DIRE:
A Neural Approach to Decompiled Identifier Renaming

Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, Bogdan Vasilescu

Carnegie Mellon University
The Real Story of Stuxnet

How Kaspersky Lab tracked down the malware that stymied Iran’s nuclear-fuel enrichment program

By David Kushner

Computer cables snake across the floor. Cryptic flowcharts are scrawled across various whiteboards adorning the walls. A life-size Batman doll stands in the hall. This office might seem no different than any other geeky workplace, but in fact it’s the front line of a war—a cyberwar, where most battles play out not in remote jungles or deserts but in
Disassembler

```
40299c:  89 f0    mov    %esi,%eax
40299e:  83 e0 04  and    $0x4,%eax
4029a1:  74 1d    je      4029c0 <main+0x8b0>
4029a3:  31 d2    xor    %edx,%edx
4029a5:  48 89 d8  mov    %rbx,%rax
4029a8:  48 f7 f7  div    %rdi
4029ab:  48 89 05 be b8 20 00 mov    %rax,0x20b8be(%rip)
4029b2:  48 89 15 ff ba 20 00 mov    %rdx,0x20baff(%rip)
4029b9:  4d 85 c0  test   %r8,%r8
4029bc:  75 14    jne    4029d2 <main+0x8c2>
4029be:  eb 31    jmp    4029f1 <main+0x8e1>
4029c0:  48 83 fb ff cmp    %0xffffffffffffffff,%rbx
4029c4:  74 07    je      4029cd <main+0x8bd>
4029c6:  48 89 1d a3 b8 20 00 mov    %rbx,0x20b8a3(%rip)
4029cd:  4d 85 c0  test   %r8,%r8
4029d0:  74 1f    je      4029f1 <main+0x8e1>
4029d2:  89 c8    mov    %ecx,%eax
4029d4:  83 e0 10  and    $0x10,%eax
4029d7:  74 18    je      4029f1 <main+0x8e1>
```
Disassembler
Decompiler
Decompiler

```c
usage(1);

v8 = v7 + 1;

switch (__ROR1__(*v6 - 99, 1))
{
    case 0:
        if (v6[1] != 111)
            goto LABEL_46;
        if (v6[2] != 110)
            goto LABEL_46;
        if (v6[3] != 118)
            goto LABEL_46;
        v9 = v6[4];
        if (v9)
            {
                if (v9 != 61)
                    goto LABEL_46;
            }
        conversions_mask |= parse_symbols(v8, conversions, 0, "invalid conversions, 0, "invalid
        goto LABEL_90;
    case 3:
        if (v6[1] != 102)
            goto LABEL_46;
        v12 = v6[2];
        if (v12 && v12 != 61)
            {
                    {
                        v13 = v6[5];
                        if (!v13 || v13 == 61)
                            {
```
The problem:

Decompilers are typically unable to assign meaningful names to variables
void *file_mmap(int V1, int V2)
{
    void *V3;
    V3 = mmap(0, V2, 1, 2, V1, 0);
    if (V3 == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return V3;
}

void *file_mmap(int fd, int size)
{
    void *ret;
    ret = mmap(0, size, 1, 2, fd, 0);
    if (ret == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return ret;
}
void *file_mmap(int V1, int V2)  
{  
  void *V3;  
  V3 = mmap(0, V2, 1, 2, V1, 0);  
  if (V3 == (void *) -1) {  
    perror("mmap");  
    exit(1);  
  }  
  return V3;  
}

void *file_mmap(int fd, int size)  
{  
  void *ret;  
  ret = mmap(0, size, 1, 2, fd, 0);  
  if (ret == (void *) -1) {  
    perror("mmap");  
    exit(1);  
  }  
  return ret;  
}
void *file_mmap(int V1, int V2)
{
    void *V3;
    V3 = mmap(0, V2, 1, 2, V1, 0);
    if (V3 == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return V3;
}

void *file_mmap(int fd, int size)
{
    void *ret;
    ret = mmap(0, size, 1, 2, fd, 0);
    if (ret == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return ret;
}
Today

