple if conditions

  Same reason as above.

  Personal preference (even from a software engineering perspective). Less Indentation too.

  the more constant cases for a single var, the more natural it feels to me to use and, thus, read, a switch case

  same as above

  snippet 2 does not look bad, since conditions are easy and we have no nesting. snipped 1 makes it clear that we are checking for var. no preference

  Here there are enough cases that a switch is more readable.

  I think the logic in a switch statement is more obvious and nicer to read than if-else blocks.

  Both are still quite readable.

  Just don't like long switch statements with blocks in it.

  Snippet 1 highlights the exclusiveness of the two arms which is not immediately visible in snippet 2 due to the missing "else if". Also, it's easier to spot that we are comparing the same variable against multiple constants in snippet 1.

  nil

  The readability of Switch Cases and Ifs are dependent on the code snippet prior, and which determines the flow of the code. The readability of switch cases/if statements can drastically change based on the value that undergoing the logic statements.

  Looks cleaner

  If there are many cases, switch is easier to understand. I would know that a block of operations will be performed based on the value of var in one glance.

  Switch is easier to read when there are many cases.

  Case switch presents a more intuitive understanding of the flow of the code.

  I prefer when switch is used for numerous options: for 5 i am okay with both versions, personal preference.

  Both are fine, as they are still uniform. - The switch-case structure has no weird fall-throughs - The if conditions have no overlaps

  neater, less braces :) and if it's a long statement, you'll immediately know that the control flow is determined by 1 variable (var).

  I immediately see that I only need to think of the value of var for all of this switch statement

  switch case for a bigger number of conditional statements is easier to read and understand

  because of the sheer number of cases, a switch statement improves readability here IMHO

  switch cases make it easy to directly see the structure of the code

  same as above, the switch statement adds inherent semantics compared to the if conditions, making it easier to glance the purpose of the code

  Snippet 1 makes it clearer that only one of the blocks get executed.

  Since every condition uses var, it only makes sense to use a switch

  While Snippet 1 is probably closer to original source code, Snippet 2 is less complicated, so I don't have a strong preference either way

  Easier to read

Even though it is longer, it is still easy to understand.

With more if conditions regarding the same variable, a switch-case statement is more readable and neater.

Easier to read and list the important values of var?

No preference. They look the same to me.

As above,switch statements slightly easier to read here, but I'm fine with either.

Easier to determine that the case statements depend on the same variable, especially when there are a lot of conditional (case/if-else) statements.

Switch-Statement and If-Statements contain more cases, but the overall structure seems to be equally complex.

For larger number of conditions, a switch-block makes the code easier to read over a series of if-blocks.

Same reason as above: Although switch case statements might be a little lengthier than the if statements from snippet 2, they have the better and cleaner structure. In specific, any case refers to the same variable, whereas in the multiple if statements this can be potentially mixed up and therefore difficult to read. Furthermore, the switch statements encapsulate all cases so that it's obvious, when the parsing ends.

Again, both are almost equally readable, but the switch case construction shows more clearly that all of the following cases/ if statements will be dependent on the value of this one variable, which gives an overview over the code functionality more quickly.

for the same reason that the break statements in snippet 1 suggest a single case being taken at most, this is easier to follow than snippet 2 which may have code in the blocks that may lead to execution of more than one if-body.

similar in length and both snippets are still easy to comprehend

Looks cleaner & condition only validate once

same as above

if..else for switch case is easily readable, no difference.

Switch statement contains less visual artefacts (it's neater)

in case when lot of blocks it's good to see 'var == x'

- Please consider the following three code snippets.
  Snippet 1:

```
/* ... */
switch(var){
    case 1:
        /* Block #1 */
        break;
    case 2:
        /* Block #2 */
        break;
    default:
        /* Block #3 */
        break;
}
/* ... */
```

Snippet 2:

```
/* ... */
if(var == 1){
    /* Block #1 */
}
if(var == 2){
    /* Block #2 */
}
if(var != 1 && var !=2 ){
    /* Block #3 */
}
/* ... */
```

Snippet 3:

```
/* ... */
if(var == 1){
    /* Block #1 */
}else{
    if(var == 2){
        /* Block #2 */
    }else{
        /* Block #3 */
    }
}
/* ... */
```

- Which snippet do you prefer?

