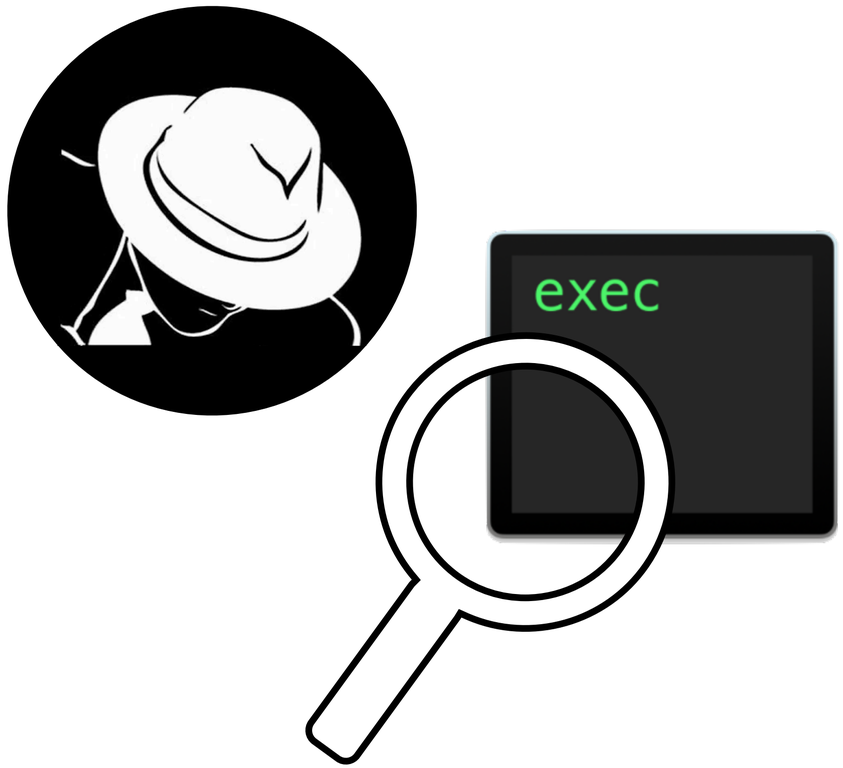
Idioms: A Simple and Effective Framework for Turbo-Charging Local Neural Decompilation with Well-Defined Types

Luke Dramko[1], Claire Le Goues[1], Edward J Schwartz[2]

1 

2 

Problem: Security practitioners need to analyze executable programs where the source code is unavailable.



Problem: Security practitioners need to analyze executable programs where the source code is unavailable.

Useful for

Problem: Security practitioners need to analyze executable programs where the source code is unavailable.

Useful for

- Malware Analysis

Problem: Security practitioners need to analyze executable programs where the source code is unavailable.

Useful for

- Malware Analysis
- Vulnerability Research

Problem: Security practitioners need to analyze executable programs where the source code is unavailable.

Useful for

- Malware Analysis
- Vulnerability Research
- Patching Legacy Software

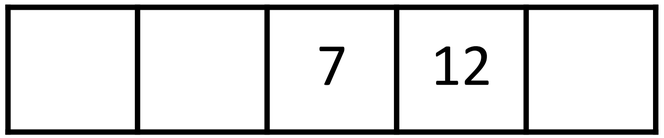
Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

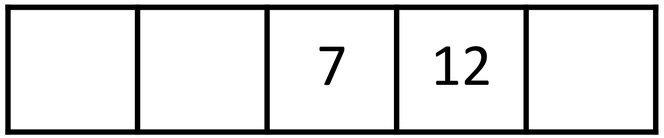
```
hash_find_index(struct hash *h, 17)
```



Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

```
hash_find_index(struct hash *h, 17)
```

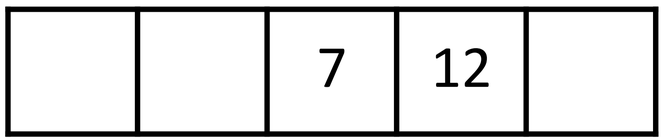


Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

```
hash_find_index(struct hash *h, 17)

    hash_make_key(h, 17);
```

2

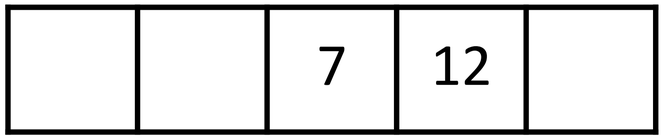


Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

```
hash_find_index(struct hash *h, 17)
```

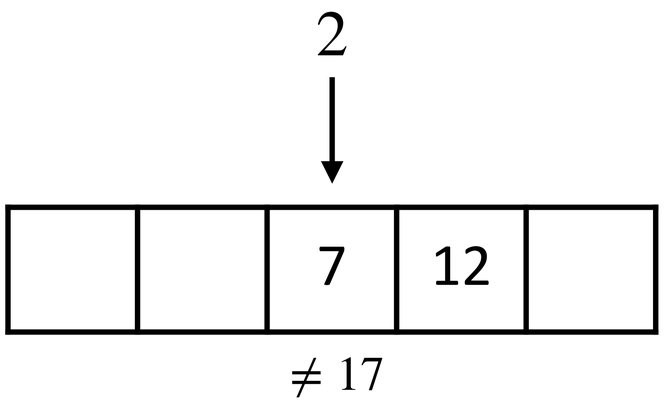
2



Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

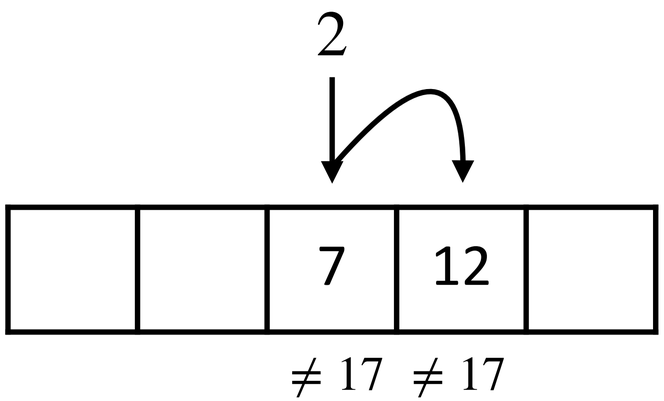
```
hash_find_index(struct hash *h, 17)
```



Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

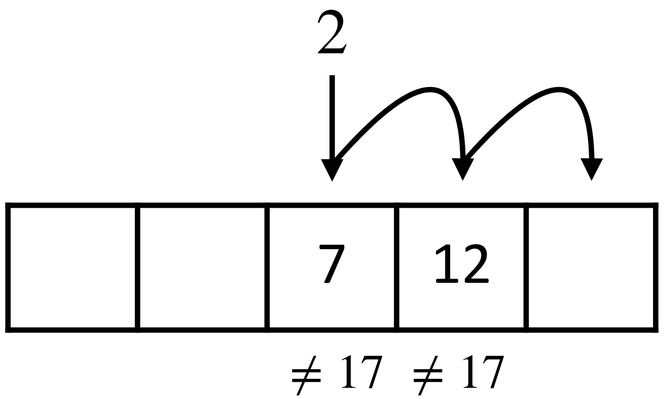
```
hash_find_index(struct hash *h, 17)
```



Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

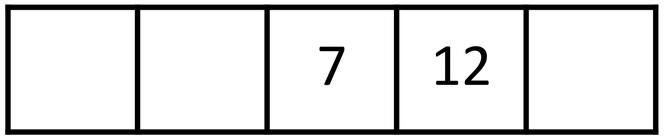
```
hash_find_index(struct hash *h, 17)
```



Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

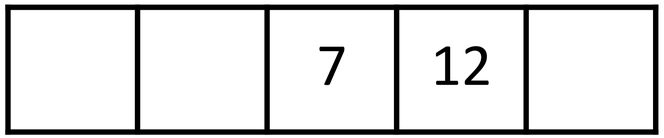
```
hash_find_index(struct hash *h, 17)
```



Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

```
hash_find_index(struct hash *h, 17)
```



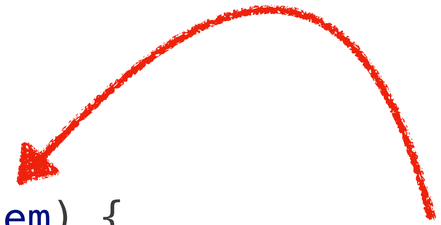

```
return -1;
```

Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

Motivational Example: Hash Table with Linear Probing

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



In practice, these are structs that represent network connections.

Compilers convert code from a higher level language into a lower-level language.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

Compilers convert code from a higher level language into a lower-level language.

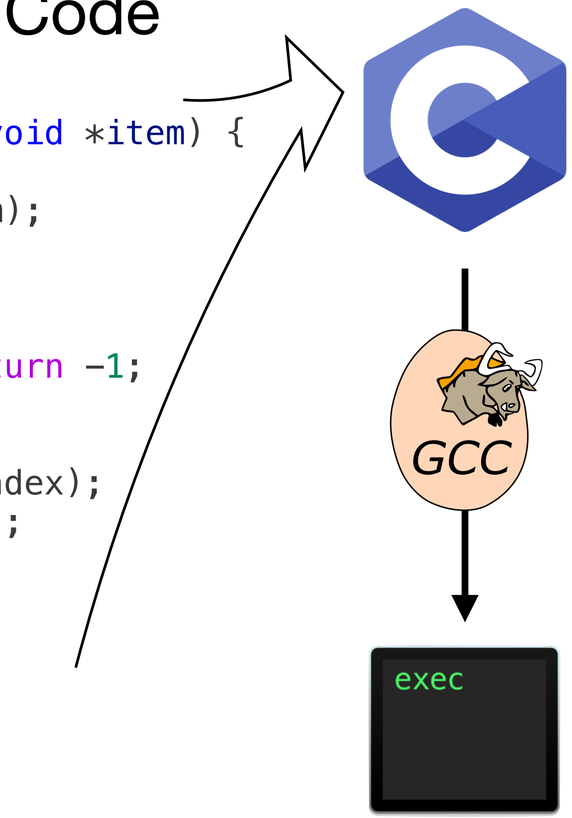
Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

Compilers convert code from a higher level language into a lower-level language.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



Assembly

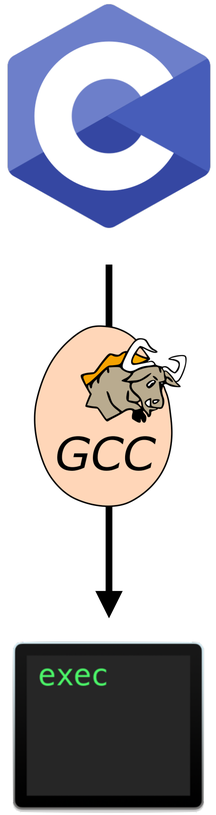
(Human-Readable Compiled)

```
155:   push   %rbp
156:   mov    %rsp,%rbp
159:   sub    $0x20,%rsp
15d:   mov    %rdi,-0x18(%rbp)
161:   mov    %rsi,-0x20(%rbp)
165:   mov    -0x20(%rbp),%rdx
169:   mov    -0x18(%rbp),%rax
16d:   mov    %rdx,%rsi
170:   mov    %rax,%rdi
173:   callq  0x94
178:   mov    %eax,-0xc(%rbp)
17b:   movl   $0x0,-0x10(%rbp)
182:   mov    -0x18(%rbp),%rax
186:   mov    0x8(%rax),%rax
18a:   mov    -0xc(%rbp),%edx
18d:   mov    %edx,%esi
18f:   mov    %rax,%rdi
192:   callq  0x5e
197:   mov    %rax,-0x8(%rbp)
19b:   jmp    0x200
19d:   mov    -0x10(%rbp),%eax
1a0:   lea    0x1(%rax),%edx
1a3:   mov    %edx,-0x10(%rbp)
```

Compilation is *lossy*: some abstractions present in source code are discarded

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



Assembly

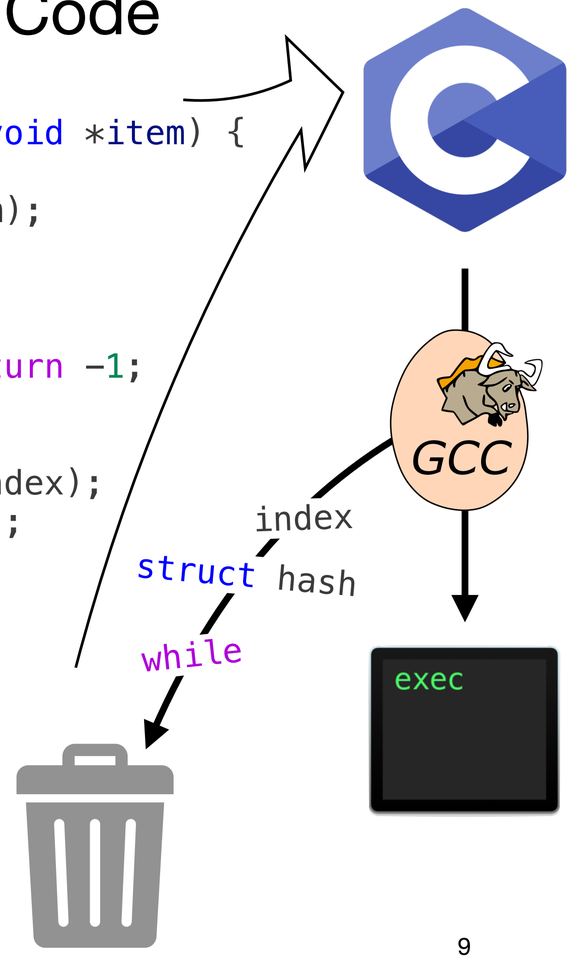
(Human-Readable Compiled)

```
155:   push   %rbp
156:   mov    %rsp,%rbp
159:   sub    $0x20,%rsp
15d:   mov    %rdi,-0x18(%rbp)
161:   mov    %rsi,-0x20(%rbp)
165:   mov    -0x20(%rbp),%rdx
169:   mov    -0x18(%rbp),%rax
16d:   mov    %rdx,%rsi
170:   mov    %rax,%rdi
173:   callq  0x94
178:   mov    %eax,-0xc(%rbp)
17b:   movl   $0x0,-0x10(%rbp)
182:   mov    -0x18(%rbp),%rax
186:   mov    0x8(%rax),%rax
18a:   mov    -0xc(%rbp),%edx
18d:   mov    %edx,%esi
18f:   mov    %rax,%rdi
192:   callq  0x5e
197:   mov    %rax,-0x8(%rbp)
19b:   jmp    0x200
19d:   mov    -0x10(%rbp),%eax
1a0:   lea    0x1(%rax),%edx
1a3:   mov    %edx,-0x10(%rbp)
```

