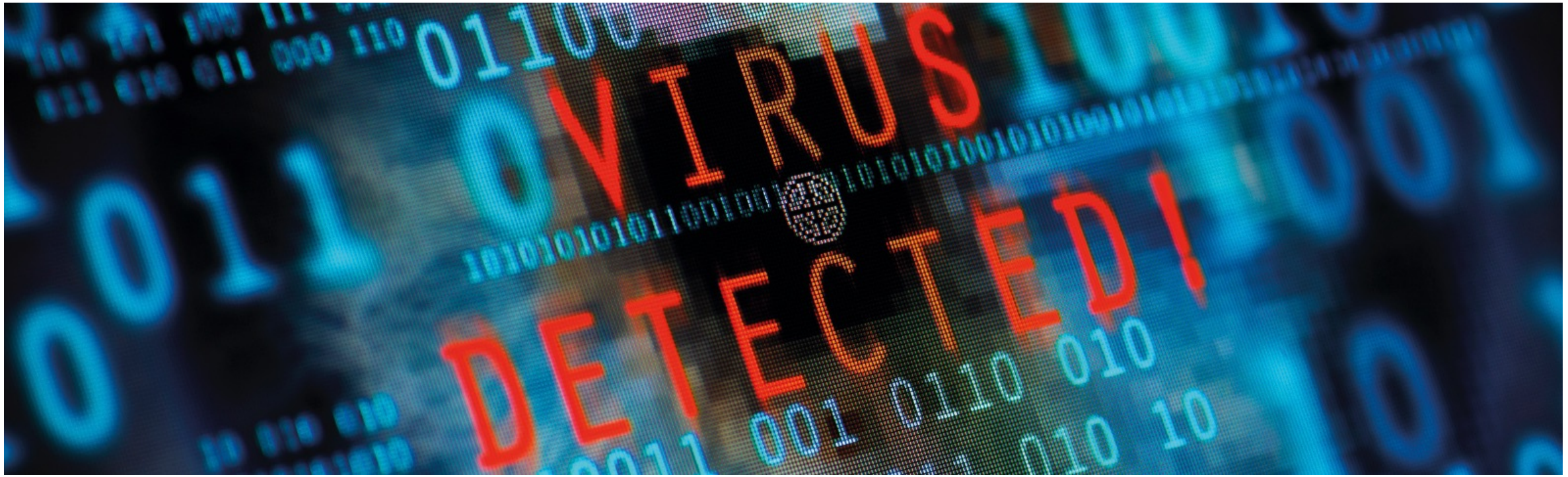
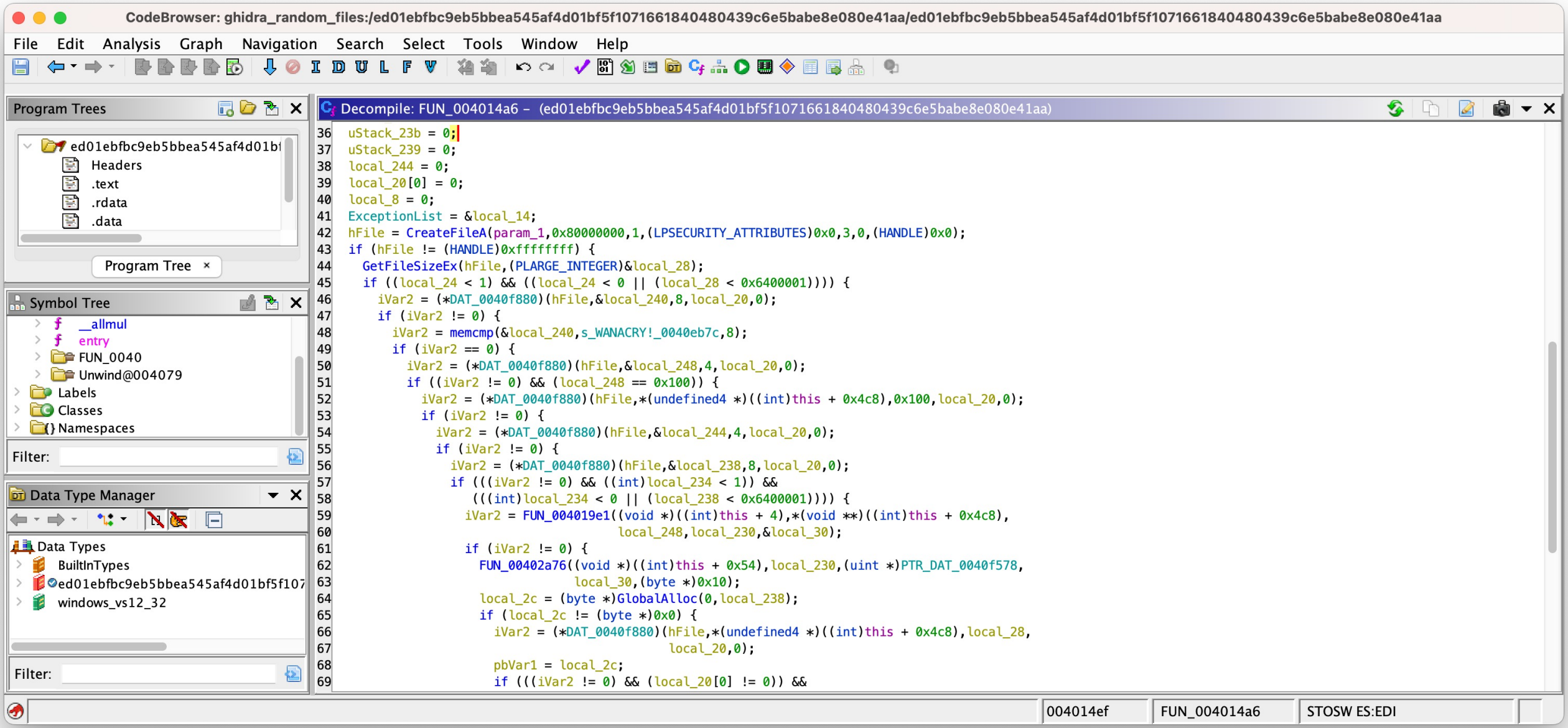
DEWOLF: IMPROVING DECOMPILATION BY LEVERAGING USER SURVEYS