Answer	Count	%
Snippet 1	46	85.2
Snippet 2	3	5.6
Snippet 3	0	0.0
Both if-Snippets	2	3.7
No preference	2	3.7
No answer	1	1.9

- Please explain your choice.

---

switch case is easy to read and default condition remains the same during growth of switch cases

If Snippet 2 accuratly represents the underlying code, meaning there are no else jumps and in the end it tests for all other conditions being wrong, then Snippet 2 is the best. Otherwise, if the code is an if-else-chain or a jump table, I would prefer Snippet 1.

Personal preference (even from a software engineering perspective). Less Indentation too.

I think a "default" case is the more expressive choice here (snippet 1). The two-part condition of snippet 2 takes more "horizontal-visual-eye-parsing" and active reading to come to the conclusion that it is basically the default case. Snippet 3 has too much nesting.

here it's a special case, it leads to less indentation

snippet 1 beats both "complex" condition and nesting

I think the switch is easier to understand as there is less nesting and clearer conditions, but it is close due to small number of cases.

The switch statement is nicer to read.

Snippet 1 is the easiest to follow, snippet 2 has additional logic to process, snippet 3 has too much nesting.

Same. Snippet 3 contains unnecessary nesting which complicates reading it.

Snippet 1 highlights the exclusiveness of the two arms which is not immediately visible in snippet 2 due to the missing ”else if”. Also, it’s easier to spot that we are comparing the same variable against multiple constants in snippet 1. In this case, the switch also has the simplest control flow.

switch-case is most readable out of the 3 choices

Switch case looks the cleanest and directly readable in this snippet

Looks cleaner

In snippet 1, have to perform the additional step of analyzing what the default case mean (var != 1 && var !=2). This means I will comment it at line 9 anyway. Thus I would prefer having the condition explicitly written out in snippet 2 like in line 8. Snippet 3 seems unnecessarily convoluted.

Switch is easier to read.

Case switch presents a more intuitive understanding of the flow of the code.

The default case for switch looks cleanest and most straight forward to me, the ifs just complicate things.

The final none-of-the-above case corresponds well to the ”default” part in switch-case. Other than Snippet 1, I would also be fine with Snippet 3, as the else case rules out overlapping conditions, unlike in Snippet 2 where I need to double-check.

neater, less braces :) and if it’s a long statement, you’ll immediately know that the control flow is determined by 1 variable (var).

I immediately see that I only need to think of the value of var for all of this switch statement (why no if - else if - else?)

if-else better for 3 conditional blocks

Snippet 2 is just ugly and snippet 3 is unnecessarily convoluted with its nested if statements. Furthermore, a default block in a switch statements is very intuitive and therefore easy to comprehend

switch cases make it easy to directly see the structure of the code

same as above, plus defaults are easier to understand with the switch statement

I generally like the readability of switch cases. Snippet 2 is the worst of the three, since one first has to check that block 1 and 2 do not change the variable before one knows that the code is equivalent to the other two. This means more analysis work compared to snippets 1 and 3.

Intuitively understandable code

This is clearly a switch variable and Snippet 1 best reflects that.

Easiest to understand the conditions

If there are more conditions in the if statements, I prefer the switch-case’s default block.

Snippet 1 looks neater and easier to understand. Snippet 2 looks too confusing with the last condition, especially when dealing with more conditional statements. Likewise for Snippet 3, the nested conditions will look horrible with many conditional statements.

Easier to read and list the important values of var?

Snippet 2 and 3 does not look right. Another alternative i would prefer is if (var ==1) else if (var == 2) - else()

As above.

I find it easier to understand the conditions under which a code block will be executed when placed within switch-case statements. Having a default case makes it clearer that a specific code block is executed when all other conditions are false.