Compilation is *lossy*: some abstractions present in source code are discarded

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



Assembly

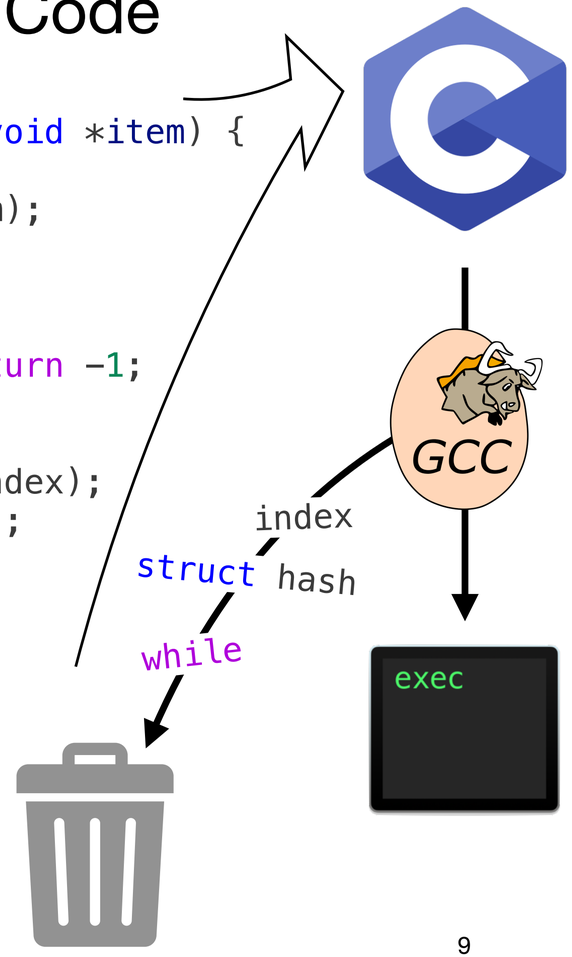
(Human-Readable Compiled)

```
155:   push   %rbp
156:   mov    %rsp,%rbp
159:   sub    $0x20,%rsp
15d:   mov    %rdi,-0x18(%rbp)
161:   mov    %rsi,-0x20(%rbp)
165:   mov    -0x20(%rbp),%rdx
169:   mov    -0x18(%rbp),%rax
16d:   mov    %rdx,%rsi
170:   mov    %rax,%rdi
173:   callq  0x94
178:   mov    %eax,-0xc(%rbp)
17b:   movl   $0x0,-0x10(%rbp)
182:   mov    -0x18(%rbp),%rax
186:   mov    0x8(%rax),%rax
18a:   mov    -0xc(%rbp),%edx
18d:   mov    %edx,%esi
18f:   mov    %rax,%rdi
192:   callq  0x5e
197:   mov    %rax,-0x8(%rbp)
19b:   jmp    0x200
19d:   mov    -0x10(%rbp),%eax
1a0:   lea    0x1(%rax),%edx
1a3:   mov    %edx,-0x10(%rbp)
```

Compilation is *lossy*: some abstractions present in source code are discarded

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



No identifier names

Assembly

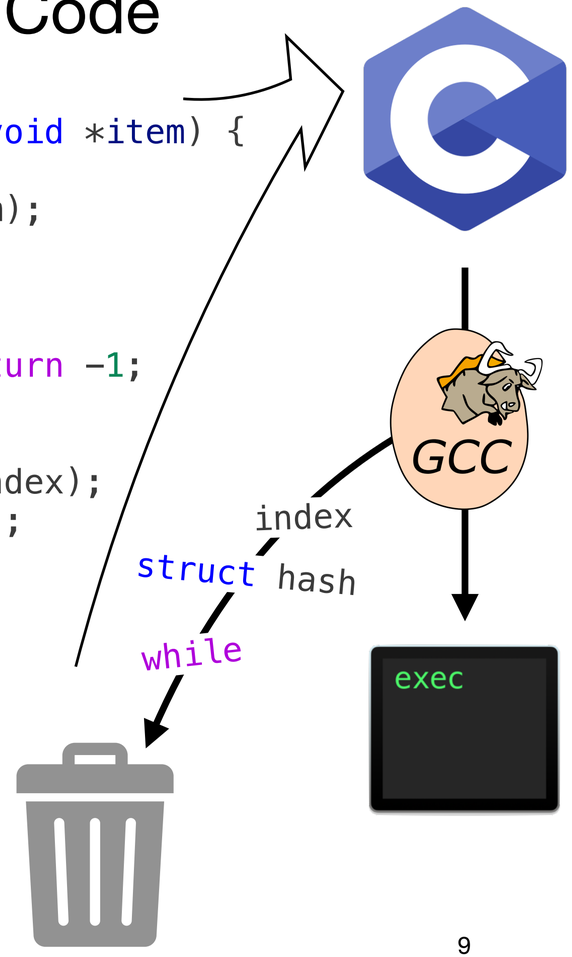
(Human-Readable Compiled)

```
155:   push   %rbp
156:   mov    %rsp,%rbp
159:   sub    $0x20,%rsp
15d:   mov    %rdi,-0x18(%rbp)
161:   mov    %rsi,-0x20(%rbp)
165:   mov    -0x20(%rbp),%rdx
169:   mov    -0x18(%rbp),%rax
16d:   mov    %rdx,%rsi
170:   mov    %rax,%rdi
173:   callq  0x94
178:   mov    %eax,-0xc(%rbp)
17b:   movl   $0x0,-0x10(%rbp)
182:   mov    -0x18(%rbp),%rax
186:   mov    0x8(%rax),%rax
18a:   mov    -0xc(%rbp),%edx
18d:   mov    %edx,%esi
18f:   mov    %rax,%rdi
192:   callq  0x5e
197:   mov    %rax,-0x8(%rbp)
19b:   jmp    0x200
19d:   mov    -0x10(%rbp),%eax
1a0:   lea    0x1(%rax),%edx
1a3:   mov    %edx,-0x10(%rbp)
```

Compilation is *lossy*: some abstractions present in source code are discarded

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



No identifier names

Assembly

(Human-Readable Compiled)

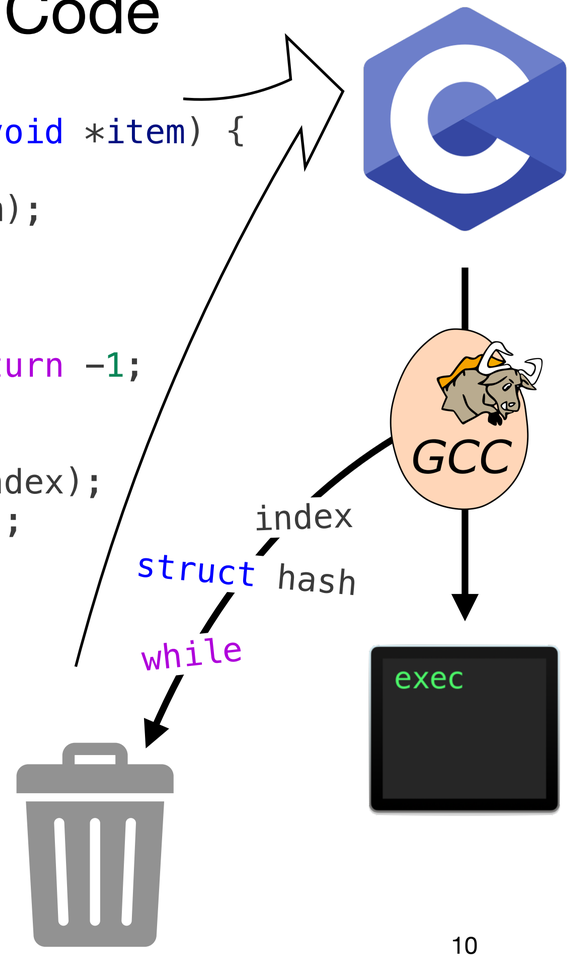
```
155:   push   %rbp
156:   mov    %rsp,%rbp
159:   sub    $0x20,%rsp
15d:   mov    %rdi,-0x18(%rbp)
161:   mov    %rsi,-0x20(%rbp)
165:   mov    -0x20(%rbp),%rdx
169:   mov    -0x18(%rbp),%rax
16d:   mov    %rdx,%rsi
170:   mov    %rax,%rdi
173:   callq  0x94
178:   mov    %eax,-0xc(%rbp)
17b:   movl   $0x0,-0x10(%rbp)
182:   mov    -0x18(%rbp),%rax
186:   mov    0x8(%rax),%rax
18a:   mov    -0xc(%rbp),%edx
18d:   mov    %edx,%esi
18f:   mov    %rax,%rdi
192:   callq  0x5e
197:   mov    %rax,-0x8(%rbp)
19b:   jmp    0x200
19d:   mov    -0x10(%rbp),%eax
1a0:   lea    0x1(%rax),%edx
1a3:   mov    %edx,-0x10(%rbp)
```

Control flow is gotos

Assembly is difficult for humans to read.

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

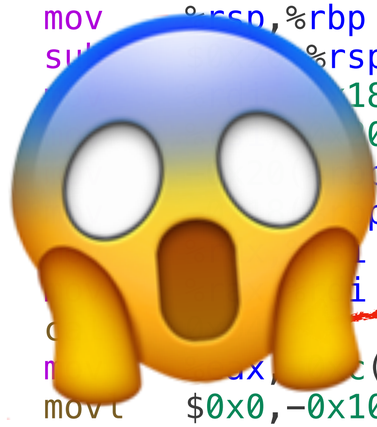


No identifier names

Assembly

(Human-Readable Compiled)

```
155:   push   %rbp
156:   mov    %rsp,%rbp
159:   sub    $0x20,%rsp
15d:   mov    %edi,-0x18(%rbp)
161:   mov    %rsi,-0x20(%rbp)
165:   mov    -0x18(%rbp),%rdx
169:   mov    -0x20(%rbp),%rax
16d:   callq  0x0
170:   mov    %rax,%rdi
173:   callq  0x0
178:   mov    %rax,-0xc(%rbp)
17b:   movl   $0x0,-0x10(%rbp)
182:   mov    -0x18(%rbp),%rax
186:   mov    0x8(%rax),%rax
18a:   mov    -0xc(%rbp),%edx
18d:   mov    %edx,%esi
18f:   mov    %rax,%rdi
192:   callq  0x5e
197:   mov    %rax,-0x8(%rbp)
19b:   jmp    0x200
19d:   mov    -0x10(%rbp),%eax
1a0:   lea    0x1(%rax),%edx
1a3:   mov    %edx,-0x10(%rbp)
```

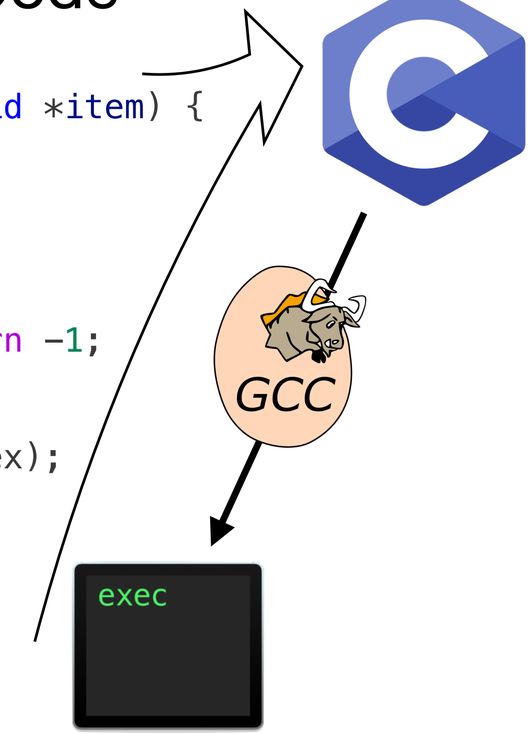


Control flow is gotos

Decompilers help by reversing the compilation process.