Steffen Enders, Eva-Maria C. Behner, Niklas Bergmann, Mariia Rybalka,
Elmar Padilla, Er Xue Hui, Henry Low, Nicholas Sim
steffen.enders@fkie.fraunhofer.de

2023-03-03 | BAR Workshop, San Diego



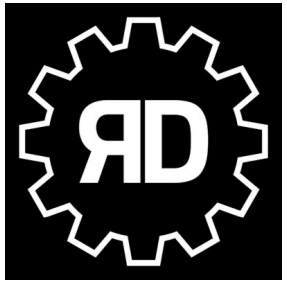
Fraunhofer FKIE



Related Work / Decompilers

- Essential for many binary analysis approaches
- **Each decompiler has a different focus!**

USECAP/**dream**



- There is still plenty of room for improvements!

https://github.com/revng
https://www.gbssoftware.com/jeb-decompiler.html
https://mobile.twitter.com/retdec
https://ghidra-sre.org
https://hex-rays.com
https://github.com/USECAP/dream
https://binary.ninja

Decompilation can be done differently!

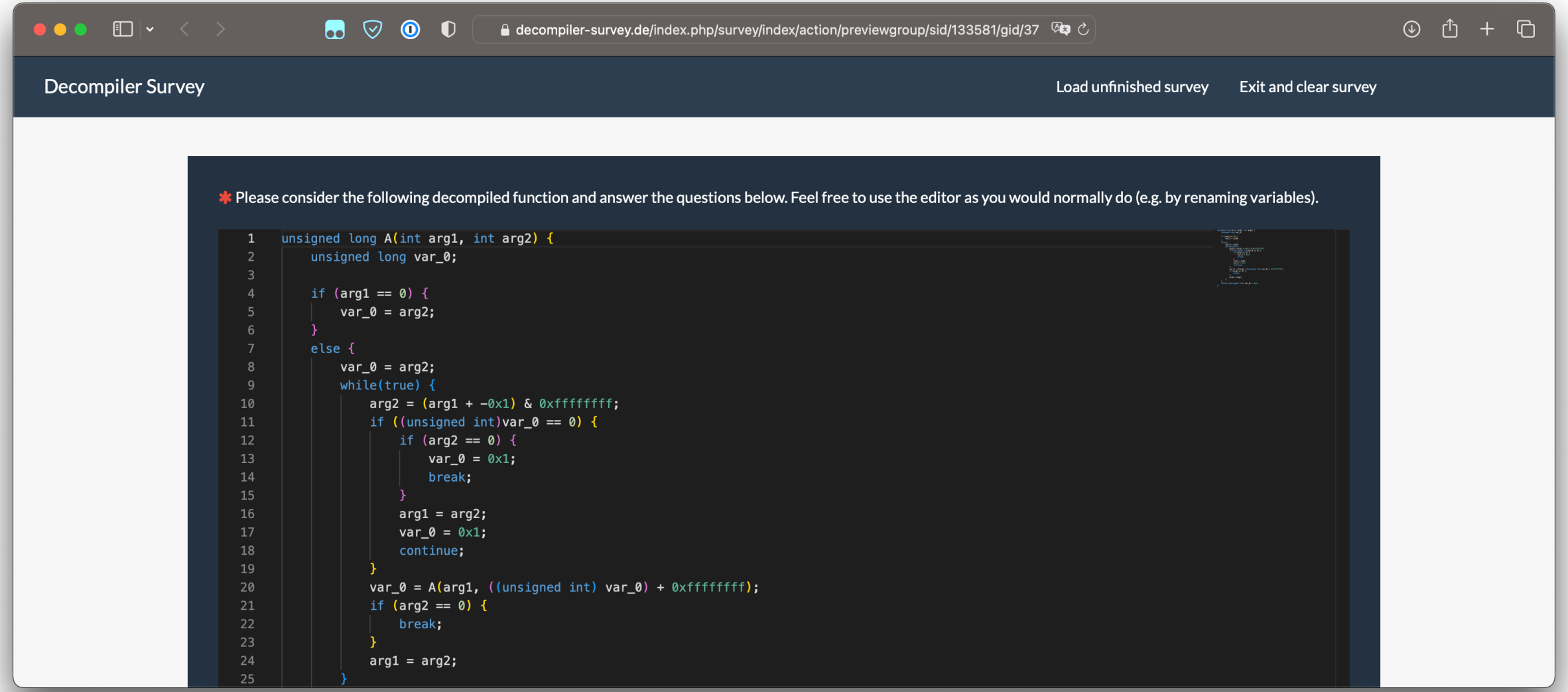
Output 1

```
undefined4 dividability_rules(void) {
  int iVar1; int local_10 [3];
  printf("Enter a number: ");
  __isoc99_scanf(&DAT_0804c025,local_10);
  iVar1 = local_10[0];
  printf("A number is dividable by %d:\n",local_10[0]);
  if (local_10[0] == 0x7d) {
    printf("last three digits dividable by 125",iVar1);
    return 0; }
  if (local_10[0] < 0x7e) {
    if (local_10[0] < 0xb) {
      switch(local_10[0]) {
        case 0:
          printf("not possible",iVar1);
          return 0;
          ...
        case 10:
          printf("last digit is 0",iVar1);
          return 0;
      }
    } else {
      if (local_10[0] == 100) {
        printf("the last two digits are 0",iVar1);
        return 0;
  }}}
  printf("we have no rule",iVar1); return 0; }
```