Snippet 3 contains nested branching, which seems to have the most complex structure compared to the other Snippets. Snippet 2 shows the same level of branching as Snippet 2. Actually, there is no real preference between Snippet 1 and Snippet 2, but I stick to Snippet 1 as it handles the ”default” case in a more elegant way.

In this case, the switch-block makes the decompiled code easier to read. There is no need to run through individual if-statements only to know that Block #3 is the ”default” case.

I still prefer snippet 1, because of its structure. Although snippet 3 has a better structure than snippet 2, it might be potentially difficult to read because it gets more and more nested for each additional IF-ELSE block.

I don’t think the nested if/ else statements are easy to read. The second snippet is also okay, but line 8 would most likely become less readable for more previous cases7 if statements, so snippet 1 is easiest to read.

snippet 1 with the same arguments as above (easy to follow thanks to the breaks) and otherwise snippet 3 over snippet 2 if the nesting remains on this level.

easier to follow

Looks cleaner & condition only validate once

same as above

all snippets are still readable. no preference so long as the conditional statements are straight forward .

It is more succinct.

- Please consider the following three code snippets.
  Snippet 1:

```
/* ... */
switch(var){
    case 1:
        /* Block #1 */
        break;
    case 2:
        /* Block #2 */
        break;
    case 3:
        /* Block #3 */
        break;
    case 4:
        /* Block #4 */
        break;
    case 5:
        /* Block #5 */
        break;
    default:
        /* Block #6 */
        break;
}
/* ... */
```

Snippet 2:

```
/* ... */
if(var == 1){
    /* Block #1 */
}
if(var == 2){
    /* Block #2 */
}
if(var == 3){
    /* Block #3 */
}
if(var == 4){
    /* Block #4 */
}
if(var == 5){
    /* Block #5 */
}
if(var != 1 && var !=2 && var != 3 && var != 4 && var != 5){
    /* Block #6 */
}
/* ... */
```

Snippet 3:

```
/* ... */
if(var == 1){
    /* Block #1 */
}else{
    if(var == 2){
        /* Block #2 */
    }else{
        if(var == 3){
            /* Block #3 */
        }else{
            if(var == 4){
                /* Block #4 */
            }else{
                if(var == 5){
                    /* Block #5 */
                }else{
                    /* Block #6 */
                }
            }
        }
    }
}
/* ... */
```

- Which snippet do you prefer?

Answer	Count	%
Snippet 1	51	94.4
Snippet 2	2	3.7
Snippet 3	0	0.0
Both if-Snippets	0	0.0
No preference	1	1.9
No answer	0	0.0

- Please explain your choice.

---

switch case doesn't need to be nested and the default condition remains slim

Same reason as above.

Both snippet 1 and 2 are fine since the indentation level is relatively significant for snippet 3 here.

Snippet 3 is bad because of the high nesting Snippet 2 is bad because the multi-part conditional is way too canonical and complicated.

same as above + less logic inside the last if statement

eating complex conditions and nesting up in a default case looks clean

Much easier to read the switch statement, handles the larger number of cases easily.

I still prefer the switch statement. It contains fewer brackets, and it is very clear in which case what happens. Snippet 3 is very complex due to the nesting, and Snippet 2 is better suited for a switch statement.

Snippet 1 is the easiest to follow, snippet 2 has additional logic to process, snippet 3 has too much nesting.

Same. The default state is nice though, but could be achieved with if/else if/else which would be more logical.

Snippet 1 highlights the exclusiveness of the two arms which is not immediately visible in snippet 2 due to the missing "else if". Also, it's easier to spot that we are comparing the same variable against multiple constants in snippet 1. In this case, the switch also has the simplest control flow.

switch-case is most readable out of the 3 choices

Switch case provides the most readable output for this snippet.

Looks cleaner

Would prefer snippet 1 over snippet 2 as the last condition (line 17) seems too long for it to be intuitive.

Switch is more concise. Nested if statements makes it hard to read.