Decompiler output

void *file_mmap(int V1, int V2)
{
    void *V3;
    V3 = mmap(0, V2, 1, 2, V1, 0);
    if (V3 == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return V3;
}

Refactored decompiler output

void *file_mmap(int fd, int size)
{
    void *ret;
    ret = mmap(0, size, 1, 2, fd, 0);
    if (ret == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return ret;
}
up to 74% recovery of original source code names on an open-source GitHub corpus
Why does it work?
Key principle: Software is "natural"
(2012 International Conference on Software Engineering)

On the Naturalness of Software

Abram Hindle, Earl Barr, Zhendong Su
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
{ajhindle,barr,su}@cs.ucdavis.edu

Mark Gabel
Dept. of Computer Science
The University of Texas at Dallas
Richardson, TX 75080 USA
mark.gabel@utdallas.edu

Prem Devanbu
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
devanbu@cs.ucdavis.edu

Abstract—Natural languages like English are rich, complex, and powerful. The highly creative and graceful use of languages like English and Tamil, by masters like Shakespeare and Avvaiyar, can certainly delight and inspire. But in practice, given cognitive constraints and the exigencies of daily life, most human utterances are far simpler and much more repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension.

We begin with the conjecture that most software is also very repetitive, and in fact even more so than natural languages. As an example use of the model, evidence supportive of a positive answer to both these questions.

Using the widely adopted n-gram model, we provide empirical evidence of the fact that code, despite being written in an artificial language, is also likely to be repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension.

We believe that this is a general, useful and practical notion that, despite its simplicity, already improves Eclipse's built-in code completion capability. We conclude the paper by laying out a vision for future research in this area.

We argue that code is also very repetitive, and in fact even more so than natural languages. As an example use of the model, evidence supportive of a positive answer to both these questions.

Using the widely adopted n-gram model, we provide empirical evidence of the fact that code, despite being written in an artificial language, is also likely to be repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension.

We believe that this is a general, useful and practical notion that, despite its simplicity, already improves Eclipse's built-in code completion capability. We conclude the paper by laying out a vision for future research in this area.
void *file_mmap(int V1, int V2)
{
    void *V3;
    V3 = mmap(0, V2, 1, 2, V1, 0);
    if (V3 == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return V3;
}

void *file_mmap(int fd, int size)
{
    void *ret;
    ret = mmap(0, size, 1, 2, fd, 0);
    if (ret == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return ret;
}
Idea:
Learn typical variable names in a given context from examples ... many many examples

If software is repetitive, so are names

```c
int main(int ?)
```
Idea:
Learn typical variable names in a given context from examples ... many many examples

If software is repetitive, so are names
int main(int_banana
Idea:
Learn typical variable names in a given context from examples ... many many examples

If software is repetitive, so are names

```c
int main(int argc
```
Good news:
We can generate arbitrarily many examples
Corpus Construction

Original Source

```c
#include <stdio.h>

int main() {
    int x = 0;
    int y = 0;
    while (x < 100) {
        printf("%d\n", x);
        x++;
    }
    return y;
}
```

Decompiled Code

```c
#include <stdio.h>

int main() {
    int v1 = 0;
    int v2 = 0;
    while (v1 < 100) {
        printf("%d\n", v1);
        v1++;
    }
    return v2;
}
```
Corpus Construction

Original Source

```c
#include <stdio.h>

int main() {
    int x = 0;
    int y = 0;
    while (x < 100) {
        printf("%d\n", x);
        x++;
    }
    return y;
}
```

Decompiled Code

```c
#include <stdio.h>

int main() {
    int v1 = 0;
    int v2 = 0;
    while (v1 < 100) {
        printf("%d\n", v1);
        v1++;
    }
    return v2;
}
```
Difficulty: Decompilation Changes Structure

Original Source

```c
#include <stdio.h>

int main() {
    int x = 0;
    int y = 0;
    while (x < 100) {
        printf("%d\n", x);
        x++;
    }
    return y;
}
```