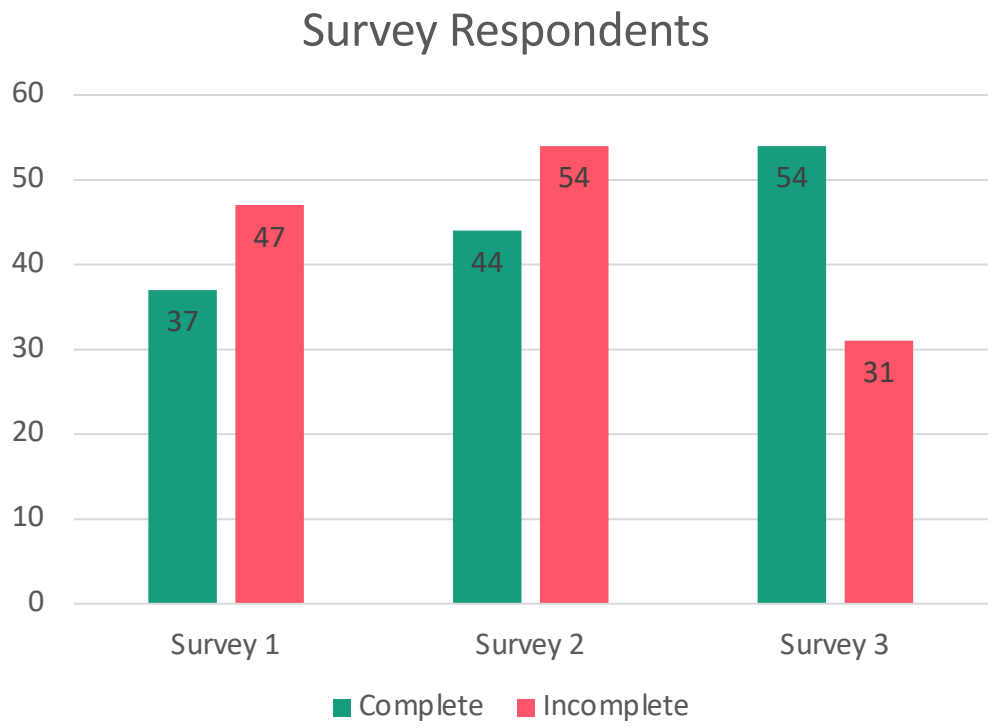
Output 2

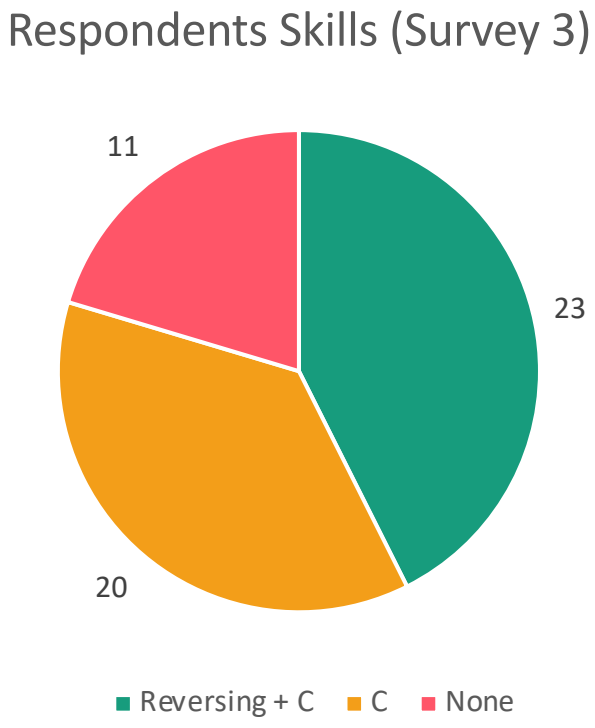
```
int dividability_rules() {
  int var_0; int * var_1;
  printf("Enter a number: ");
  var_1 = &var_0;
  __isoc99_scanf("%d", var_1);
  printf("A number is dividable by %d:\n", var_0);
  switch(var_0) {
    case 0x0:
      printf("not possible");
      break;
      ...
    case 0xa:
      printf("last digit is 0");
      break;
    case 100:
      printf("the last two digits are 0");
      break;
    case 125:
      printf("last three digits dividable by 125");
      break;
    default:
      printf("we have no rule");
  }
  return 0;
}
```

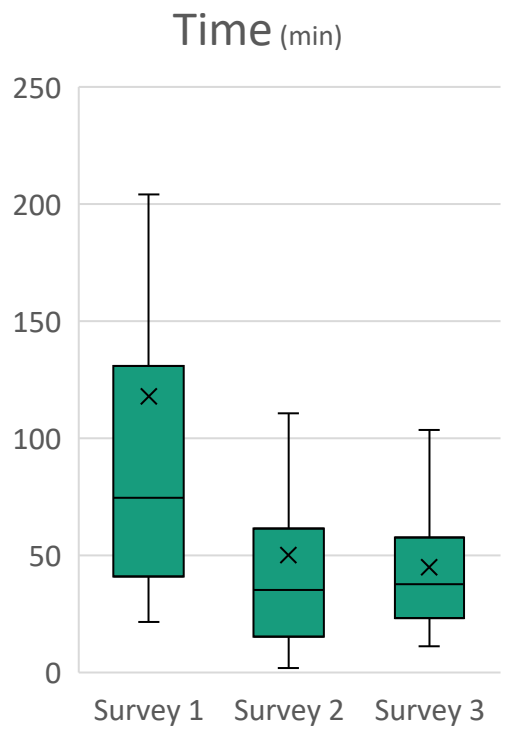
User Surveys



User Surveys Metadata







User Survey 1

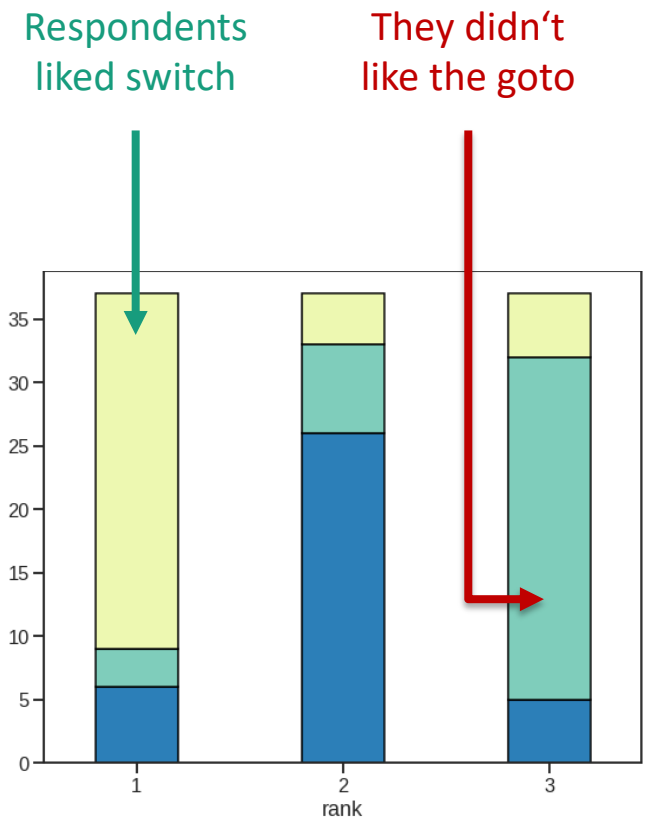
- Research Questions:
  - What decompilation aspects should be improved to enhance manual analysis?
  - What readability aspects are important to users?
  - What are the limitations of current approaches and the state-of-the-art?