Same as earlier, even more so for this case. It simplifies the structure massively

Again, with default cases, switch should probably be used

The final none-of-the-above case corresponds well to the ”default” part in switch-case. Snippet 2 has a messy final case with too many conditions to eyeball. Snippet 3 has too much nesting, but it would work fine if the formatting allowed ”else if” to be on the same line and not indented.

neater, less braces :) and if it’s a long statement, you’ll immediately know that the control flow is determined by 1 variable (var).

I immediately see that I only need to think of the value of var for all of this switch statement

Easier to understand

Is there even a choice here? Snippets 2 and 3 are gruesome in comparison.

switch cases make it easy to directly see the structure of the code

same as above, still in favour of the switch, only even worse if snippets than before

I still like the readability of switch-cases. Here snippet 3 is the worst since it reaches a nesting depth that greatly hurts readability.

Intuitively understandable code

As in the previous example

Easiest to understand the conditions

Same reasoning as above.

Snippet 3’s nested conditions looks like an eyesore, especially when the blocks of code inside are huge. I will not be able to tell the levels of indentation where my block belongs. Snippet 2’s last condition is a mess, and very confusing. Snippet 1 is the easiest to understand

Easier to read and list the important values of var?

Same as my previous explanation

As above.

I find it easier to understand the conditions under which a code block will be executed when placed within switch-case statements. Having a default case makes it clearer that a specific code block is executed when all other conditions are false. It also avoids too much nesting when there are many possible cases.

Snippet 3 contains even more nested branches. The expression to handle the ”default” case in Snippet 2 is too long. The Switch-Statement appears as it has been made for this kind of branching.

I’m not a fan of nested if-blocks. Switch-blocks are much easier to read in this case, given that the matching condition is a single variable. However, in the event where the if-condition has multiple variables, ”if (a && b ||c)” etc., then it will be difficult to represent the code in switch-blocks. In such cases, if-blocks would be preferred. As mentioned earlier as well, the switch-block is preferred here as it is easy to identify the ”default” case.

As already explained above, I like the switch statement the most. Although snippet 2 is less nested and a bit shorter, snippet 1 is better to read. Especially the “default” case is clearer than the long IF-statement from snippet 2 at the end (line 17 / 18)

The nested if else statements are really hard to read, especially if there are a lot of them and one has to work on a smaller screen, and line 17 in the second snippet becomes too long. Snippet one is the easiest and fastest to read.

switch is preferred in this structure. snippet 2 is at least more flat and indentation is really painful once those single-line ”block #n” have several instructions.

the other two snippets have too much conditions and if-else statements

Looks cleaner, condition only validate once & code more understandable

wow, same as before!

all snippets are still readable. no preference so long as the conditional statements are straight forward .

It is more succinct.

Conditions

- Please consider the following code snippets which all represent the same semantics and rank them from your favourite to least favourite.
  Snippet 1:

```
/* ... */
if( (a == 5) && (c <= 3d-4) && ( ((x+y) < (z
↪  <<2)) || (u > 20) ) ){
    // do something
}
/* ... */
```

  Snippet 2:

```
/* ... */
if(a == 5){
    if(c <= 3d-4){
        if(((x+y) < (z <<2)) || (u > 20) ){
            // do something
        }
    }
}
/* ... */
```

Snippet 3:

```
/* ... */
bool cond1 = a == 5;
bool cond2 = c <= 3d-4;
bool cond3 = ((x+y) < (z <<2)) || (u > 20);
if( cond1 && cond2 && cond3 ){
    // do something
}
/* ... */
```

Count	Ranks		
%	1	2	3
Snippet 1	13 35.1	13 35.1	18 48.6
Snippet 2	7 18.9	28 75.7	19 51.4
Snippet 3	34 91.9	13 35.1	7 18.9

- Please explain your choice.

---

snippet 2 doesn't introduce unnecessary variables ot if statements

In general I think reducing the amout of indentation and the amount of parens is always good. Though the Snippets are small enought that it does not really matter.

Snippet 3 highlights the requirement for conjunction well, with each condition displayed clearly. Snippet 2 implicitly indicate conjunction well due to nested ifs, which comes at the cost of additional indentation. Snippet 1 personally is the messiest at first glance.