Decompiled Code

```c
int __cdecl main(int argc, const char **argv)
{
    signed int i; // [rsp+8h] [rbp-8h]
    for (i = 0; i < 100; ++i )
        printf("%d\n", (unsigned int)i, envp);
    return 0;
}
```
Difficulty: Decompilation Changes Structure

Original Source

```
#include <stdio.h>

int main() {
    int y = 0;
    while (x < 100) {
        printf("%d\n", x);
        x++;
    }
    return y;
}
```

Decompiled Code

```
int __cdecl main(int argc, 
signed int i; // [rsp+8h] [rbp-8h]
    for ( i = 0; i < 100; ++i )
        printf("%d\n", (unsigned int)i, envp);
    return 0;
```

Different function signatures
Difficulty: Decompilation Changes Structure

Original Source

```c
#include <stdio.h>

int x = 0;
int y = 0;

int main() {
    printf("%d\n", x);
    x++;
}
return y;
```

Decompiled Code

```c
int cdecl main(int argc, const char **argv) {
    for (i = 0; i < 100; ++i) {
        printf("%d\n", (unsigned int)i, envp);
    }
    return 0;
}
```

Different numbers of variables
Difficulty: Decompilation Changes Structure

Different types of loops

Original Source

```c
#include <stdio.h>

int main() {
  int x = 0;
  while (x < 100) {
    printf("%d\n", x);
    x++;
  }
  return 0;
}
```

Decompiled Code

```c
int __cdecl main(int argc, const char **argv)
{
  signed int i; // [rsp+8h] [rbp-8h]
  for ( i = 0; i < 100; ++i )
  {
    printf("%d\n", (unsigned int)i); nVp);
    return 0;
  }
```
Difficulty: Decompilation Changes Structure

Original Source

```c
#include <stdio.h>

int main() {
    int x = 0;
    int y = 0;
    while (x < 100) {
        printf("%d\n", x);
        x++;
    }
    return y;
}
```

Decompiled Code

```c
int __cdecl main(int argc, const char **argv) {
    signed int i; // [rsp+8h] [rbp-8h]
    for (i = 0; i < 100; ++i )
        printf("%d\n", (unsigned int)i, envp);
    return 0;
}
```
Two different loops.

```c
int i, z;
for (i = 0; i < 10; i++) {
    z += i;
}

int v1, v2;
v1 = 0;
while (v1 < 10) {
    v2 += v1;
    v1++;
}
```
Two different loops.

```c
int i, z;
for (i = 0; i < 10; i++) {
    z += i;
}

int v1, v2;
v1 = 0;
while (v1 < 10) {
    v2 += v1;
    v1++;
}
```

Same assembly code.

```assembly
var1 = dword ptr -8
var2 = dword ptr -4

mov [rbp+var2], 0
jmp loc_4a5

loc_49B:
mov eax, [rbp+var2]
add [rbp+var1], eax
add [rbp+var2], 1

loc_4a5:
cmp [rbp+var2], 9
jle loc_49b
```

Alignment
Key insight: Operations on variables and their offsets are the same
Alignment

Key insight: Operations on variables and their offsets are the same.
Learning from examples

Binary → Decompiler → DIRE → Meaningful Variable Names

Decompiled Identifier Renaming Engine
Recall:
Names are repetitive in a given context

int main(int argc
Running example

```c
char* mystrcopy(char *VAR1, char *VAR2){
    char *result;
    if (VAR1 && VAR2)
        result = strncopy(VAR1, VAR2);
    else
        result = 0LL;
    return result;
}
```
Code can be a sequence of lexical tokens ...

... or a syntax tree

char * mystrcopy
(char * VAR1, ...
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

Structural Encoder (GGNN)

Decoder

Sequential Decoder with Attention
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

Structural Encoder (GGNN)
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

char * my _str _copy ( char * VAR1

h1 h2 h3 h4 h5 h6 h7 h8 h9 ...