Example 1

```
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;
}
```

```
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;
}
```

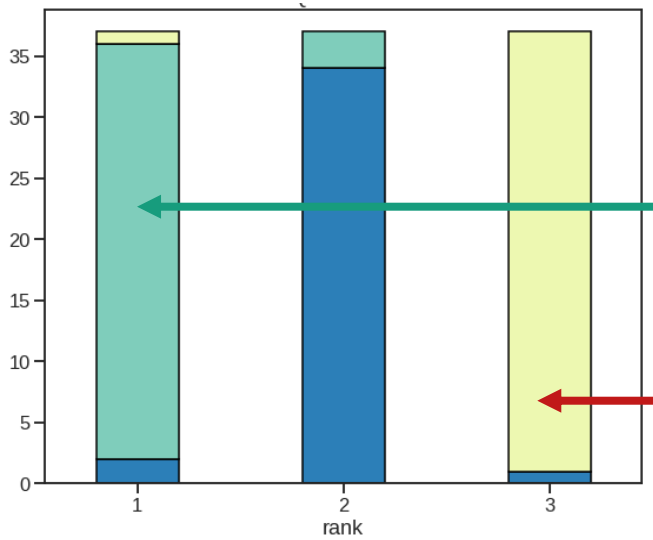
```
int test1() {
  int var_0;

  var_0 = rand();
  rand();
  if ( var_0 == 10 ) {
    goto Label_1;
  }
  if ( var_0 <= 10 ) {
    if ( var_0 == 1 ) {
      return 0;
    }
    if ( var_0 == 5 ) {
      var_0 = 6;
Label_1:
      var_0 *= 2;
      return var_0;
    }
  }
  return var_0;
}
```



Example 2

```
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;
```

```
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;
}
```



+ Short & Clean
+ For-Loop

- Long & Complex
- Too many Veriables

```
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);
}
```

Survey 1 TL;DR

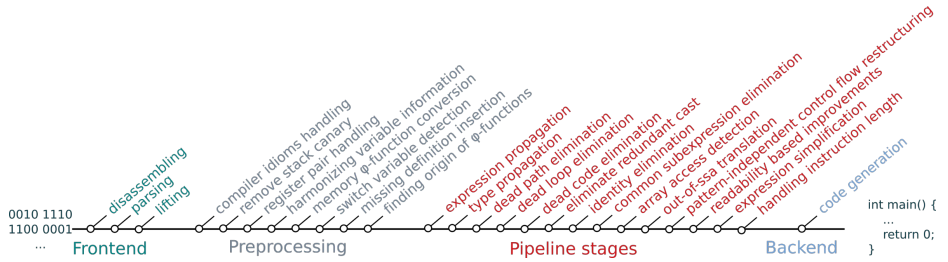
- Readability depends on user preference *and* the given analysis task
  - Users crave configurable decompilers
- Readability > Assembly Structure
- Identified many aspects with room for improvements
  - Instruction Idioms, Switch Reconstruction, …

Contributions

- Various improvements for DREAM to get academia “back on track” with commercial state-of-the-art decompilers
- **dewolf Decompiler**
  - **highly configurable, easily expandable, and open-source**
  - Available in Binary Ninja’s Plugin Manager and on dogbolt.org
- We publish all survey results [1]

[1] https://github.com/steffenenders/dewolf-surveys

dewolf Overview



Frontend

```
0010 1110
1100 0001
```

```
fibonacci:
08049186  push    ebp {__saved_ebp}
08049187  mov     ebp, esp {__saved_ebp}
08049189  sub     esp, 0x28
0804918c  mov     dword [ebp-0x10 {var_14}], 0x0
08049193  mov     dword [ebp-0x14 {var_18}], 0x1
```

```
var_14#1 = 0
var_18#1 = 1
mem#1 = printf(format: "Enter the number of terms: ") @ mem#0
var_38#1 = &var_20
mem#2 = __isoc99_scanf(format: data_804a028, var_38#1) @ mem#1
```

Lifting

```
var_14#1 = 0x0
var_18#1 = 0x1
printf("Enter the number of terms: ")
var_38#1 = &(var_20#1)
__isoc99_scanf("%d", var_38#1)
```

(Preprocessing/) Pipeline Stages

```
0.
var_14#1 = 0x0
var_18#1 = 0x1
printf("Enter the number of terms: ")
var_38#1 = &(var_20#1)
__isoc99_scanf("%d", var_38#1)
printf("Fibonacci Series: ")
var_10#1 = 0x1
```

```
1.
var_38_1#2 = φ(var_38#1,var_38_1#3)
var_1c_1#1 = φ(var_1c#0,var_1c_1#2)
var_18#2 = φ(var_18#1,var_18#3)
var_14#2 = φ(var_14#1,var_14#3)
var_10#2 = φ(var_10#1,var_10#3)
eax_4#1 = φ(eax#0,eax_4#6)
edx_1#1 = φ(edx#0,edx_1#2)
mem#4 = φ(mem#3,mem#5)
eax_5#2 = var_20#4
if(var_10#2 <= eax_5#2)
```

```
2.
var_38_1#3 = var_14#2
printf("%d, ", var_38_1#3)
edx_1#2 = var_14#2
eax_1#3 = var_18#2
eax_2#4 = eax_1#3 + edx_1#2
var_1c_1#2 = eax_2#4
eax_3#5 = var_18#2
var_14#3 = eax_3#5
eax_4#6 = var_1c_1#2
var_18#3 = eax_4#6
var_10#3 = var_10#2 + 0x1
```

```
3.
eax_6#7 = 0x0
return 0x0
```