Original Source Code

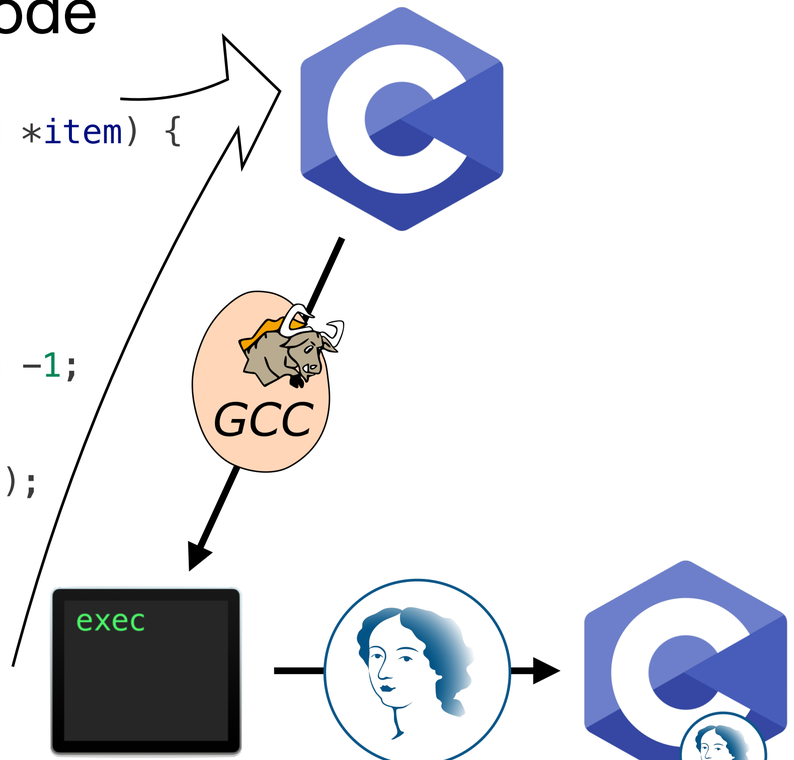
```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



Decompilers help by reversing the compilation process.

Original Source Code

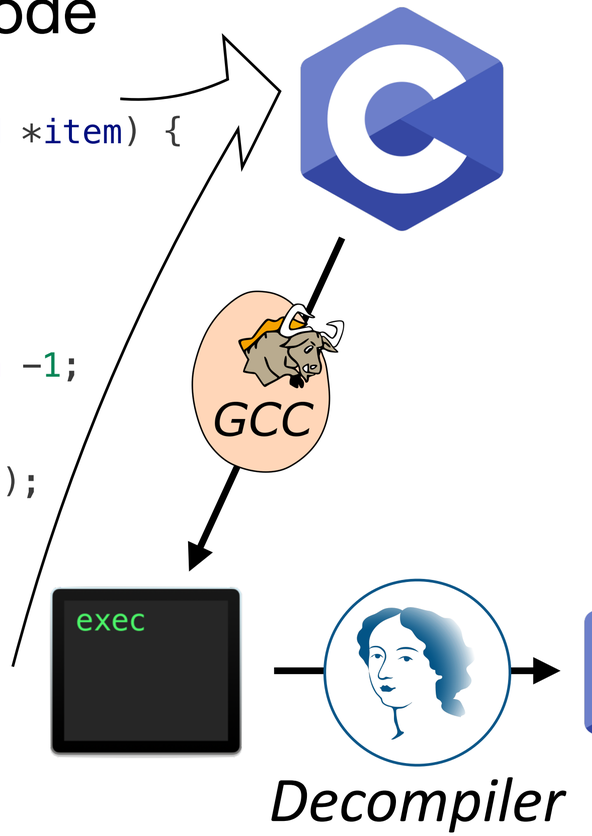
```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



Decompilers help by reversing the compilation process.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



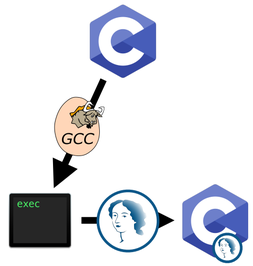
Decompiled Code

```c
__int64 __fastcall func4(__int64 a1, __int64 a2){
  int v2;           // eax
  __int64 result;   // rax
  int v4;           // [rsp+10h] [rbp-10h]
  unsigned int v5; // [rsp+14h] [rbp-Ch]
  __int64 i;        // [rsp+18h] [rbp-8h]

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD *)(a1 + 8), v5); i;
       i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

However, decompiled code is still much less easy to read than the original.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



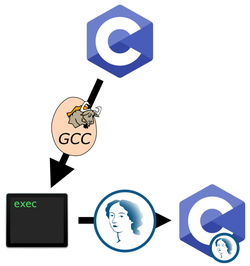
Decompiled Code

```c
__int64 __fastcall func4(__int64 a1, __int64 a2){
  int v2;           // eax
  __int64 result;   // rax
  int v4;           // [rsp+10h] [rbp-10h]
  unsigned int v5; // [rsp+14h] [rbp-Ch]
  __int64 i;        // [rsp+18h] [rbp-8h]

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD *)(a1 + 8), v5); i;
       i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

However, decompiled code is still much less easy to read than the original.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



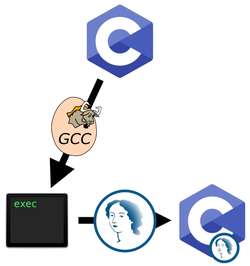
Decompiled Code

```c
__int64 __fastcall func4(__int64 a1, __int64 a2){
  int v2;           // eax
  __int64 result;   // rax
  int v4;           // [rsp+10h] [rbp-10h]
  unsigned int v5; // [rsp+14h] [rbp-Ch]
  __int64 i;        // [rsp+18h] [rbp-8h]

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD *)(a1 + 8), v5); i;
       i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

However, decompiled code is still much less easy to read than the original.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



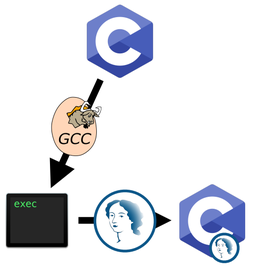
Decompiled Code

```c
__int64 __fastcall func4(__int64 a1, __int64 a2)
{
  int v2;           // eax
  __int64 result;   // rax
  int v4;           // [rsp+10h] [rbp-10h]
  unsigned int v5; // [rsp+14h] [rbp-Ch]
  __int64 i;        // [rsp+18h] [rbp-8h]

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD *)(a1 + 8), v5); i;
       i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

However, decompiled code is still much less easy to read than the original.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



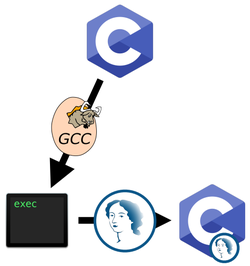
Decompiled Code

```c
__int64 __fastcall func4(__int64 a1, __int64 a2)
{
  int v2;           // eax
  __int64 result;   // rax
  int v4;           // [rsp+10h] [rbp-10h]
  unsigned int v5; // [rsp+14h] [rbp-Ch]
  __int64 i;        // [rsp+18h] [rbp-8h]

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD *)(a1 + 8), v5); i;
       i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

However, decompiled code is still much less easy to read than the original.

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



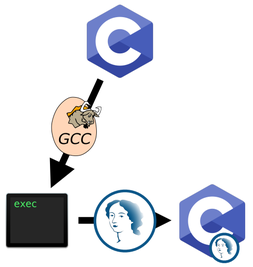
Decompiled Code

```
__int64 __fastcall func4(__int64 a1, __int64 a2)
{
  int v2;           // eax
  __int64 result;   // rax
  int v4;           // [rsp+10h] [rbp-10h]
  unsigned int v5; // [rsp+14h] [rbp-Ch]
  __int64 i;        // [rsp+18h] [rbp-8h]

  v5 = func2(a1, a2);
  v4 = 0;
  for ( i = func1(*(_QWORD *)(a1 + 8), v5); i;
        i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

However, decompiled code is still much less easy to read than the original.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



Decompiled Code

```c
__int64 __fastcall func4(__int64 a1, __int64 a2)
{
  int v2;           // eax
  __int64 result;   // rax
  int v4;           // [rsp+10h] [rbp-10h]
  unsigned int v5; // [rsp+14h] [rbp-Ch]
  __int64 i;        // [rsp+18h] [rbp-8h]

  v5 = func2(a1, a2);
  v4 = 0;
  for ( i = func1(*(_QWORD *)(a1 + 8), v5); i;
        i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

In general, arbitrary transformations are required to match the original source.

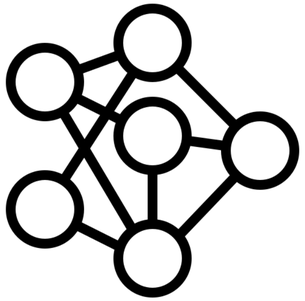
Original Source Code

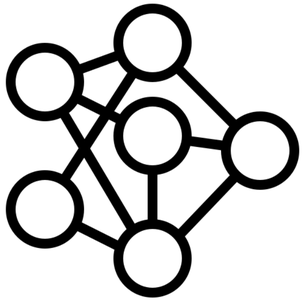
```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