Structural Encoder (GGNN)
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

char * my _str _copy ( char * VAR1

Look behind and ahead for more context

Structural Encoder (GGNN)
**DIRE Neural Architecture**

Encoder

Lexical Encoder (LSTM)

Sub-tokenization (reduces vocabulary & training time)

Structural Encoder (GGNN)
Encoder

Lexical Encoder (LSTM)

Structural Encoder (GGNN)
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

Structural Encoder (GGNN)

block
name: mystrcopy

If

land

asg

call

result

VAR2

VAR1

strcopy

VAR1

VAR2

result

0LL

VAR1
type: char *
"Super node" — different uses of the same variable
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

Structural Encoder (GGNN)

block
name: mystrcopy

If

land

asg

call

result

strncpy

VAR1

type: char *

VAR2

result

0LL

Link function name to arguments

VAR1

VAR1

VAR1

VAR2

VAR1

VAR2
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

Structural Encoder (GGNN)
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

VAR1
VAR2

VAR1
VAR2

VAR1
VAR2

Structural Encoder (GGNN)

Decoder

Sequential Decoder with Attention
DIRE Neural Architecture

Encoder

- Lexical Encoder (LSTM)
- Identifier Representations
- Structural Encoder (GGNN)

Decoder

- Sequential Decoder with Attention
DIRE Neural Architecture

Encoder
- Lexical Encoder (LSTM)
- Structural Encoder (GGNN)

Identifier Representations

Decoder
- Sequential Decoder with Attention
DIRE Neural Architecture

Encoder
- Lexical Encoder (LSTM)
- Structural Encoder (GGNN)
- Code Element Representations

Decoder
- Identifier Representations
- Sequential Decoder with Attention
DIRE Neural Architecture

Encoder
- Lexical Encoder (LSTM)
- Structural Encoder (GGNN)

Decoder
- Sequential Decoder with Attention

Code Element Representations
Identifier Representations
DIRE Neural Architecture

Decoder

Code Element Representations

Attention

Identifier Representations

VAR1

VAR1

VAR1

VAR2
DIRE Neural Architecture

Encoder

Lexical Encoder (LSTM)

Structural Encoder (GGNN)

Decoder

Code Element Representations

Sequential Decoder with Attention

Identifier Representations
How good are the renamings?
Assumption:
Original (human-written) names are good

How many can we recover?
Dataset

• 164,632 unique x86-64 binaries
• 1,259,935 decompiled functions
• Split by binary into test/training/validation
• Open dataset, link in paper/on ASE site
## Variable Recovery Rate (%)

<table>
<thead>
<tr>
<th></th>
<th>Lexical</th>
<th>Structural</th>
<th>Prior Work*</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRE</td>
<td>74.3</td>
<td>72.9</td>
<td>64.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16.2</td>
</tr>
</tbody>
</table>

# Variable Recovery Rate (%)

<table>
<thead>
<tr>
<th></th>
<th>DIRE</th>
<th>Lexical</th>
<th>Structural</th>
<th>Prior Work*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>74.3</td>
<td>72.9</td>
<td>64.6</td>
<td>16.2</td>
</tr>
</tbody>
</table>

## Variable Recovery Rate (%)

<table>
<thead>
<tr>
<th></th>
<th>DIRE</th>
<th>Lexical</th>
<th>Structural</th>
<th>Prior Work*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>74.3</td>
<td>72.9</td>
<td>64.6</td>
<td>16.2</td>
</tr>
</tbody>
</table>

```c
file *f_open(char **V1, char *V2, int V3) {
    int fd;
    if (!V3)
        return fopen(*V1, V2);
    if (*V2 != 119)
        assert_fail("fopen");
    fd = open(*V1, 577, 384);
    if (fd >= 0)
        return reopen(fd, V2);
    else
        return 0;
}
```

<table>
<thead>
<tr>
<th>Developer</th>
<th>Lexical</th>
<th>Structural</th>
<th>DIRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>filename</td>
<td>file</td>
<td>fname</td>
</tr>
<tr>
<td>V2</td>
<td>mode</td>
<td>name</td>
<td>oname</td>
</tr>
<tr>
<td>V3</td>
<td>is_private</td>
<td>mode</td>
<td>flags</td>
</tr>
</tbody>
</table>
Decompiler output

```
void *file_mmap(int V1, int V2)
{
    void *V3;
    V3 = mmap(0, V2, 1, 2, V1, 0);
    if (V3 == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return V3;
}
```