Dataflow algorithms

Control-flow algorithms

```
0.
printf("Enter the number of terms: ")
var_1 = &(var_0)
__isoc99_scanf("%d", var_1)
printf("Fibonacci Series: ")
var_2 = 0x0
var_3 = 0x1
var_1 = 0x1
```

```
1.
if(var_3 <= var_0)
```

```
2.
printf("%d, ", var_2)
var_4 = var_1 + var_2
var_3 = var_3 + 0x1
var_2 = var_1
var_1 = var_4
```

```
3.
return 0x0
```

```
0. SeqNode
Sequence
```

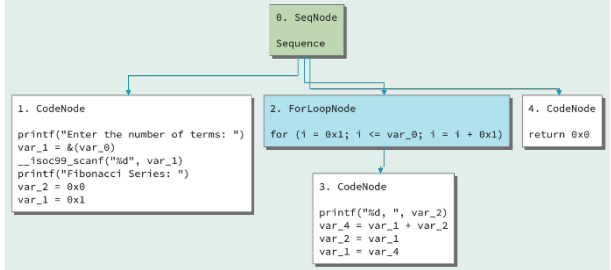
```
1. CodeNode

printf("Enter the number of terms: ")
var_1 = &(var_0)
__isoc99_scanf("%d", var_1)
printf("Fibonacci Series: ")
var_2 = 0x0
var_1 = 0x1
```

```
2. ForLoopNode

for (i = 0x1; i <= var_0; i = i + 0x1)
```

```
3. CodeNode

printf("%d, ", var_2)
var_4 = var_1 + var_2
var_2 = var_1
var_1 = var_4
```

```
4. CodeNode

return 0x0
```

Backend



Code Generator

```
int fibonacci() {
    int i;
    int var_0;
    int var_4;
    int * var_1;
    int * var_2;
    printf(/* format */ "Enter the number of terms: ");
    var_1 = &var_0;
    __isoc99_scanf(/* format */ "%d", var_1);
    printf(/* format */ "Fibonacci Series: ");
    var_2 = 0;
    var_1 = 1;
    for (i = 1; i <= var_0; i++) {
        printf(/* format */ "%d, ", var_2);
        var_4 = var_1 + var_2;
        var_2 = var_1;
        var_1 = var_4;
    }
    return 0;
}
```

dewolf Improvements over DREAM

Switch Variable Detection
For-loop Recovery
Elimination of Dead Paths and Loops
Custom Logic Engine

+

Subexpression Elimination
Instruction Idiom Handling
Elimination of Redundant Casts
Improved Out-of-SSA

+

Constant Representation
Array Access Detection
Continue in Loops

=

```
int64_t xor(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4) {
  printf("%lu",arg2);
  int32_t var_c = 0x0L;
  while(true) {
    int64_t rax_14 = var_c /* sx None */;
    if ((arg2 <= var_c /* sx None */)) {
      break;
    }
    char* rax_4 = (arg1 + var_c /* sx None */);
    uint64_t rsi_1 = *rax_4 /* zx char */ /* zx uint32_t */;
    int64_t rax_6 = var_c /* sx None */;
    int64_t rdx_1 = 0x0L;
    char* rax_9 = (arg3 + ((rdx_1:rax_6) % arg4));
    uint64_t rcx = *rax_9 /* zx char */ /* zx uint32_t */;
    int64_t rax_12 = (arg1 + var_c /* sx None */);
    uint64_t rdx_4 = (rsi_1.esi ^ rcx.ecx) /* zx None */;
    *rax_12 = rdx_4.dl;
    int32_t var_c = (var_c + 0x1L);
  }
  return rax_14;
}
```

vs.

```
long xor(void * arg1, long arg2, long arg3, long arg4) {
  int i;
  int var_1;
  void * var_0;
  printf(/* format */ "%lu\n", arg2);
  for (i = 0; arg2 > i; i++) {
    var_0 = arg1 + i;
    *var_0 = *var_0 ^ *(arg3 + (i + (0L << 0x40)) % arg4);
  }
  var_1 = i;
  return var_1;
}
```