Decompiled Code

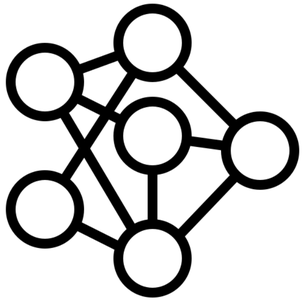
```c
__int64 __fastcall func4(__int64 a1, __int64 a2) {
  int v2;
  __int64 result;
  int v4;
  unsigned int v5;
  __int64 i;

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD *)(a1 + 8), v5); i;
       i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

Discarded Details

Discarded Details $\Longrightarrow$  Probabilistic Prediction

Discarded Details $\Longrightarrow$  Probabilistic Prediction

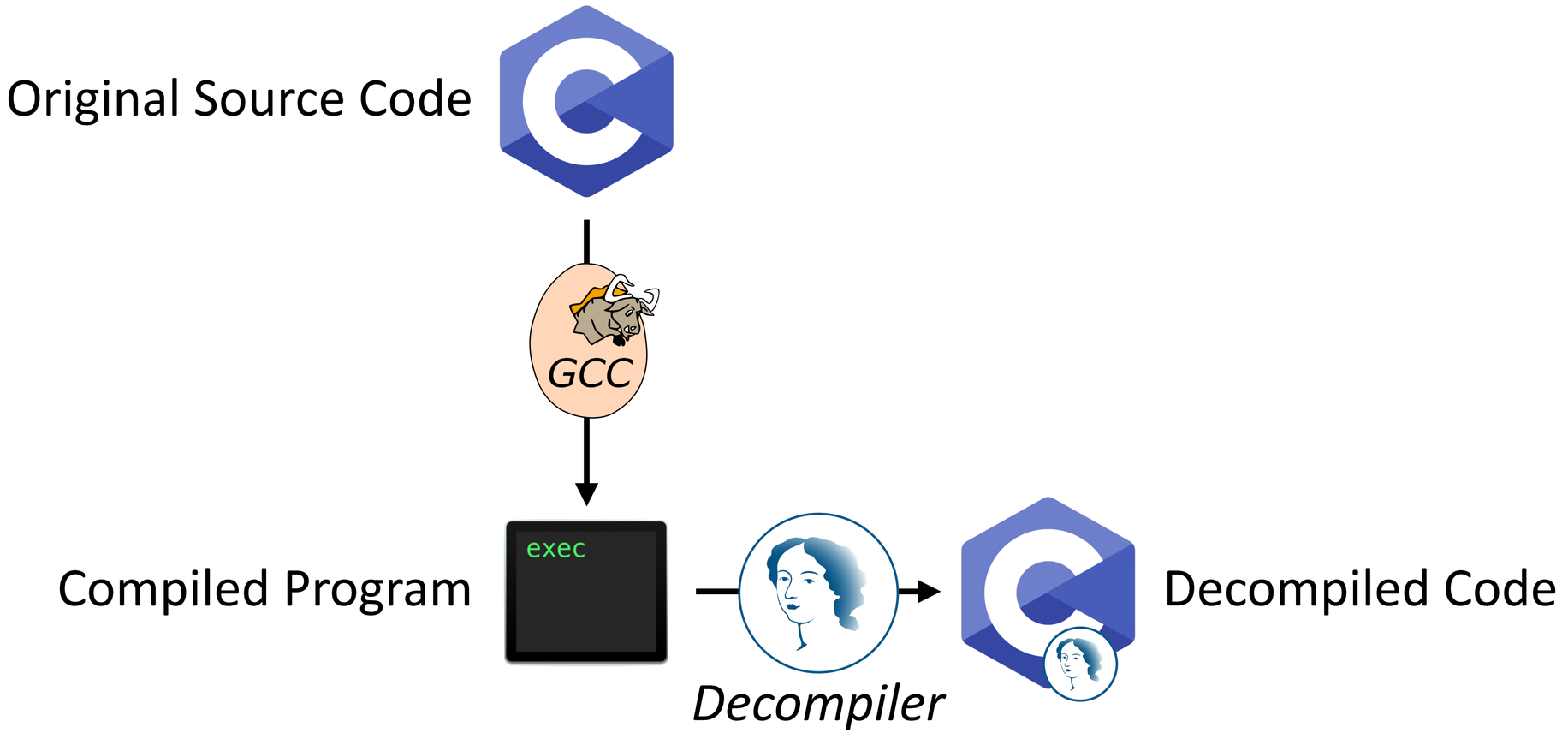
Best performing models for code processing

Discarded Details $\Longrightarrow$  Probabilistic Prediction

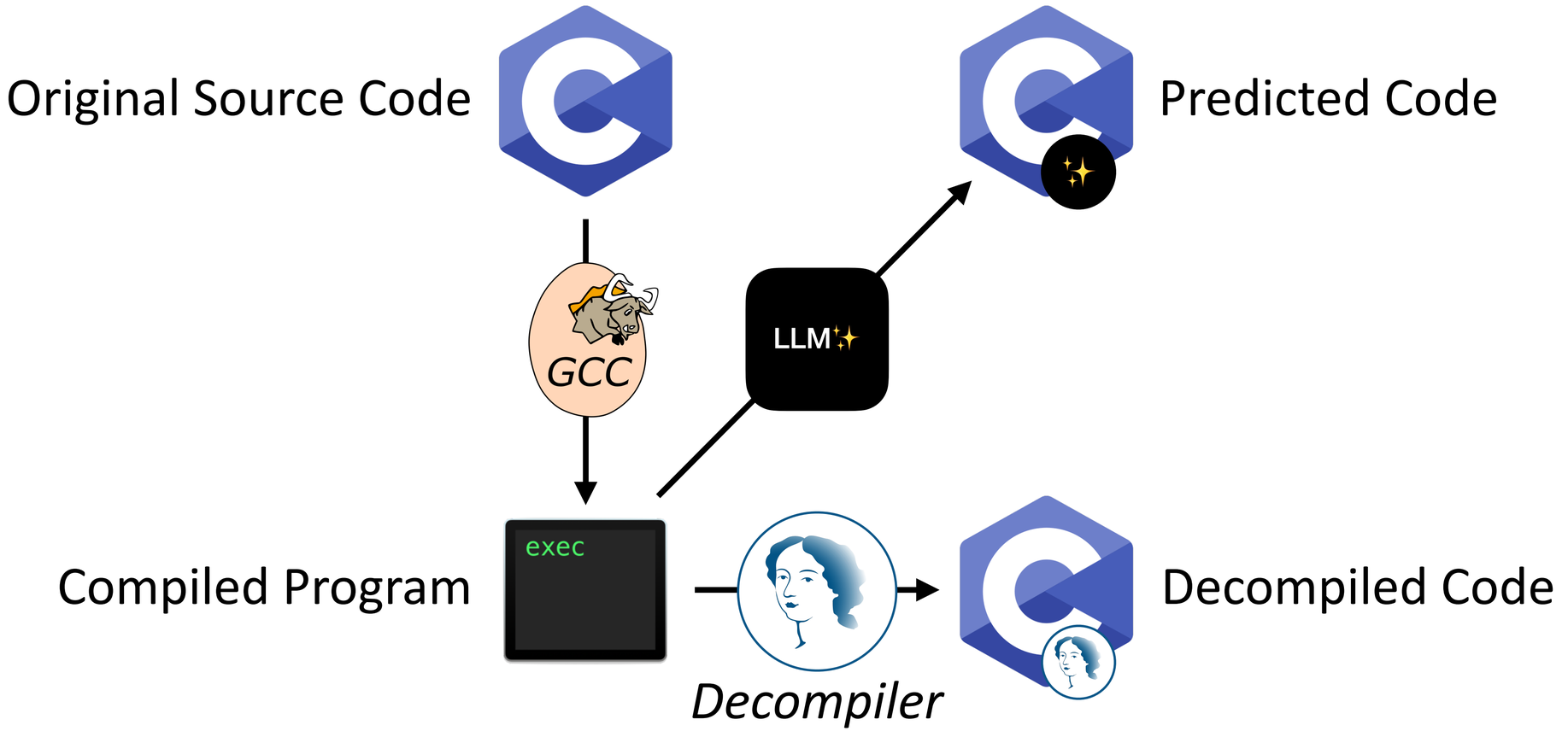
LLM✨ ← Best performing models for code processing

We fine-tuned a *local* LLM to perform this task

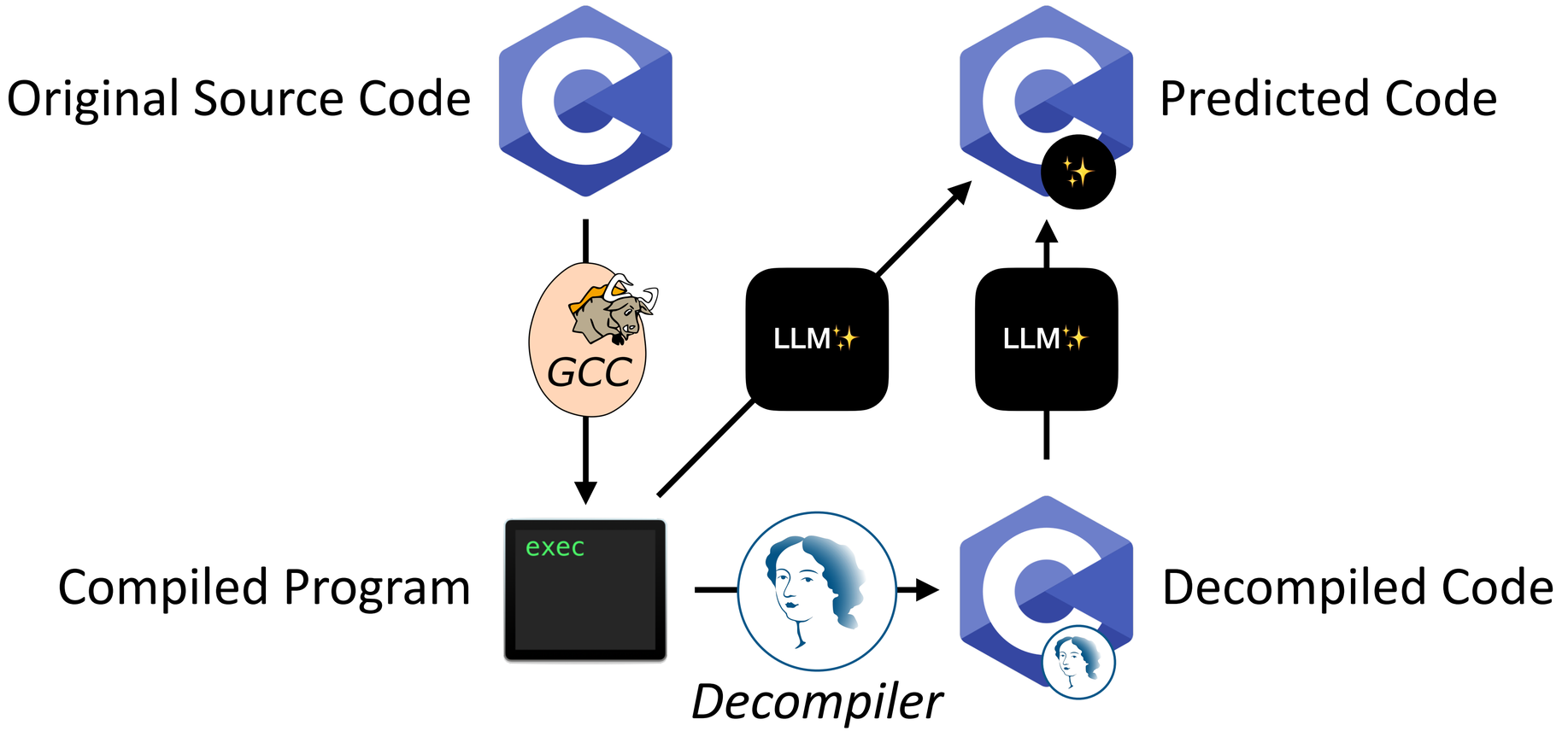
There are two main ways of doing neural decompilation



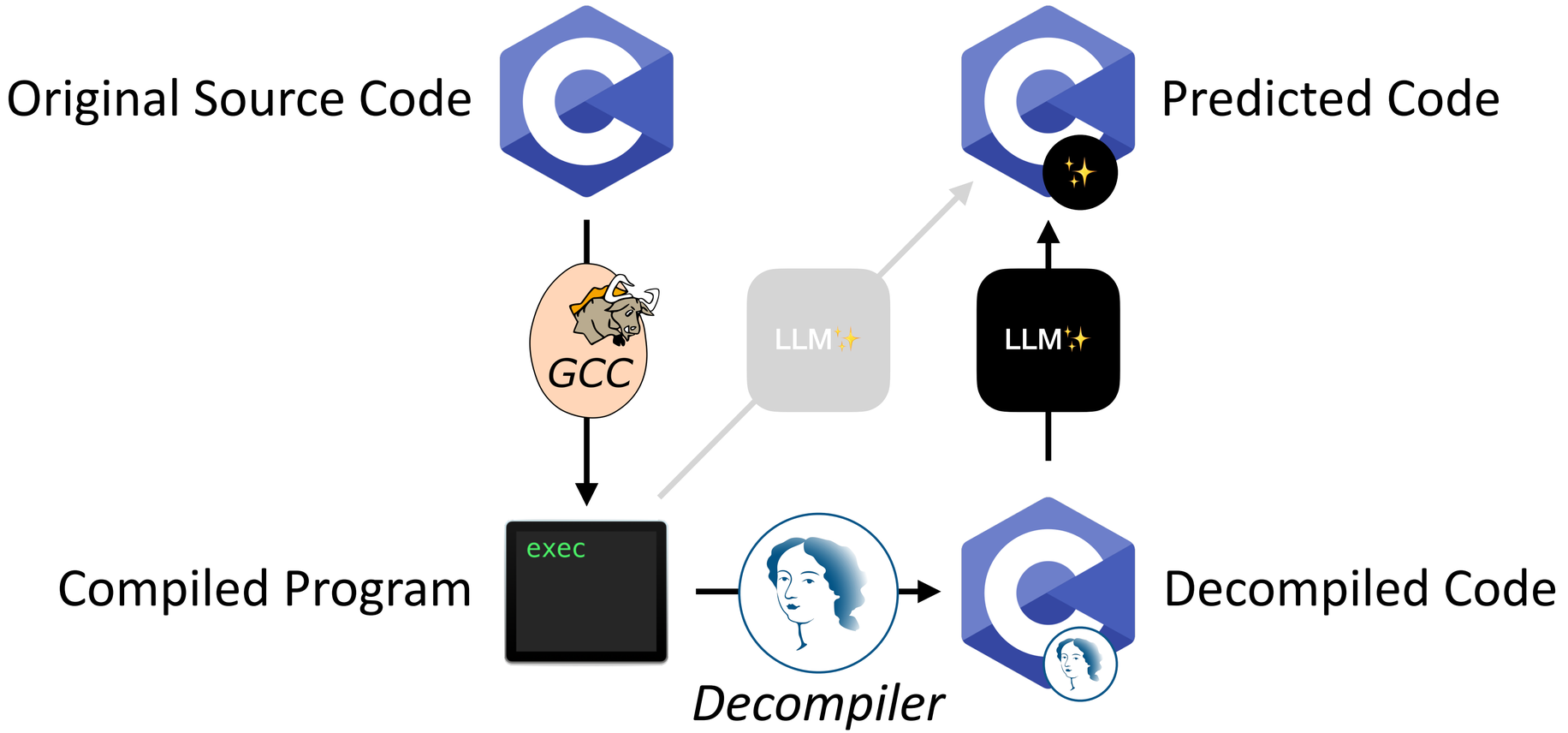
There are two main ways of doing neural decompilation



There are two main ways of doing neural decompilation



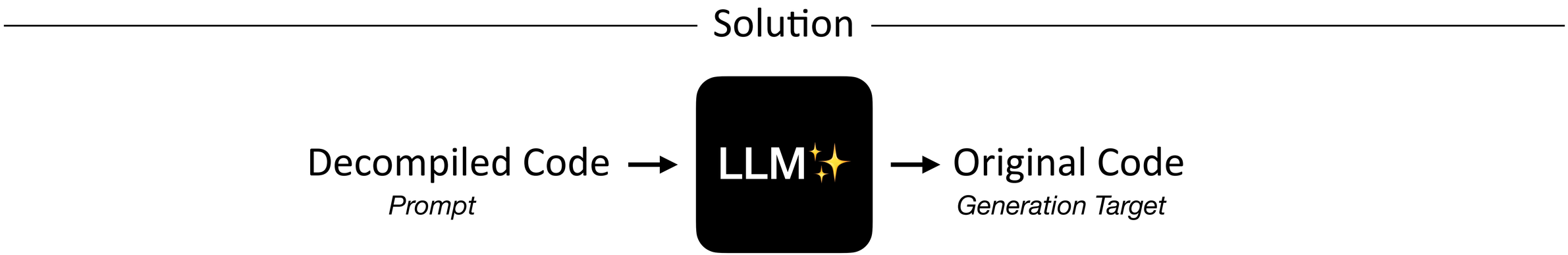
We treat the LLM as the last stage of decompilation



This type of textual transformation forms the basis for our solution, which we call *Idioms*

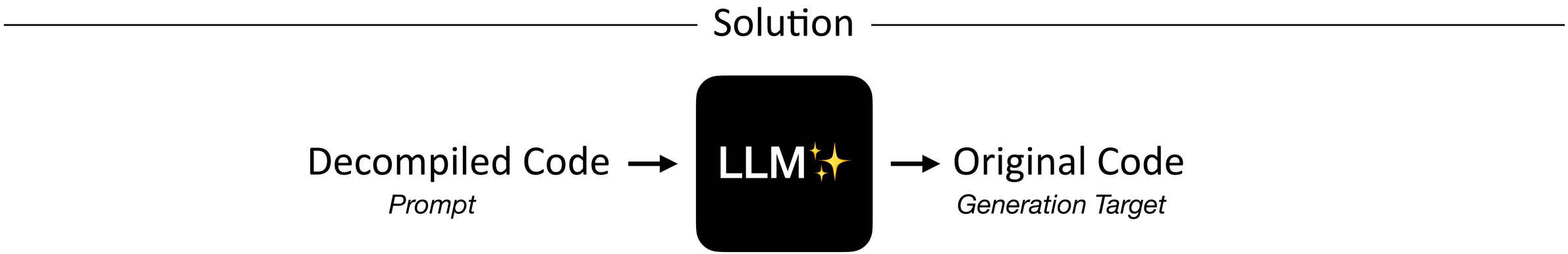
Solution

This type of textual transformation forms the basis for our solution, which we call *Idioms*



This type of textual transformation forms the basis for our solution, which we call *Idioms*

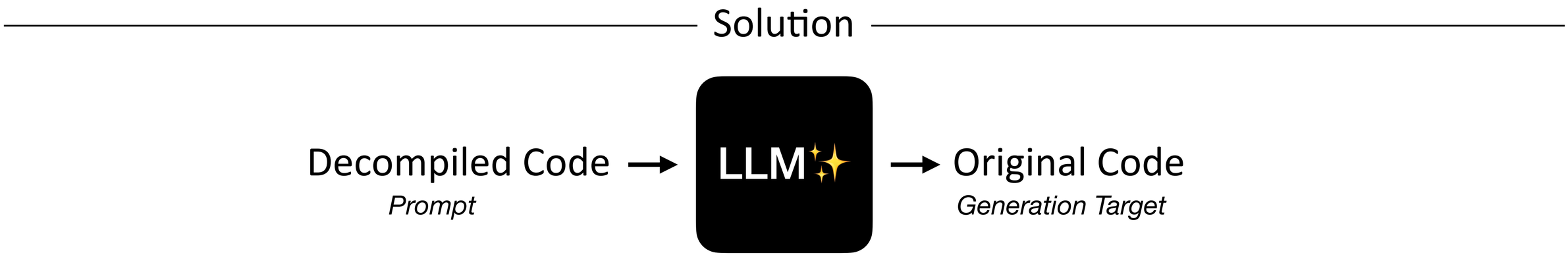
We build upon this base by incorporating problem-specific insights.



This type of textual transformation forms the basis for our solution, which we call *Idioms*

We build upon this base by incorporating problem-specific insights.

We focus on things that are *fundamental*: those which remain true even as general LLM performance improves over time.



Insight 1: Types are integral to the structure of the code.

Types are integral to the structure of the code.

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

Decompiled Code

```
__int64 __fastcall func4(__int64 a1, __int64 a2) {
  int v2;
  __int64 result;
  int v4;
  unsigned int v5;
  __int64 i;

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD *)(a1 + 8), v5); i;
       i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