Refactored decompiler output

```
void *file_mmap(int fd, int size)
{
    void *ret;
    ret = mmap(0, size, 1, 2, fd, 0);
    if (ret == (void *) -1) {
        perror("mmap");
        exit(1);
    }
    return ret;
}
```

Jeremy Lacomis

✉️ jlacomis@cmu.edu

✈️ @jlacomis

🌐 jeremylacomis.com
BONUS SLIDES
NUMERICAL RECIPES in C
The Art of Scientific Computing
Second Edition
William H. Press
Saul A. Teukolsky
William T. Vetterling
Brian P. Flannery
will always converge, provided that the initial guess is good enough. Indeed, one can even determine in advance the rate of convergence of most algorithms.

It cannot be overemphasized, however, how crucially success depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numbers. Carefully crafted initial estimates reward you not only with reduced computational effort, but also with understanding and increased self-esteem. Hamming’s motto, “The purpose of computing is insight, not numbers,” is particularly apt in the area of finding roots. You should repeat this motto aloud whenever your program converges, with analytical accuracy, to the wrong root of a problem, or whenever it fails to converge—because there is actually no root, or because your initial estimate was not sufficiently close to it.

“The lack of insight is all very well, but what do I actually do?” For one-dimensional root finding, it is possible to give some straightforward answers: You should try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like.

Next, you should always bracket a root—that is, know that the function changes sign in an identified interval, before trying to converge to the root’s value.

Finally, if this is advice which some during soils might disagree, but we give it nonetheless—never let your iteration method get outside of the bounds of the root finding branch calculated at any stage. We will see below that some pedagogically important algorithms, such as secant method or Newton-Raphson, can violate this last constraint, and are thus not recommended unless certain steps are implemented.

Multiple roots, or very close roots, are a real problem, especially if the multiplicity is as great as one. In such cases, there may be no readily apparent sign change in the function, so the notion of bracketing a root — and maintaining the bracket — becomes difficult. We are hard-line: we nevertheless insist on bracketing a root, even if it takes the minimum-searching techniques of Chapter 9 to determine whether a neutral point in the function really does cross zero or not. (You can easily modify the simple golden section routine of §9.1 to return early if it detects a sign change in the function. And, if the minimum of the function is exactly zero, then you have found a double root.)

As usual, we want to discourage you from using routines as black boxes without understanding them. However, as a guide to beginners, here are some reasonable starting points:

- Brent’s algorithm (§9.3) is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function’s derivative. Ridders’ method (§9.2) is concise, and a close competitor.

- When you can compute the function’s derivative, the routine rtsafe in §9.4, which combines the Newton-Raphson method with some bookkeeping in bounds, is recommended. Again, you must find bracket your root.

- Roots of polynomials are a special case. Laguerre’s method, in §9.5, is recommended as a starting point. Beware! Some polynomials are ill-conditioned.

- Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), which works very well if you can supply a

---

```
 [9.0] Introduction

Number of horizontal and vertical positions in display

for (x=0; x<=1000; x++)
for (y=0; y<=1000; y++)
  if ((x+y) % 2 == 0)
    printf("%d,%d\n", x, y);
  else
    printf("%d,%d\n", y, x);
```

---

good first guess of the solution. Try it. Then read the more advanced material in §9.7 for some more complicated, but globally more convergent, alternatives.