User Survey 2+3 Research Questions

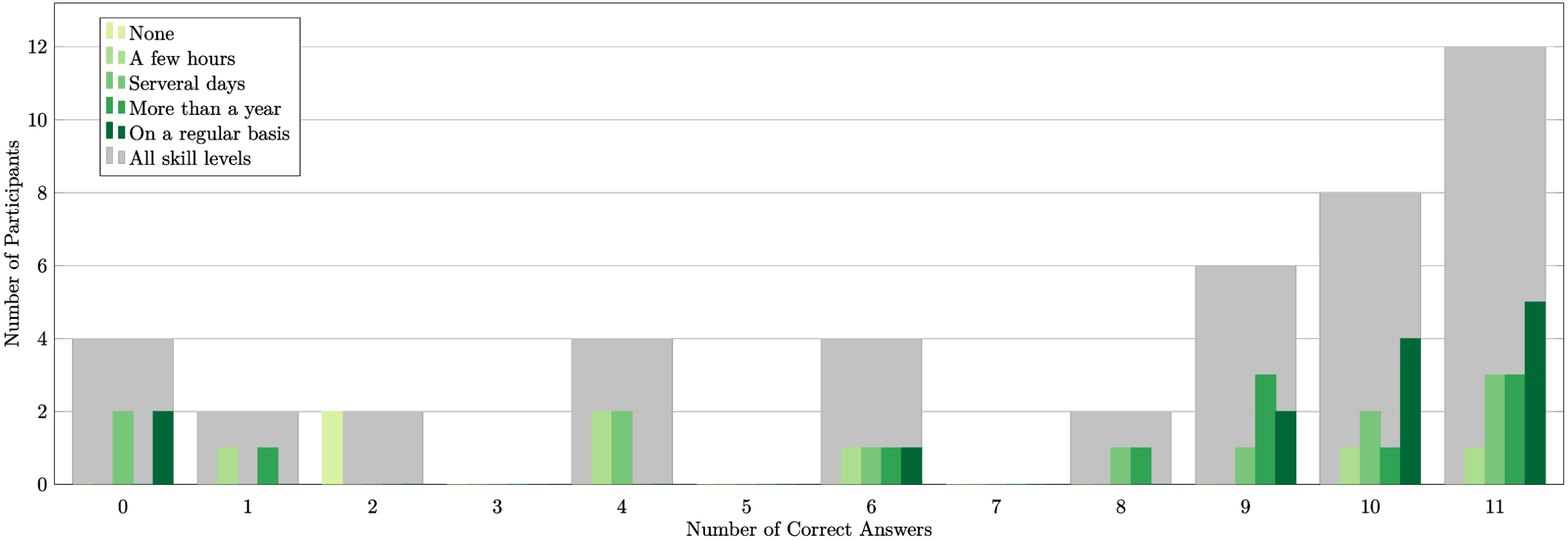
- Research Questions:
  - Does dewolf produce readable & comprehensible output to qualify as a base for future research?
  - (Can dewolf exceed state-of-the-art decompilers in certain cases?)
  - To what extent do respondents tolerate diverging from the assembly during decompilation?
  - What are further areas to improve readability in future research?

Comprehension Questions

- What is the purpose of the function?
- What TLDs does the DGA utilize?
- What is the most used TLD?
- Which of the provided domains could be generated by the DGA?
- How many different domains can be generated by the DGA?

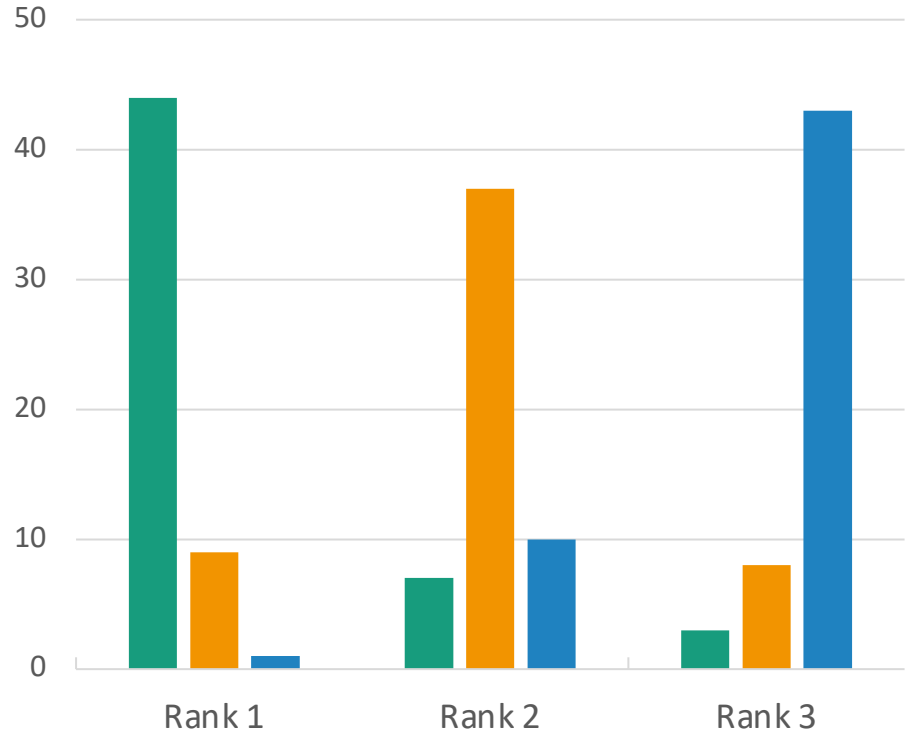
```
int sub_401000() {
    char i;
    char * var_0;
    char var_1;
    char var_3;
    int var_4;
    var_0 = malloc(16);
    var_1 = GetTickCount();
    GetSystemTime(lpSystemTime: var_0);
    for (i = 0; i < 8; i++) {
        var_3 = var_0[i];
        var_0[i] = ((unsigned int)(var_3 ^ var_1) % 24 & 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;
}
```