Snippet 3 works towards me: The relationship between the conditions is clear. I'd then continue, try to understand and then rename them for more expressiveness. For me, snippet 1 would be one manual step away from snippet 3 $->$ I'd associate vars and then continue with the same approach used with snippet 3. Snippet 2 simply sucks: it adds unnecessary nesting and makes reading it so hard, as the conditions are non-trivial an confusing by theirselves

clarity

in snippet 1 I get lost in long condition. snippet 2 and 3 tie in ease of following logic

2 is a bit too much nesting for just a single condition, 1 and 3 are reasonably readable (comparable).

The extraction of the conditional components in Snippet 3 makes it easier than Snippet 1 to understand the result of the condition. Snippet 2 also does this, but introduces mare nesting, which can be harder to read if the code inside is also complex and adds additional nesting.

I think my preferences of the 3 snippets are not really very far apart, but I least like snippet 2 due to the nesting, and probably prefer snippet 1 as that would be closest to how I would write the code.

3: wordy but easy to read 2: easy to read but very nested 1: not nested but difficult to read

If the third condition was more complex, I'd prefer snippet 2 over 1, but in this case it's simple enough to be easily readable in snippet 1. Snippet 3 is very unnatural, snippet 2 is fine but the nesting is unnecessary.

Tidy

Code is most verbose and has semantics

Easiest to understand

Most prefer snippet 1 as I like to have the conditions in one view. Snippet 3 helps break down the analysis, but I would prefer everything in one place. Least prefer snippet 2 as I don't like nested if statements.

Snippet 3 allows user to rename the conditions to make it clearer. Snippet 2 is clearer than 1 since it breaks up the conditions.

Snippet 3 most succinctly describes the conditional arguments in an intuitive manner, stating each condition before chaining them together. For similar reasons, Snippet 2 is preferred over snippet 1 despite containing nested if-else statements

I dislike having too many conditions in one line, so Snippet 1 is my least favourite. Snippet 3 is my favourite as it clearly marks out each of the 3 conditionals, allowing me to consider them more easily.

I am used to Snippet 1 where all conditions are crammed into one "if", but I like Snippet 3 for extracting booleans that I can label. For Snippet 2, nesting is unnecessary when there is no else case.

snippet 3 is way easier to read because the conditions for the if-statement is very clearly defined.

top choice makes it clear that we have 3 conditions, what they are, and that all have to be true. Bottom choice has too much in one line

Sufficient verbosity and readability tradeoff

Snippet 1 is harder to read. Snippets 2 and 3 are both OK, I guess. Snipped 2 looks better but #3 uses less nesting. I prefer #2 a bit. because storing the values in variables distracts a bit from program flow.

1 is barely readable, and 3 breaks everything down in seperate parts, making it easier to understand

snippet 3 is easiest to glance, while snippet 2 hides the inherent and-ing of the conditions

I actually would prefer snippet 4 here (inspired by what the standard code formatter for the Rust language does): if((a == 5) && (c <= 3d-4) && (((x+y) < (z <<2)) ||(u > 20))) { // do something }

I like the concise if statement of Snippet 3. The conditions are simple enough that I prefer their conjunction over nested if statements.

Clean

For Snippet 1, there are too many conditions and

parentheses and the reverse engineer might get lost or read something wrongly due to operator's precedence. For Snippet 2 and 3, it breaks the condition into smaller chunks and is easier to understand.

Snippet 2 is more readable and easy to understand. Snippet 1 requires me to understand all the condition at once which can be quite confusing. Snippet 3 requires me to calculate the conditions before going into the if statement.

Snippet 3 seems easiest to follow and know the conditions use distinct variables. Snippet 1 is ok (for this example), can tell it's "all or nothing" Snippet 2 for me, nested ifs are visually hard to read (bracing for nested elses)

I dislike nested instructions. Snippet 1 and 3 looks cleaner although i would much prefer snippet 1 as it looks more like a proper source code.