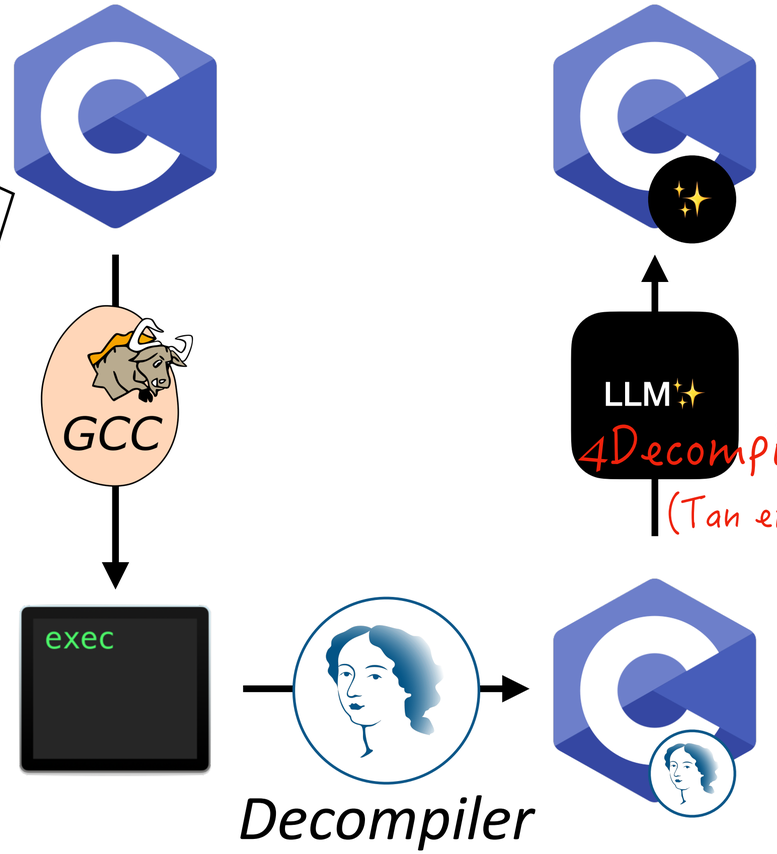
Pointer arithmetic to access struct members

Pointer dereference to access first struct member

Types are necessary to define the behavior of the function

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



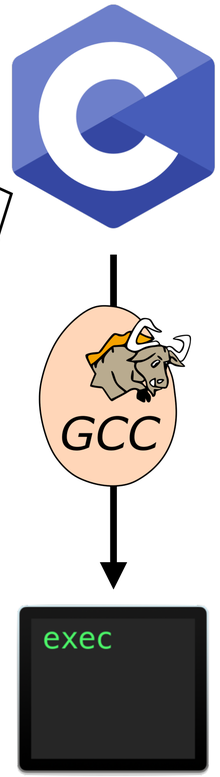
LLM4Decompile Prediction

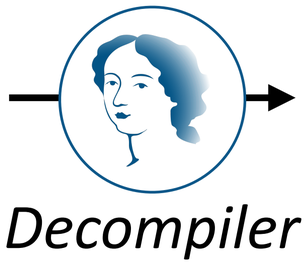
```c
int FUN_00100155 (struct FUN_0009ff84 *VAR_0,
                  void *VAR_1) {
  int VAR_2;
  int VAR_3;
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) {
      return -1;
    }
    if (!VAR_0->VAR_7(VAR_4, VAR_1)) {
      break;
    }
    VAR_2 = FUN_0009ff89(VAR_0, VAR_2);
    VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
    VAR_3++;
  }
  if (VAR_4) {
    return VAR_2;
  }
  return -1;
}
```

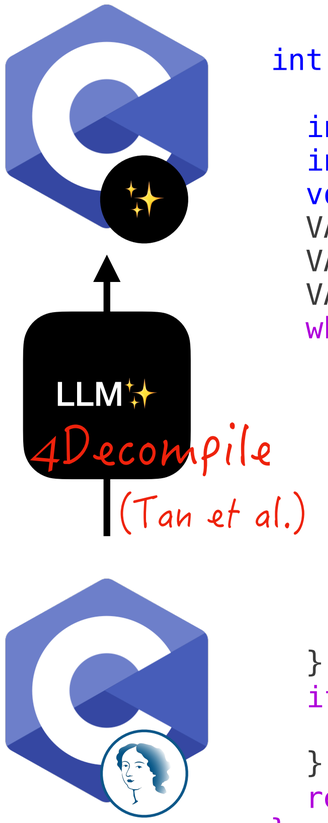
Types are necessary to define the behavior of the function

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```







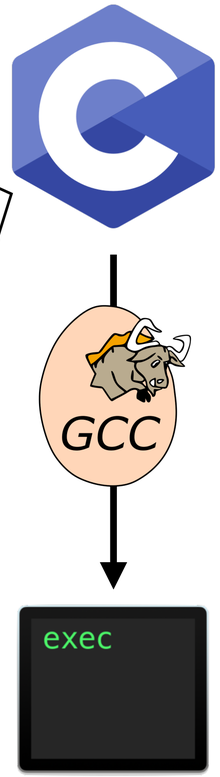
LLM4Decompile Prediction

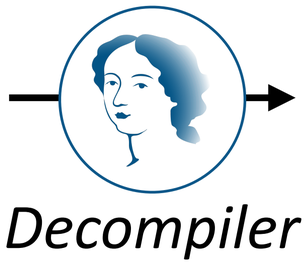
```
struct FUN_0009ff84 *
int FU                    void *VAR_1) {
  int __int64
  int VAR_3;
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) {
      return -1;
    }
    if (!VAR_0->VAR_7(VAR_4, VAR_1)) {
      break;
    }
    VAR_2 = FUN_0009ff89(VAR_0, VAR_2);
    VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
    VAR_3++;
  }
  if (VAR_4) {
    return VAR_2;
  }
  return -1;
}
```

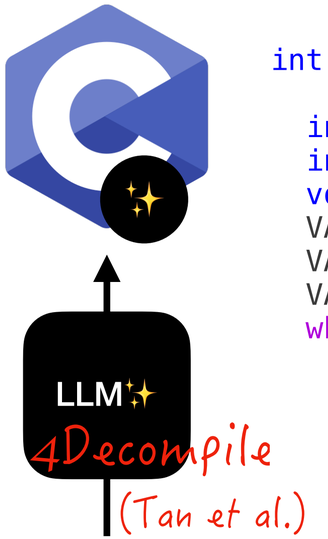
Types are necessary to define the behavior of the function

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```







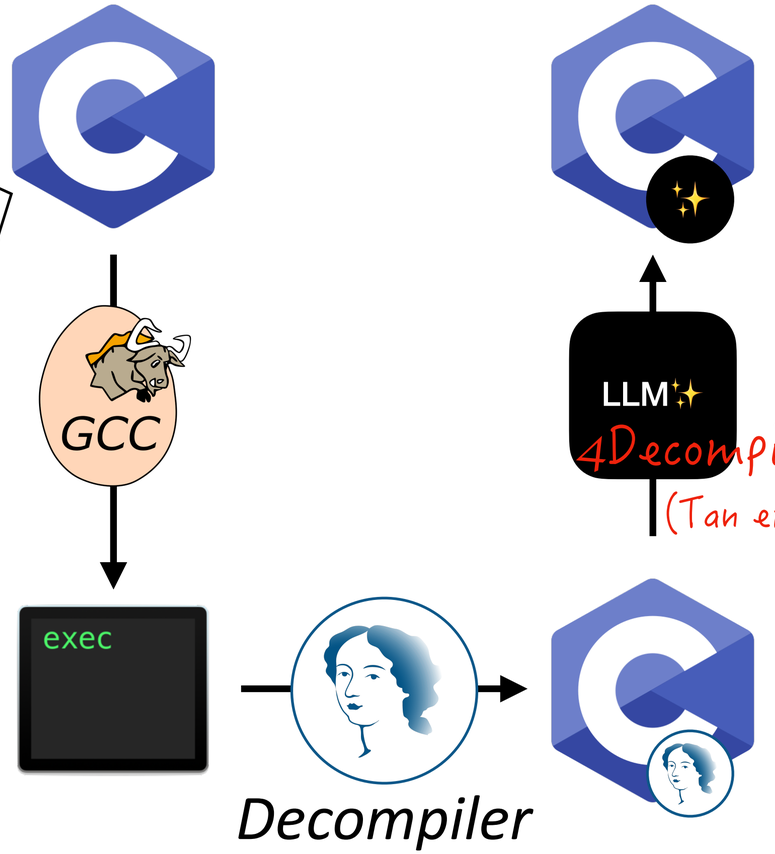
LLM4Decompile Prediction

```c
struct FUN_0009ff84 *
int FU                    void *VAR_1) {
  int ___                 t64
  int VA                  3;
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) {
      return -1;
    }
    if (!VAR_0->VAR_7(VAR_4, VAR_1)) {
      break;
    }
    VAR_2 = FUN_0009ff89(VAR_0, VAR_2);
    VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
    VAR_3++;
  }
  if (VAR_4) {
    return VAR_2;
  }
  return -1;
}
```

Types are necessary to define the behavior of the function

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



LLM4Decompile Prediction

```c
int FU                                    void *VAR_1) {
  int ___                                 t64
  int VA                                  3;
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) {
      return -1;
    }
    if (!VAR_0->VAR_7(VAR_4, VAR_1)) {
      VAR_2 = FUN_00                      R_2);
      VAR_4 = FUN_00
      VAR_3++;
    }
    if (VAR_4) {
      return VAR_2;
    }
    return -1;
  }
```

struct FUN_0009ff84 *

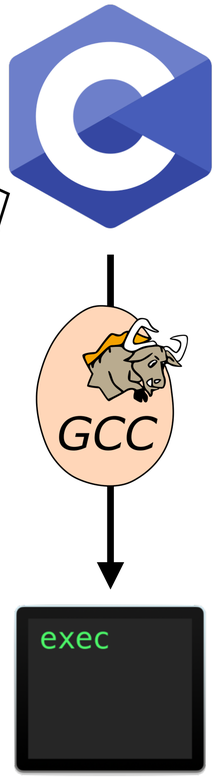
*(_QWORD *)(a1 + 8)

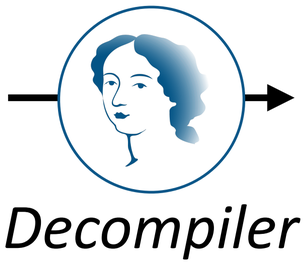
VAR_0->VAR_5

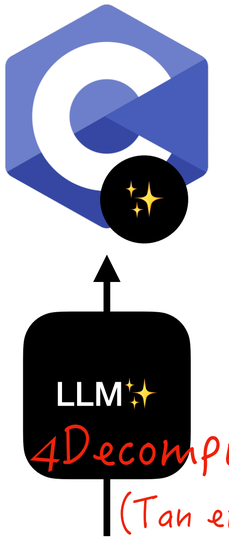
Types are necessary to define the behavior of the function

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```







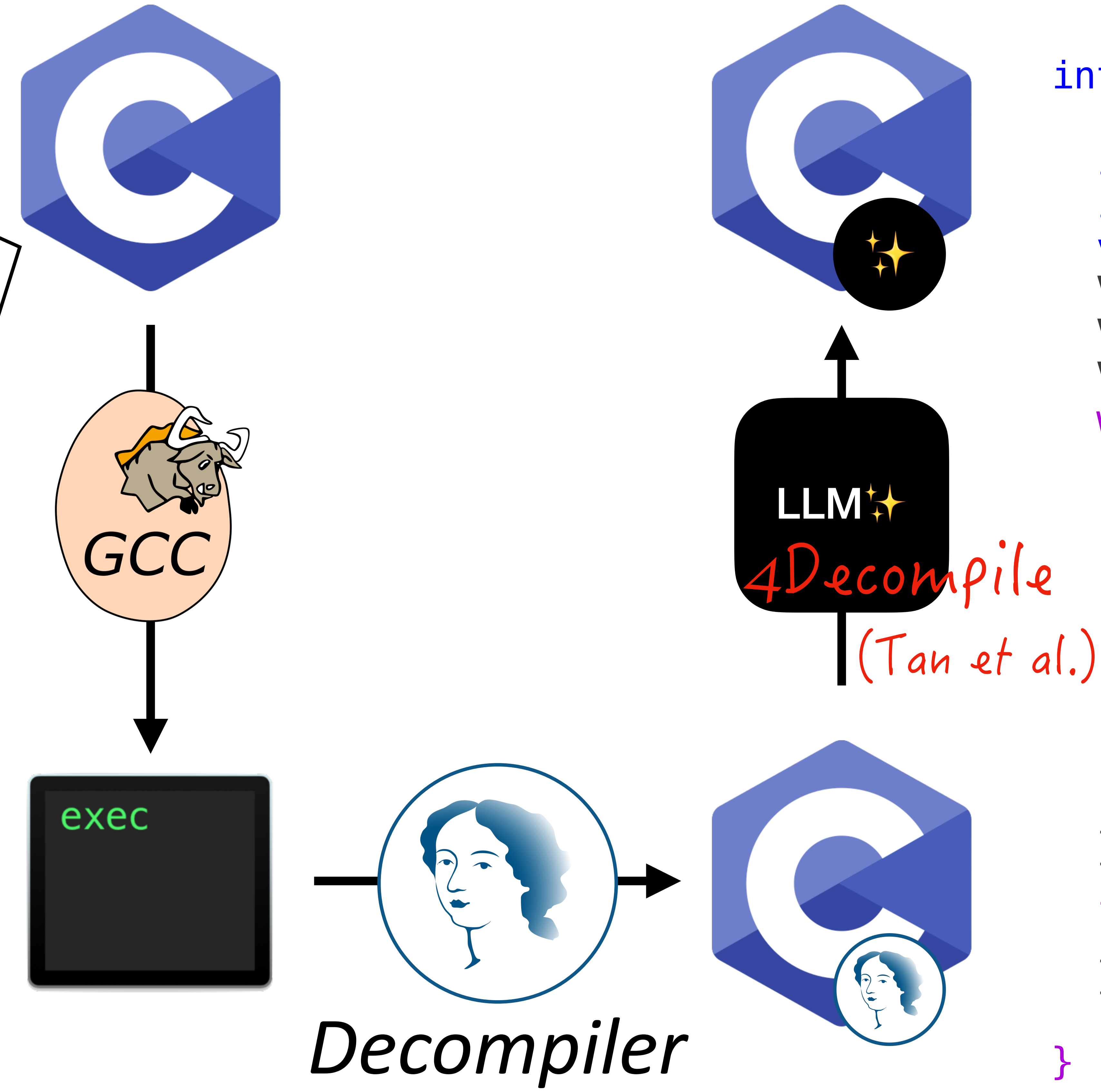
LLM4Decompile Prediction

```c
int FU                                    struct FUN_0009ff84 *
                                          void *VAR_1) {
  int ___                                 t64
  int VA
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) {
      return -1;
    }
    if (!VAR_0->VA                        7(  AR_4, VAR_1)) {
      *(_QWORD                            )(a1 + 8)
      VAR_2 = FUN_0
      VAR_4 = FUN_00                      VAR_0->VAR_5      _2);
      VAR_3++;
    }
    if (VAR_4) {
      return VAR_2;
    }
    return -1;
  }
```