Avoiding implementations for specific computers, this book must generally appeal more clearly to interactive or graphics-related routines. We make an exception right now. The following routine, which produces a crude function plot with interactively select points, can save you a lot of grief as you enter the world of root finding.

```
for (x=0; x<=1000; x++)
for (y=0; y<=1000; y++)
  if ((x+y) % 2 == 0)
    printf("%d,%d\n", x, y);
  else
    printf("%d,%d\n", y, x);
```
will always converge, provided that the initial guess is good enough. Indeed, one can even determine in advance the rate of convergence of most algorithms. It cannot be overemphasized, however, how crucially success depends on having a good first guess for the solution, especially for multidimensional problems. This crucial beginning usually depends on analysis rather than numerics. Carefully crafted initial estimates reward you not only with reduced computational effort, but also with understanding and increased self-esteem. Hammering’s motto, “the process of computing is insight, not numbers,” is particularly apt in the area of finding roots. You should always rephrase this motto whenever your program converges, with an eye to the wrong root of a problem, or whenever it fails to converge because there is actually no root, or because your initial estimate was insufficiently close to it.

“Talk of insight is all very well, but what do I actually do?” For multidimensional root finding, it is possible to give some straightforward answers. You should try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like. Next, you should always bracket a root, that is, know that the function changes sign in an identified interval, before trying to converge to the root’s value.

Finally, this is advice worth which some during soups might disagree, but we give it nonetheless: never let your iteration method get outside of the bracketing bands obtained at any stage. We will see below that some pedagogically important algorithms, such as Steffensen’s method of Newton-Raphson, can violate this last constraint, and are thus not recommended unless certain tricks are implemented.

Multiple roots, or very close roots, are a real problem, especially if the multiplicity is an even number. In that case, there may be no readily apparent sign change in the function, so the notion of bracketing a root — and maintaining the bracket — becomes difficult. We are hard-core: we nevertheless insist on bracketing a root, even if it takes the minimum-searching techniques of Chapter 8. In this case to determine whether a numerical dip in the function really does cross zero or not. (You can easily modify the standard golden section routine of §9.1 to return early if it detects a sign change in the function. And, if the minimum of the function is exactly zero, then you have found a double root.)

As usual, we want to discourage you from using routines as black boxes without understanding them. However, as a guide to beginners, here are some reasonable starting points:

Brent’s algorithm is §9.3. The method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the function’s derivative. Ridders’ method (§9.2) is concise, and a close competitor.

When you can compute the function’s derivative, the routine rtsafe of §9.4, which combines the Newton-Raphson method with some bookkeeping on bounds, is recommended. Again, you must first bracket your root.

Roots of polynomials are a special case. Laguerre’s method, in §9.5, is recommended as a starting point. Beware: Some polynomials are ill-conditioned.

Finally, for multidimensional problems, the only elementary method is Newton-Raphson (§9.6), which works very well if you can supply a

```
# Example of a simple function for Newton-Raphson method.

def f(x):
    return x**2 - 3

# Initial guess for the root
x0 = 1.0

# Newton-Raphson method
for i in range(10):
    x1 = x0 - f(x0) / f'(x0)
    if abs(x1 - x0) < 1e-6:
        print(f'Root found: x = {x1}')
        break
    x0 = x1
```

In our example, we perform 10 iterations and stop when the relative difference between the current and previous approximations is less than 1e-6.
int jz, j, i;
float ysm1, ybig, x2, x1, x, dyj, dx, y[ISCR+1];
char scr[ISCR+1][JSCR+1];
Preliminary Human Study

Presented users with short snippets (<50 lines) of decompiled code, asked to perform various maintenance tasks, graded and timed:

```
1 int x = 1;
2 int y = 0;
3 while (x <= 5) {
4    y += 2;
5    x += 1;
6 }
7 printf("%d", y);
```

What is the value of the variable y on line 7?
The amount of training data matters

![Graph showing the relationship between the size of the training set and accuracy. As the size of the training set increases, accuracy also increases.]
The uniqueness of the functions matters

![Graph showing the relationship between size of training set and accuracy. The x-axis represents the size of the training set, ranging from 1% to 100%. The y-axis represents accuracy, ranging from 0% to 100%. The graph includes three lines: Overall, Body in Train, and Body not in Train. The Overall line shows a steady increase in accuracy as the training set size increases. The Body in Train line is slightly lower than the Overall line, and the Body not in Train line is the lowest.](image-url)