Snippet 3 is much easier to read compared to the first 2. In this case, I might be accessing each of the conditionals separately. Having each of the conditionals as separate intermediate variables lets me rename them accordingly. I can do something similar in snippet 2 by commenting on each line. Snippet 1 is the worst because inlined conditionals make it difficult mark which check is doing what.

Fewer lines of code, especially when there are no else clauses for each of the conditions, are preferred as they help make code more readable and quicker to understand.

Snippets 3 & 2 are tied. They make reading the conditions necessary to enter the loop easier to interpret.

Snippet 1 has a single but very complicated boolean statement, so that is hard to understand. Snippet 2 is easy to read, but the longer the statement gets, the more nested the function will be. At some points, very nested function also might be difficult to understand. I like snippet 3 the most, as it consist only of few, simple boolean functions, which are eventually combined by simple operations (AND).

Snippet 1 seems most familiar and easiest at first glance, but the nested parentheses are hard to read. Snippet 2 and 3 are easy to read. While the nested if statements in 2 would probably become harder to read if there were even more of them, one only has to read them once when looking through the code, whereas in snippet 3 in line 5 one has to remember the contents of lines 2-4.

Thinking about visual flow, snippet 3 has the best presentation of the conditions, while snippet 2 at least breaks them also down on individual lines.

having all the conditions in one line is too much

Snippet 3 looks cleaner without the nested if else compared to snippet 2 & easier to evaluate compared to snippet 1

Snippet 3 is very easy to read and I have the opportunity to rename the three 'cond' variables, hence it's ranked highest. Snippet 2 is deeply nested just for the sake of a logical and, hence I prefer Snippet 1 over it even though Snippet 1 has a hard-to-parse arithmetic expression (it was a close call between 1 and 2, both are relatively hard to read).

Snippet 3 is easier to read.

makes it easier to follow conditions

They are ranked in order of clarity of the conditions. Nested if-else are hard to read.

For-loops

- Please consider the following code snippets which all represent the same code and rank them from your favourite to least favourite.
  Snippet 1:

```
/* ... */
for(i = var_0; i > 10; i = (i * 3257 >> 5 >>
↪  2) - (i >> 32)){
    /* Block #1 */
}
/* ... */
```

  Snippet 2:

```
/* ... */
c = car_0;
while(c > 10){
    /* Block #1 */
    c = (c * 3257 >> 5 >> 2) - (c >> 32);
}
/* ... */
```

  Snippet 3:

```
/* ... */
for(i = var_0; i > 10; i = x - y){
    /* Block #1 */
    x = i * 3257 >> 5 >> 2;
    y = i >> 32;
}
/* ... */
```

  Snippet 4:

```
/* ... */
for(i = var_0; i > 10; i = x - (i >> 32)){
    /* Block #1 */
    x = i * 3257 >> 5 >> 2;
}
/* ... */
```

Count	**Ranks**			
%	1	2	3	4
Snippet 1	17 45.9	7 18.9	10 27.0	19 51.4
Snippet 2	14 37.8	19 51.4	11 29.7	10 27.0
Snippet 3	17 45.9	17 45.9	10 27.0	9 24.3
Snippet 4	6 16.2	10 27.0	22 59.5	15 40.5

- Please explain your choice.

---

the update on the while loops variable seems the most understable

It somewhat depends on what the actual code is. If there is a variable in the code that caculates x and y I might be persuaded to favor Snippet 3.

Assuming the incrementation amount is not affected by Block #1, it would be best preferred for the amount to be in the for loop.

Snippet 2 feels most natural for me. Snippet 3 is a nice and short for-loop, though I don't like updating x and y in the inner loop scope. Snippet 1 is LESS expressive than 3, but replaces the x-y update of snippet 3. Yet snippet 3 wins over 2 because of overall clarity. Snippet 4 mixes 1 and 3 and is "all over the place". I have to watch out for non-trivial variable updates in both the inner loop and the official iteration update. No bueno.

clarity

I prefer easy loop headers (with complexity in body)

I think the update is not complicated enough to warrant splitting out of the loop head. But if it is split out, i prefer 2 the most as everything is in one place. 3 and 4 are close, maybe tied.