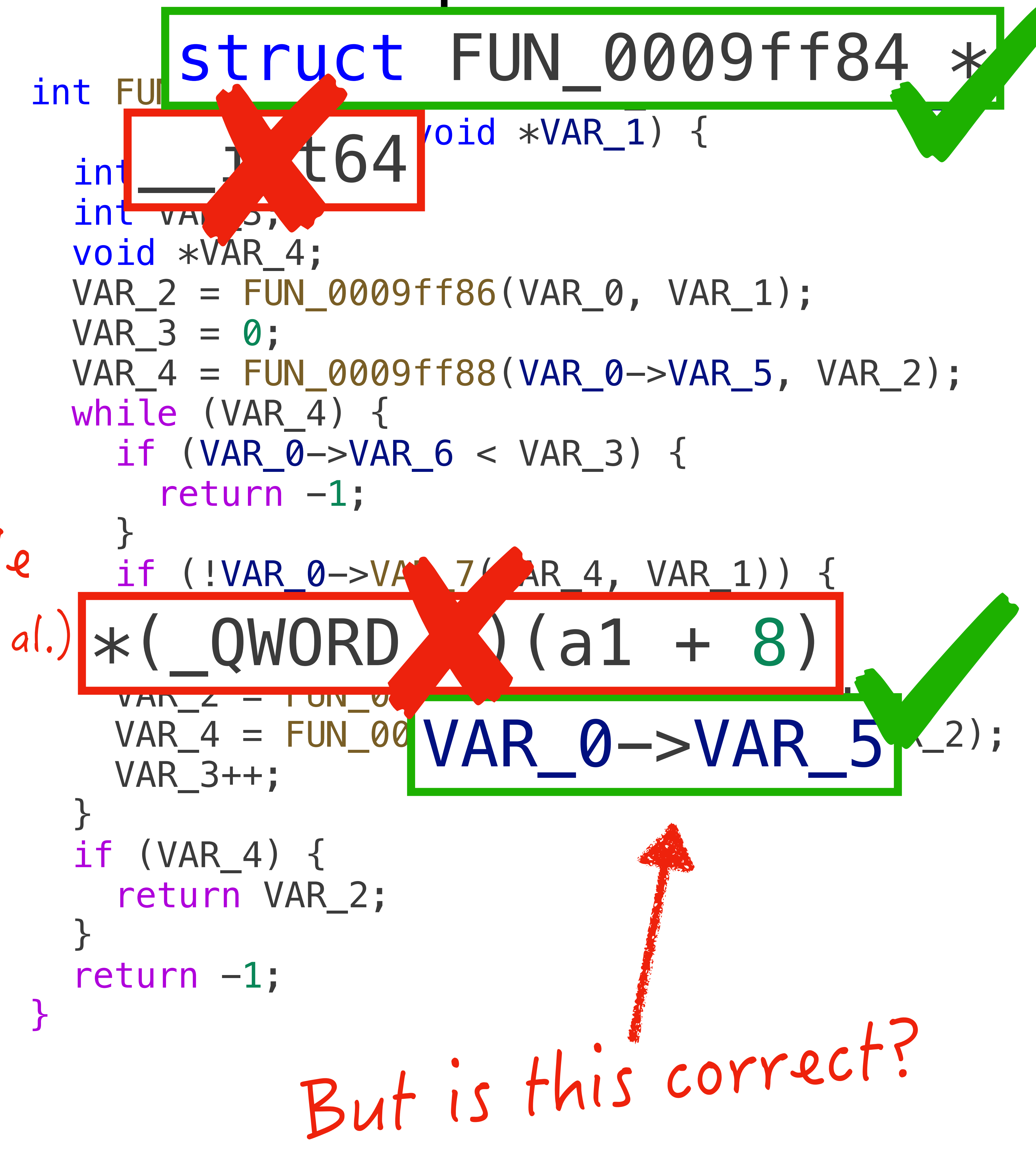
Types are necessary to define the behavior of the function

Original Source Code

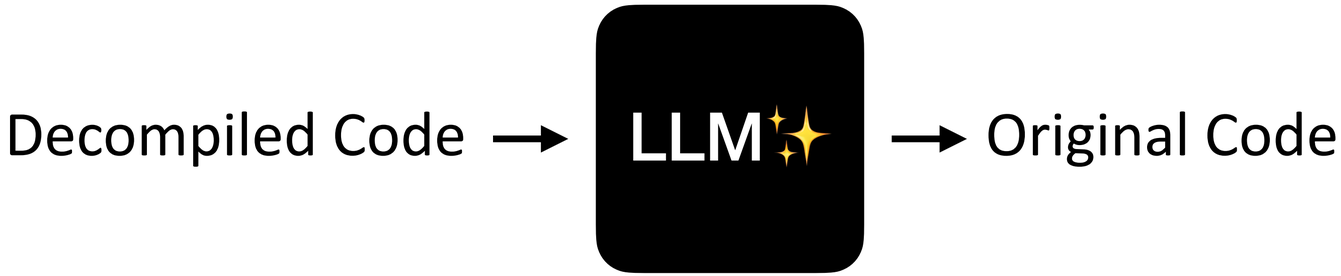
```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```



LLM4Decompile Prediction

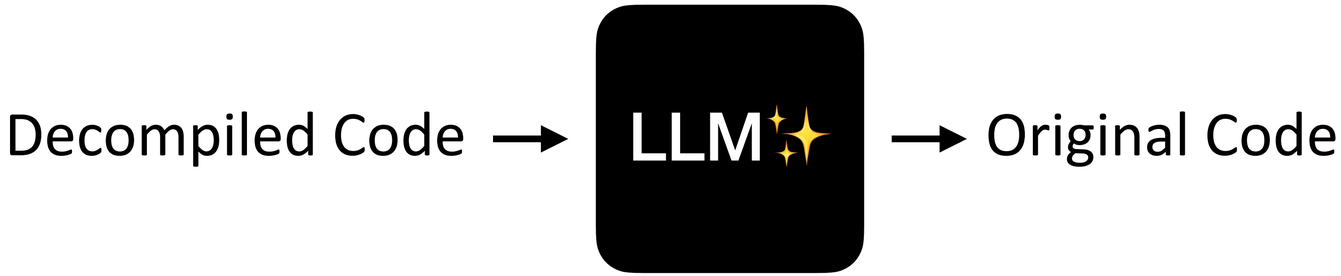


Solution



Solution: Predict necessary type definitions along with the original code

Solution



Solution: Predict necessary type definitions along with the original code

Solution

Our solution, *Idioms*, jointly predicts code and types.

Our Solution (*Idioms*) Prediction

```c
int hash_find(struct hash_t *hash, void *key)
{
    int index = hash_index(hash, key);
    int i = 0;
    void *item = hash_get(hash->table, index);
    while (item != ((void *)0)) {
        if (i++ > hash->size) {
            return -1;
        }
        if (hash->cmp(item, key) == 0) {
            break;
        }
        index = hash_next(hash, index);
        item = hash_get(hash->table, index);
    }
    return (item == ((void *)0)) ? -1 : index;
}

struct hash_t {
    int size;
    int count;
    struct hash_table_t *table;
    int (*hash)(void *key);
    int (*cmp)(void *key1, void *key2);
};
```

```c
struct hash_table_t {
    int size;
    void **items;
};
```

LLM4Decompile Prediction

```c
int FUN_00100155 (struct FUN_0009ff84 *VAR_0,
                  void *VAR_1) {
  int VAR_2;
  int VAR_3;
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) {
      return -1;
    }
    if (!VAR_0->VAR_7(VAR_4, VAR_1)) {
      break;
    }
    VAR_2 = FUN_0009ff89(VAR_0, VAR_2);
    VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
    VAR_3++;
  }
  if (VAR_4) {
    return VAR_2;
  }
  return -1;
}
```

Our solution, *Idioms*, jointly predicts code and types.

Our Solution (*Idioms*) Prediction

```
int hash_find(struct hash_t *hash, void *key)
{
    int index = hash_index(hash, key);
    int i = 0;
    void *item = hash_get(hash->table, index);
    while (item != ((void *)0)) {
        if (i++ > hash->size) {
            return -1;
        }
        if (hash->cmp(item, key) == 0) {
            break;
        }
        index
        item =
    }
    return (item == ((void *)0)) ? -1 : index;
}
```

What's a hash->table?

```
struct hash_t {
    int size;
    int count;
    struct hash_table_t *table;
    int (*hash)(void *key);
    int (*cmp)(void *key1, void *key2);
};
```

```
struct hash_table_t {
    int size;
    void **items;
};
```

LLM4Decompile Prediction

```
int FUN_00100155 (struct FUN_0009ff84 *VAR_0,
                  void *VAR_1) {
  int VAR_2;
  int VAR_3;
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) {
      return -1;
    }
    if (!VAR_0->VAR_7(VAR_4, VAR_1)) {
      break;
    }
    VAR_2 = FUN_0009ff89(VAR_0, VAR_2);
    VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
    VAR_3++;
  }
  if (VAR_4) {
    return VAR_2;
  }
  return -1;
}
```

Our solution, *Idioms*, jointly predicts code and types.

Our Solution (*Idioms*) Prediction

```
int hash_find(struct hash_t *hash, void *key)
{
    int index = hash_index(hash, key);
    int i = 0;
    void *item = hash_get(hash->table, index);
    while (item != ((void *)0)) {
        if (i++ > hash->size) {
            return -1;
        }
        if (hash->cmp(item, key) == 0) {
            break;
        }
        index
        item =
    }
    return (item == ((void *)0)) ? -1 : index;
}

struct hash_t {
    int size;
    int count;
    struct hash_table_t *table;
    int (*hash)(void *key);
    int (*cmp)(void *key1, void *key2);
};
```

What's a hash->table?

```
struct hash_table_t {
    int size;
    void **items;
};
```

LLM4Decompile Prediction

```
int FUN_00100155 (struct FUN_0009ff84 *VAR_0,
                  void *VAR_1) {
  int VAR_2;
  int VAR_3;
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) {
      return -1;
    }
    if (!VAR_0->VAR_7(VAR_4, VAR_1)) {
      break;
    }
    VAR_2 = FUN_0009ff89(VAR_0, VAR_2);
    VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
    VAR_3++;
  }
  if (VAR_4) {
    return VAR_2;
  }
  return -1;
}
```

Our solution, *Idioms*, jointly predicts code and types.

Our Solution (*Idioms*) Prediction

```c
int hash_find(struct hash_t *hash, void *key)
{
    int index = hash_index(hash, key);
    int i = 0;
    void *item = hash_get(hash->table, index);
    while (item != ((void *)0)) {
        if (i++ > hash->size) {
            return -1;
        }
        if (hash->cmp(item, key) == 0) {
            break;
        }
        index
        item =
    }
    return (item == ((void *)0)) ? -1 : index;
}

struct hash_t {
    int size;
    int count;
    struct hash_table_t *table;
    int (*hash)(void *key);
    int (*cmp)(void *key1, void *key2);
};

struct hash_table_t {
    int size;
    void **items;
};
```

What's a hash->table?

LLM4Decompile Prediction

```c
int FUN_00100155 (struct FUN_0009ff84 *VAR_0,
                  void *VAR_1) {
  int VAR_2;
  int VAR_3;
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) {
      return -1;
    }
    if (!VAR_0->VAR_7(VAR_4, VAR_1)) {
      break;
    }
    VAR_2 = FUN_0009ff89(VAR_0, VAR_2);
    VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
    VAR_3++;
  }
  if (VAR_4) {
    return VAR_2;
  }
  return -1;
}
```

Insight 2: Type recovery requires interprocedural analysis in general—especially for composite types

Type recovery requires interprocedural analysis in general—especially for composite types

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

Decompiled Code

```
__int64 __fastcall func4(__int64 a1, __int64 a2) {
  int v2;
  __int64 result;
  int v4;
  unsigned int v5;
  __int64 i;

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD *)(a1 + 8), v5); i;
       i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

Type recovery requires interprocedural analysis in general—especially for composite types

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

Decompiled Code

```c
__int64 __fastcall func4(__int64 a1, __int64 a2) {
  int v2;
  __int64 result;
  int v4;
  unsigned int v5;
  __int64 i;

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD **)(a1 + 8), v5); i;
       i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

What's the type of the field at a1 + 8?

Type recovery requires interprocedural analysis in general—especially for composite types

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

```
__int64 __fastcall func1(__int64 a1, int a2) {
    __int64 result; // rax

    if ( a2 < *(_DWORD *)a1 )
        result = *(_QWORD *)(*(_QWORD *)(a1 + 8) + 8LL * a2);
    else
        result = 0LL;
    return result;
}
```

Decompiled Code

```
__int64 __fastcall func4(__int64 a1, __int64 a2) {
  int v2;
  __int64 result;
  int v4;
  unsigned int v5;
  __int64 i;

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD **)(a1 + 8), v5); i;
        i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

What's the type of the field at a1 + 8?