Comprehension Results Divided by C-Skills (self assessment)



Decompiler Comparison

```
int convert_binary_to_hex()
{
    long long binary;
    char hex[65] = "";
    int rem;
    printf("Enter any binary number: ");
    scanf("%lld", &binary);
    int original_binary = binary;
    while(binary > 0)
    {
        rem = binary % 10000;
        switch(rem)
        {
            case 0:
                strcat(hex, "0");
                break;
            case 1:
                strcat(hex, "1");
                break;
            ...
            case 1111:
                strcat(hex, "F");
            break;
        }
        binary /= 10000;
    }
    printf("Binary number: %lld\n", original_binary);
    printf("Hexadecimal number: %s", hex);
    return 0;
}
```



What participants (dis)liked in state-of-the-art decompilers

Positive		Negative
Switch-Recovery		Deeply Nested If/Else
Low Nesting Depth		High Nesting Depth
for-loops, if possible		While-loops instead of for-loops
Good Type Recovery	*vs.*	Bad Type Recovery
Less Casts		Explicit Casts
Array Access Detection		Too many unnecessary variables
Constant Annotations		String Parameters

Introducing Copies during Restructuring?

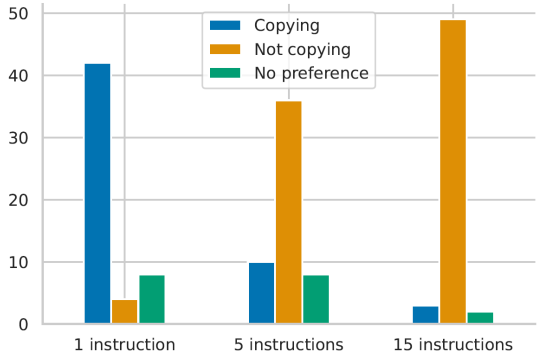
```
return var;
```

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

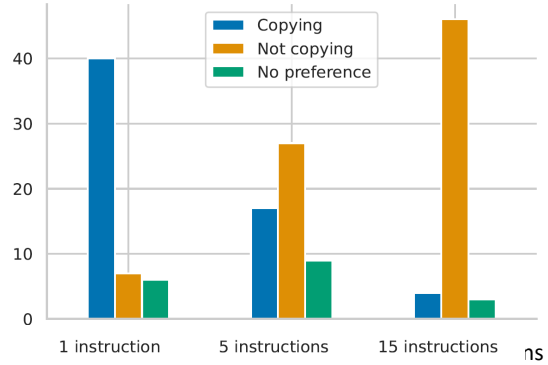
```
y = x + 5;
z = bar(y);
printf("Enter a number larger than %d: \\n", z);
scanf("%d", &numb1);
printf("Enter a number smaller than %d: \\n", z);
scanf("%d", &numb2);
y += z;
printf("Enter a number larger than %d: \\n", y);
scanf("%d", &numb3);
printf("Enter a number smaller than %d: \\n", y);
scanf("%d", &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;
```

?

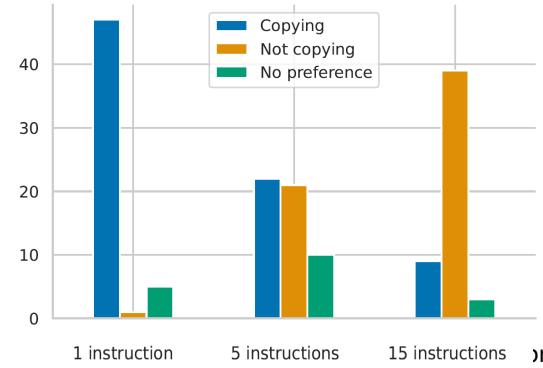
If-Else



Multi-Exit



Multi-Entry



TL;DR

- Future decompilers (focusing on manual analysis) should:
  - Favor readability over sticking to the assembly
  - Be highly configurable!

- If you are into decompilation, you should:
  - Check out dewolf (great base for new research, imho)
  - Read the paper and survey results

Q&A

steffen.enders@fkie.fraunhofer.de

@steffenenders_

https://github.com/fkie-cad/dewolf