I think a for-loop is easier to read in this case as a while-loop. And to extract a complex operation from the head of the loop increases the readability of the loop.

1 and 2 are quite clear, and I personally feel for-loops are easier to read. 4 and 3 tends to be a bit confusing as the for-loop operation occurs within the for-loop itself.

1: it's a bit difficult to read, but easy to see, that i is calculated without other dependencies 2,3,4: more easy to read but the additional variable(s) in the loop and different places of calculation complicates up reading it.

Having the loop increment statement reference variables that are set in the loop is hard to follow (snippet 4 and 3), having a while loop instead of a for loop can also be harder to read (You have to watch out for continues to make sure the loop counter changes in each iteration).

Neat

Snippet 3 was easily understood, while snippet 4 required determining the logic within the for loop. Snippet one was essentially code-golfing (not essential), and while loop is harder to understand without extra effort into anlysing it.

Closest to how I would implement in C

Prefer for loops the most as I could explicitly see when the loop terminates. Prefer the for loop in snippet 1 the most as I could easily see how the loop variable i updates in one statement.

For statement is more concise that while statement. Separating the conditions into x and y variable allows user to rename them.

Snippet 1 most clearly states the conditions of the for loop in a single sentence without additional confusing variables. Snippet 2 is deemed the worst since the changes in c can easily be lost in the block of code in between

Snippet 1 least favourite because it is all in one line, hardest to parse. Snippet 2 and 4 is next because I prefer to see more than 1 variable to manage the loop counter, but 4 is a bit confusing still. Snippet 3 is the cleanest to me.

Snippet 4 seems to have the best balance between simplifying the update statement and introducing new variables. I prefer for loops to while loops, as it's easier to find the update statement(s).

putting the loop exit condition logic together makes it easier to read the code. It gets harder/more annoying when they are separated like in Snippet 4 (x - (i >> 32)). Then I have to look at what is 'x' at the bottom of Block #1. Then it turns out that x is related to i as well... Might as well put them together.

I hate for loops with anything else but i++ or i–

How easy it is to understand the logic

Having complex operations inside the for loop definition is a bad idea IMHO. What is more, I think while loops have better readability most of the time. By substituting most of the operations, #3 is "the best of the rest". P.S.: Why the '>> 5 >> 2'? Isn't that just the same as '>> 7'?

i prefer the least amount of logic possible inside the loop definition

snippet 1 is too complex in the for header. I generally prefer for-loops over while, but with a complex update such as this I prefer the while loop. snippet 3 is okay but I feel like the update of a for-loop should be self-contained, i.e. not partially in the for-header and partially in the body. Snippet 4 has the same problem as snippet 3 but is also more complex in the for-header

Snippets 3 and 4 split the loop variable handling between the top and the bottom of the loop, which is bad. Snippets 1 and 2 have the same readability to me, here I can only say on a case-by-case base which one I prefer.

I prefer Snippet 1 and 2, which keep the statements involving the loop variable 'together'.

Simpler to understand

For Snippet 2, the While Condition is very clear and one needs to only calculate c and check if it is greater than 10.

Snippet 3's for loop has an easy to understand (x-y) update statement, although the calculations for x

and y are rather confusing Snippet 2 is a simple one-liner that is also easy to understand Snippet 4 has a rather intimidating (x - (i >> 32)) update statement, which is made worse when x also has a rather confusing calculation. Snippet 1 looks the most unpleasant because the update statement is long and confusing.

Personally prefer if the loop looks as simple as possible? For Snippet 2, at least the calculation has its own line Prefer 1 over 3 and 4, to avoid splitting across multiple lines and adding new variables

Snippet 1 is simply too complicated. It is easier to understand the loop condition from snippet 3 and 2.

Snippet 3 is much easier to process by creating intermediate variables x and y and operating on them within the loop brackets. Snippet 2 is second, with a similar reason as above. Snippet 4 is similar to 3, but not as good because it only creates 1 intermediate variable x - think it is better to commit to either having the operation within the loop syntax or not. Snippet 1 is difficult to analyze because it creates a lot of clutter on the for loop syntax.