Type recovery requires interprocedural analysis in general—especially for composite types

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
}
```

```
__int64 __fastcall func1(__int64 a1, int a2) {
    __int64 result; // rax

    if ( a2 < *(_DWORD *)a1 )
        result = *(_QWORD *)(*(_QWORD *)(a1 + 8) + 8LL * a2);
    else
        result = 0LL;
    return result;
}
```

Decompiled Code

```
__int64 __fastcall func4(__int64 a1, __int64 a2) {
  int v2;
  __int64 result;
  int v4;
  unsigned int v5;
  __int64 i;

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD **)(a1 + 8), v5); i;
        i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

What's the type of the field at a1 + 8?

Type recovery requires interprocedural analysis in general—especially for composite types

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

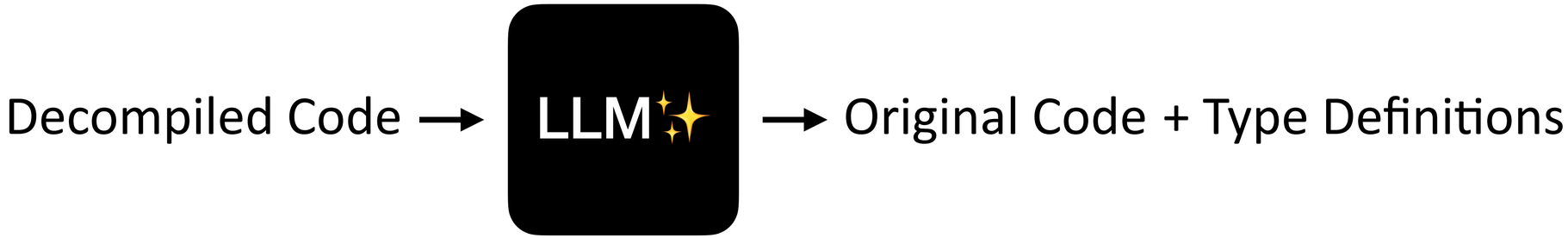
```
__int64 __fastcall func1(__int64 a1, int a2) {
    __int64 result; // rax

    if ( a2 < *(_DWORD *)a1 )
        result = *(_QWORD *)(*(_QWORD *)(a1 + 8) + 8LL * a2);
    else
        result = 0LL;
    return result;
}
```

Decompiled Code

```
__int64 __fastcall func4(__int64 a1, __int64 a2) {
  int v2;
  __int64 result;
  int v4;
  unsigned int v5;
  __int64 i;

  v5 = func2(a1, a2);
  v4 = 0;
  for (i = func1(*(_QWORD **)(a1 + 8), v5); i;
        i = func1(*(_QWORD *)(a1 + 8), v5)) {
    v2 = v4++;
    if (v2 > *(_DWORD *)a1)
      return 0xFFFFFFFFLL;
    if (!(*(unsigned int(__fastcall **)
          (__int64, __int64))(a1 + 24))(i, a2))
      break;
    v5 = func3((_DWORD *)a1, v5);
  }
  if (i)
    result = v5;
  else
    result = 0xFFFFFFFFLL;
  return result;
}
```

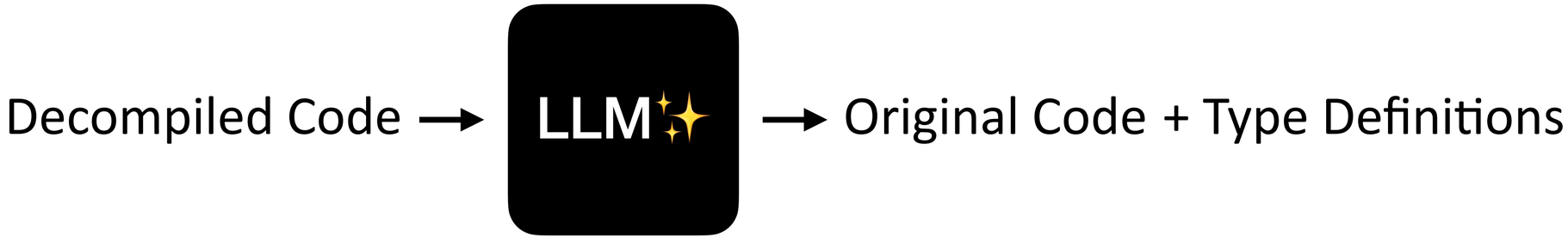
What's the type of the field at a1 + 8?

Solution



Solution: Include the bodies of other decompiled functions in the context

Solution



Solution: Include the bodies of other decompiled functions in the context

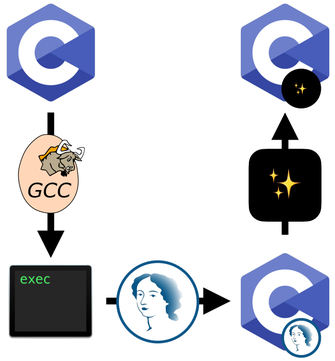
Solution

Decompiled Code + Other Functions → LLM ✨ → Original Code + Type Definitions

The context of the target function alone may be insufficient to correctly interpret that function.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```




```c
struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

```c
struct gap_array {
    int len;
    void **array;
}
```

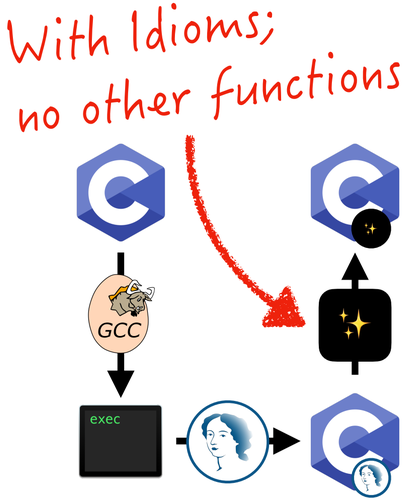
The context of the target function alone may be insufficient to correctly interpret that function.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

```c
struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

```c
struct gap_array {
    int len;
    void **array;
}
```

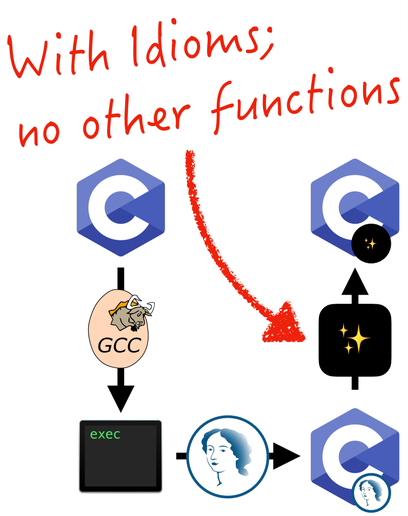


The context of the target function alone may be insufficient to correctly interpret that function.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}

struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

```c
struct gap_array {
    int len;
    void **array;
}
```



Bad Idioms: NO additional context

```c
int hash_table_find(struct hash_table *ht, void *key) {
    struct hash_entry *he;
    int hash_val = hash_table_hash(ht, key);
    int i = 0;
    for (he = hash_table_get(ht->ht, hash_val); he;
         he = hash_table_get(ht->ht, hash_val)) {
        if (i++ > ht->size)
            return -1;
        if (!ht->cmp(he, key))
            break;
        hash_val = hash_table_next(ht, hash_val);
    }
    if (he)
        return hash_val;
    return -1;
}

struct hash_table {
    int size;
    struct hash_entry **ht;
    int (*cmp)(void *, void *);
    int (*hash)(void *);
};
```

```c
struct hash_entry {
    void *key;
    void *value;
    struct hash_entry *next;
};
```

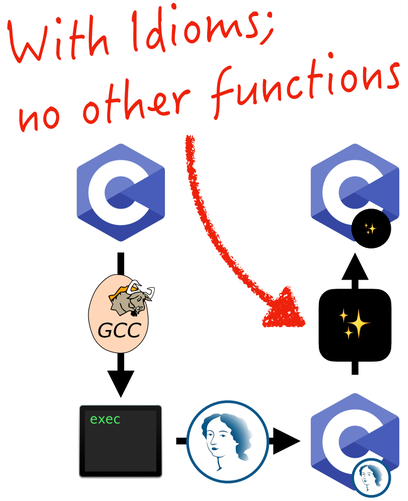
The context of the target function alone may be insufficient to correctly interpret that function.

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}

struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

All data is stored in an array

```
struct gap_array {
    int len;
    void **array;
}
```



Bad Idioms: NO additional context

```
int hash_table_find(struct hash_table *ht, void *key) {
    struct hash_entry *he;
    int hash_val = hash_table_hash(ht, key);
    int i = 0;
    for (he = hash_table_get(ht->ht, hash_val); he;
         he = hash_table_get(ht->ht, hash_val)) {
        if (i++ > ht->size)
            return -1;
        if (!ht->cmp(he, key))
            break;
        hash_val = hash_table_next(ht, hash_val);
    }
    if (he)
        return hash_val;
    return -1;
}

struct hash_table {
    int size;
    struct hash_entry **ht;
    int (*cmp)(void *, void *);
    int (*hash)(void *);
};
```

```
struct hash_entry {
    void *key;
    void *value;
    struct hash_entry *next;
};
```

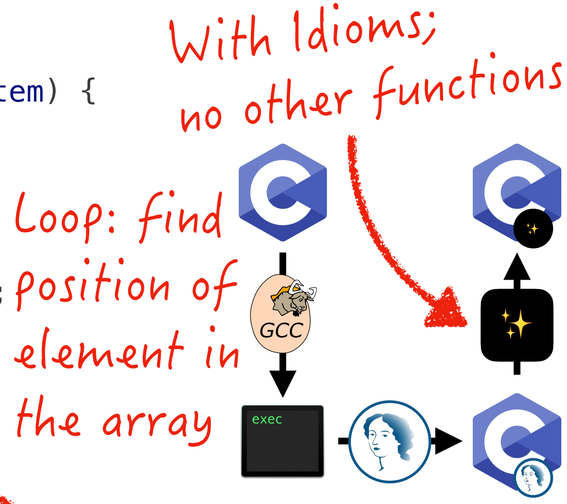
The context of the target function alone may be insufficient to correctly interpret that function.

Original Source Code

```c
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}

struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

All data is stored in an array

```c
struct gap_array {
    int len;
    void **array;
}
```



Bad Idioms: NO additional context

```c
int hash_table_find(struct hash_table *ht, void *key) {
    struct hash_entry *he;
    int hash_val = hash_table_hash(ht, key);
    int i = 0;
    for (he = hash_table_get(ht->ht, hash_val); he;
         he = hash_table_get(ht->ht, hash_val)) {
        if (i++ > ht->size)
            return -1;
        if (!ht->cmp(he, key))
            break;
        hash_val = hash_table_next(ht, hash_val);
    }
    if (he)
        return hash_val;
    return -1;
}

struct hash_table {
    int size;
    struct hash_entry **ht;
    int (*cmp)(void *, void *);
    int (*hash)(void *);
};
```

```c
struct hash_entry {
    void *key;
    void *value;
    struct hash_entry *next;
};
```

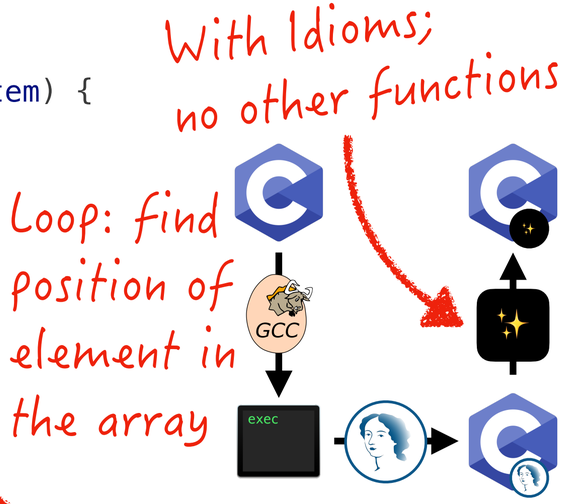
The context of the target function alone may be insufficient to correctly interpret that function.

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}

struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

All data is stored in an array

```
struct gap_array {
    int len;
    void **array;
}
```



Bad Idioms: NO additional context

```
int hash_table_find(struct hash_table *ht, void *key) {
    struct hash_entry *he;
    int hash_val = hash_table_hash(ht, key);
    int i = 0;
    for (he = hash_table_get(ht->ht, hash_val); he;
         he = hash_table_get(ht->ht, hash_val)) {
        if (i++ > ht->size)
            return -1;
        if (!ht->cmp(he, key))
            break;
        hash_val = hash_table_next(ht, hash_val);
    }
    if (he)
        return hash_val;
    return -1;
}

struct hash_table {
    int size;
    struct hash_entry **ht;
    int (*cmp)(void *, void *);
    int (*hash)(void *);
};
```

```
struct hash_entry {
    void *key;
    void *value;
    struct hash_entry *next;
};
```

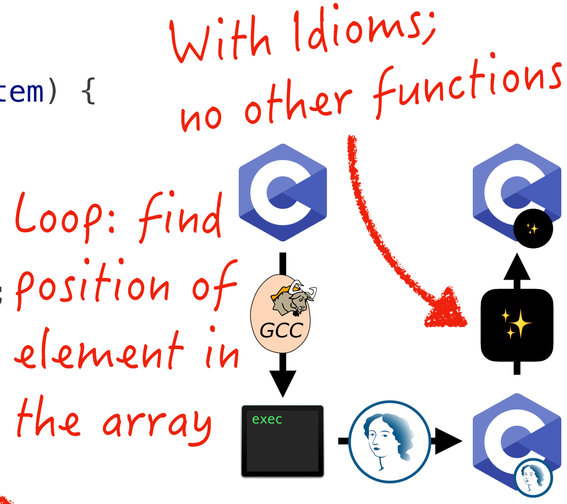
The context of the target function alone may be insufficient to correctly interpret that function.

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}

struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

All data is stored in an array

```
struct gap_array {
    int len;
    void **array;
}
```



Bad Idioms: NO additional context

```
int hash_table_find(struct hash_table *ht, void *key) {
    struct hash_entry *he;
    int hash_val = hash_table_hash(ht, key);
    int i = 0;
    for (he = hash_table_get(ht->ht, hash_val); he;
         he = hash_table_get(ht->ht, hash_val)) {
        if (i++ > ht->size)
            return -1;
        if (!ht->cmp(he, key))
            break;
        hash_val = hash_table_next(ht, hash_val);
    }
    if (he)
        return hash_val;
    return -1;
}

struct hash_table {
    int size;
    struct hash_entry **ht;
    int (*cmp)(void *, void *);
    int (*hash)(void *);
};
```

```
struct hash_entry {
    void *key;
    void *value;
    struct hash_entry *next;
};
```

An array of linked lists

The context of the target function alone may be insufficient to correctly interpret that function.

Original Source Code

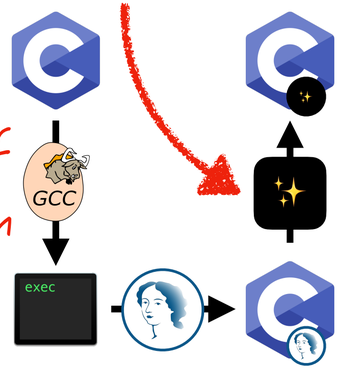
```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}

struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

All data is stored in an array

```
struct gap_array {
    int len;
    void **array;
}
```

With Idioms; no other functions

Loop: find position of element in the array



Bad Idioms: NO additional context

```
int hash_table_find(struct hash_table *ht, void *key) {
    struct hash_entry *he;
    int hash_val = hash_table_hash(ht, key);
    int i = 0;
    for (he = hash_table_get(ht->ht, hash_val); he;
         he = hash_table_get(ht->ht, hash_val)) {
        if (i++ > ht->size)
            return -1;
        if (!ht->cmp(he, key))
            break;
        hash_val = hash_table_next(ht, hash_val);
    }
    if (he)
        return hash_val;
    return -1;
}

struct hash_table {
    int size;
    struct hash_entry **ht;
    int (*cmp)(void *, void *);
    int (*hash)(void *);
};
```

Loop: search the linked list for the element

```
struct hash_entry {
    void *key;
    void *value;
    struct hash_entry *next;
};
```

An array of linked lists

The context of the target function alone may be insufficient to correctly interpret that function.