Snippet 1 allows quicker understanding of what the number of rounds in a for loop is dependent on. Snippet 3 is better than snippet 4 as the values for x and y are placed together, allowing for quicker deriving of how variable i is calculated.

Snippet 4 appears to be the most intuitive solution as one would expect a for-loop-statement to be (even though the update statement is complex). Considering that the update statement is complex, Snippet 3 is the next favorable solution: Rather simple update statement, values are determined in the for-loop. Even though Snippet 2 is no for-loop, the while-loop looks like it was written by human beings. Snippet 4 appears to be the least favorable solution as the Update-Statement is a mixture of a value determined in the loop and the former iteration value. This is a mixture of Snippet 1 and Snippet 3.

Snippet 1 clearly shows the for-loop's continuation and increment conditions. However, if they become too complicated, breaking them down into that shown in Snippet 3 can be handy. However, in Snippet 4, it is neither within the for-loop's brackets nor extracted fully at the end of the loop. This feels half-baked and somewhat confusing to follow. The use of while-loops in Snippet 2 requires the initialization to be done outside and before of it and is least favourable for me.

Shifting, especially inside the condition of the FOR-loop, is difficult to understand. So, it might be unclear, over which exact elements are iterated. Therefore, snippet 3 and 2 are the best readable snippets. I prefer snippet 3 over snippet 2 because the iteration step is a bit more clear. Snippet 1 is the in my opinion the worst as the iteration step consists of multiple complicated statements, including shifts, and is therefore hard to interpret.

i think the for loop is easier to read in this case. I find splitting up the i update in snippet 4 more confusing than helpful, since now one has to look in 2 places. Either snippet 3 or snippet 1 are fine. While in snippet 1 the functionality of the for loop is more clearly stated in one place, snippet 3 is easier to read. Here the structure, length and content of Block #1 would most likely also influence the readability, i.e. snippet 3 would be less favorable than snippet 1 if block #1 was long and updates to x and y somehow hidden in the block.

if I see "for" my first thought is some action repeated in a sequence for <n> times. While on the other hand implies a repetition for as long as a condition is met. That makes snippet 2 feel more natural here. Snippet 1 at least keeps the condition out of the body, 3+4 are a mix where need to read and think about too many lines to understand what and why this condition is updated.

having too much calculations at the for loop makes the code harder to follow

It's easier to understand if the condition is modified outside of the loop rather than putting it all in one line

for loops can transport a lot more meaning by basically deciding that a certain variable is a counter. Hence I prefer snippets with for loops over the one with the while loop. I ranked the for-loop snippets by how well the decompiler as able to distinguish in-loop calculations from progression-step calculations: Snippet 4 looks the cleanest there.

no exact preference between snippet 1 and 3 as they are the same and easy to distinguish the conditions and the instructions after the loop. snippet 4 makes it easy to make mistakes by splitting the instructions between the code block and the for end-loop

Conditions of the loop are not diluted within the loop.

D. Feedback

- Thank you very much for getting this far and also for participating in any previous surveys! This is very likely to be our last decompiler survey for now.
- If you would like to leave us any feedback about the survey, please use the lines below to help us improve ourselves.

---

some preferences don't necessarily only relate to the mainly investigated issue of the page

It would be nice to have some kind of auto-save to our survey token, as both times I accidentally exited before finishing or saving and had to redo it. Thanks for the effort and good ideas!

Thanks for the opportunity.

I feel like using the /* BLOCK x */ stuff skews

perception of the snippets heavily

The usability was very good IMHO

I think I already mentioned that looking at what the standard Rust code formatter does for way too complex code constructs could also be useful for decompiler output. Sometimes smart intendation of code helps more in the way of readability than rewriting the code with a different control flow style.

Most of the code snippets where very similar. I would suggest to show more different examples. Furthermore, it was unclear what the intention of the code supposed to be. If more context was shown, choosing which option is preferable would be probably easier.

good luck on the paper submission! :)

very interesting survey, made me think about my preferences and why I have them!

great piece of work! look forward to using it to analyze codes

---