Original Source Code

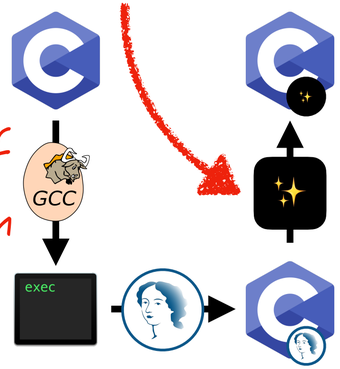
```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}

struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

All data is stored in an array

```
struct gap_array {
    int len;
    void **array;
}
```

With Idioms; no other functions

Loop: find position of element in the array



Bad Idioms: NO additional context

```
int hash_table_find(struct hash_table *ht, void *key) {
    struct hash_entry *he;
    int hash_val = hash_table_hash(ht, key);
    int i = 0;
    for (he = hash_table_get(ht->ht, hash_val); he;
         he = hash_table_get(ht->ht, hash_val)) {
        if (i++ > ht->size)
            return -1;
        if (!ht->cmp(he, key))
            break;
        hash_val = hash_table_next(ht, hash_val);
    }
    if (he)
        return hash_val;
    return -1;
}

struct hash_table {
    int size;
    struct hash_entry **ht;
    int (*cmp)(void *, void *);
    int (*hash)(void *);
};
```

Loop: search the linked list for the element

```
struct hash_entry {
    void *key;
    void *value;
    struct hash_entry *next;
};
```

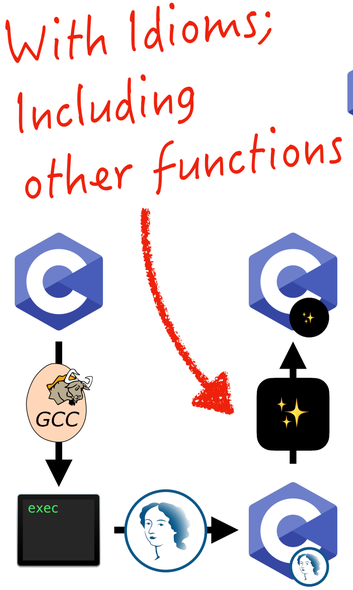
An array of linked lists

The additional context from other functions improves prediction correctness

Original Source Code

```
int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item))
            break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}

struct hash {
    int hash_size;
    int item_cnt;
    struct gap_array *data;
    int (*hash_make_key)(void *item);
    int (*cmp_item)(void *item1, void *item2);
}
```

```
struct gap_array {
    int len;
    void **array;
}
```

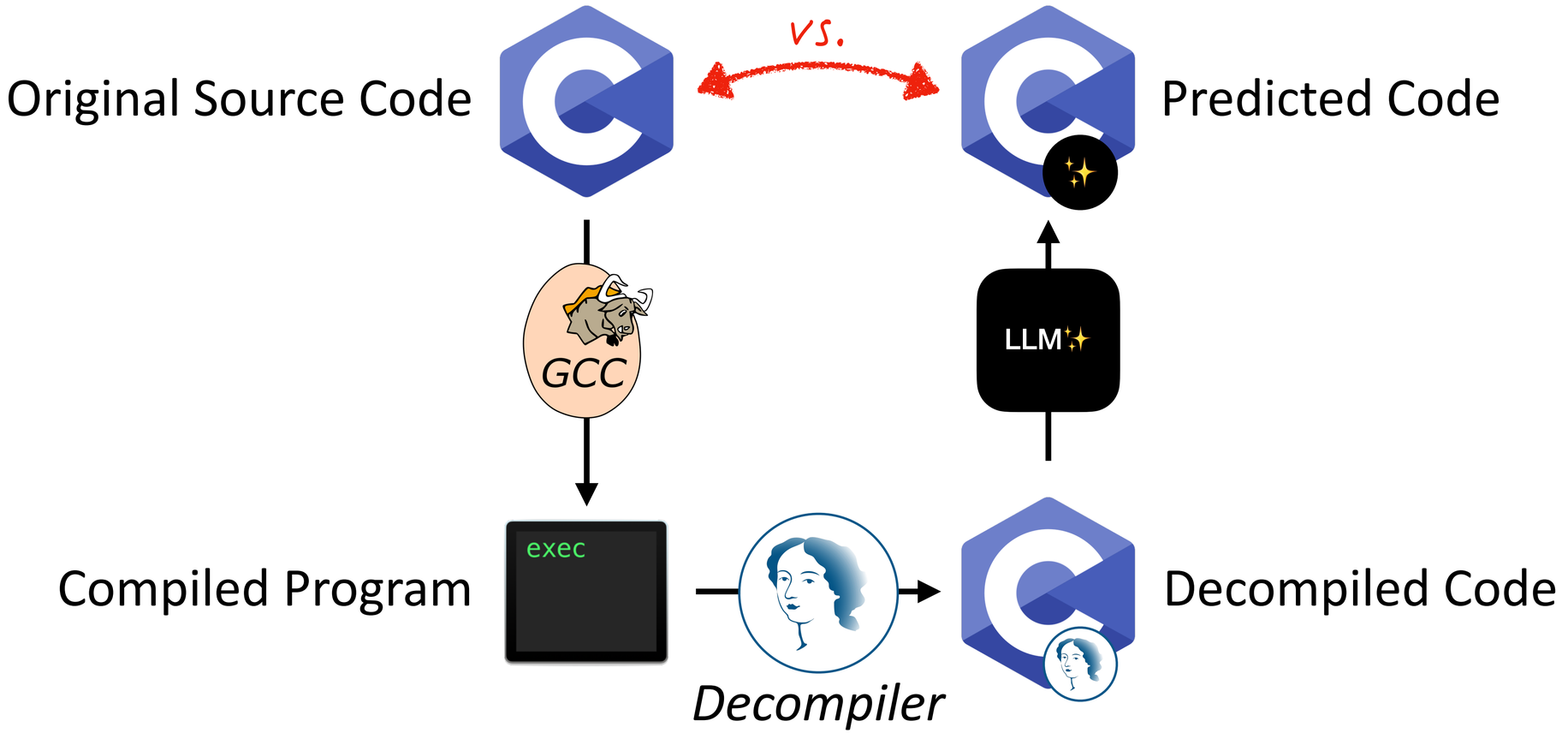


Idioms: YES additional context

```
int hash_find(struct hash_t *hash, void *key)
{
    int index = hash_index(hash, key);
    int i = 0;
    void *item = hash_get(hash->table, index);
    while (item != ((void *)0)) {
        if (i++ > hash->size) {
            return -1;
        }
        if (hash->cmp(item, key) == 0) {
            break;
        }
        index = hash_next(hash, index);
        item = hash_get(hash->table, index);
    }
    return (item == ((void *)0)) ? -1 : index;
}

struct hash_t {
    int size;
    int count;
    struct hash_table_t *table;
    int (*hash)(void *key);
    int (*cmp)(void *key1, void *key2);
};
```

```
struct hash_table_t {
    int size;
    void **items;
};
```

To evaluate, we compare the predicted and original code.



We measure multiple aspects of neural decompiler success.

We measure multiple aspects of neural decompiler success.

Neural decompilers compete with deterministic decompilers

We measure multiple aspects of neural decompiler success.

Neural decompilers compete with deterministic decompilers

- Semantic preservation:

We measure multiple aspects of neural decompiler success.

Neural decompilers compete with deterministic decompilers

- Semantic preservation:
  - Machine learning models can make arbitrary mistakes.

We measure multiple aspects of neural decompiler success.

Neural decompilers compete with deterministic decompilers

- Semantic preservation:
  - Machine learning models can make arbitrary mistakes.
  - *Want to measure*: How frequently are neural decompilers correct?

We measure multiple aspects of neural decompiler success.

Neural decompilers compete with deterministic decompilers

- Semantic preservation:
  - Machine learning models can make arbitrary mistakes.
  - *Want to measure*: How frequently are neural decompilers correct?
- Code improvements

We measure multiple aspects of neural decompiler success.

Neural decompilers compete with deterministic decompilers

- Semantic preservation:
  - Machine learning models can make arbitrary mistakes.
  - *Want to measure*: How frequently are neural decompilers correct?
- Code improvements
  - Decompiled code is hard to read

We measure multiple aspects of neural decompiler success.

Neural decompilers compete with deterministic decompilers

- Semantic preservation:
  - Machine learning models can make arbitrary mistakes.
  - *Want to measure*: How frequently are neural decompilers correct?
- Code improvements
  - Decompiled code is hard to read
  - *Want to measure*: Quality of improvements made to the code.

We built a new benchmark focused on user-defined types (UDTs): `structs` and `unions`

We built a new benchmark focused on user-defined types (UDTs): `structs` and `unions`

Existing Work

We built a new benchmark focused on user-defined types (UDTs): `structs` and `unions`

Existing Work

Benchmark with unit tests

We built a new benchmark focused on user-defined types (UDTs): `structs` and `unions`

Existing Work

Benchmark with unit tests

Very few UDTs

We built a new benchmark focused on user-defined types (UDTs): `structs` and `unions`

Existing Work	*Realtype* (Our dataset)
Benchmark with unit tests	
Very few UDTs	Lots of UDTs

We built a new benchmark focused on user-defined types (UDTs): `structs` and `unions`

Existing Work	*Realtype* (Our dataset)
Benchmark with unit tests	
Difficult to generate meaningful test cases for complex data structures	
Very few UDTs	Lots of UDTs

We built a new benchmark focused on user-defined types (UDTs): `structs` and `unions`

Existing Work	*Realtype* (Our dataset)
Benchmark with unit tests	??
Very few UDTs	Lots of UDTs

Difficult to generate meaningful test cases for complex data structures

Our approach is based on the interaction of operations within a function.

Our approach is based on the interaction of operations within a function.

- A function performs a sequence of operations

Our approach is based on the interaction of operations within a function.

- A function performs a sequence of operations
- Functions which perform the same sequence of operations do the same thing.

Our approach is based on the interaction of operations within a function.

- A function performs a sequence of operations
- Functions which perform the same sequence of operations do the same thing.
- Some flexibility is allowed:

Our approach is based on the interaction of operations within a function.

- A function performs a sequence of operations
- Functions which perform the same sequence of operations do the same thing.
- Some flexibility is allowed:

```
x = x + 1;      =   y = y + 1;
y = y + 1;          x = x + 1;
```

Our approach is based on the interaction of operations within a function.

- A function performs a sequence of operations
- Functions which perform the same sequence of operations do the same thing.
- Some flexibility is allowed:

```
x = x + 1;
y = y + 1;
```
=
```
y = y + 1;
x = x + 1;
```

```
x = x + 1;
y = y + x;
```
≠
```
y = y + x;
x = x + 1;
```

Our approach is based on the interaction of operations within a function.

- A function performs a sequence of operations
- Functions which perform the same sequence of operations do the same thing.
- Some flexibility is allowed:

```
x = x + 1;        =   y = y + 1;
y = y + 1;            x = x + 1;

x = x + 1;        ≠   y = y + x;
y = y + x;            x = x + 1;
```

- Functions which perform the same partially ordered sequence of operations do the same thing, where $a < b$ if b depends on a*

*loops introduce cycles of dependencies; therefore, this is true for a dynamic trace but technically not statically

Our approach is based on the interaction of operations within a function.

- A function performs a sequence of operations
- Functions which perform the same sequence of operations do the same thing.
- Some flexibility is allowed:

```
x = x + 1;        =   y = y + 1;
y = y + 1;            x = x + 1;

x = x + 1;        ≠   y = y + x;
y = y + x;            x = x + 1;
```

- Functions which perform the same partially ordered sequence of operations do the same thing, where $a < b$ if b depends on a*
- Functions are the same if they have isomorphic dependency graphs

*loops introduce cycles of dependencies; therefore, this is true for a dynamic trace but technically not statically

Our approach is based on the interaction of operations within a function.

- A function performs a sequence of operations
- Functions which perform the same sequence of operations do the same thing.
- Some flexibility is allowed:

```
x = x + 1;        =   y = y + 1;
y = y + 1;            x = x + 1;

x = x + 1;        ≠   y = y + x;
y = y + x;            x = x + 1;
```

- Functions which perform the same partially ordered sequence of operations do the same thing, where $a < b$ if b depends on a*
- Functions are the same if they have isomorphic dependency graphs

For more info, check out this paper!

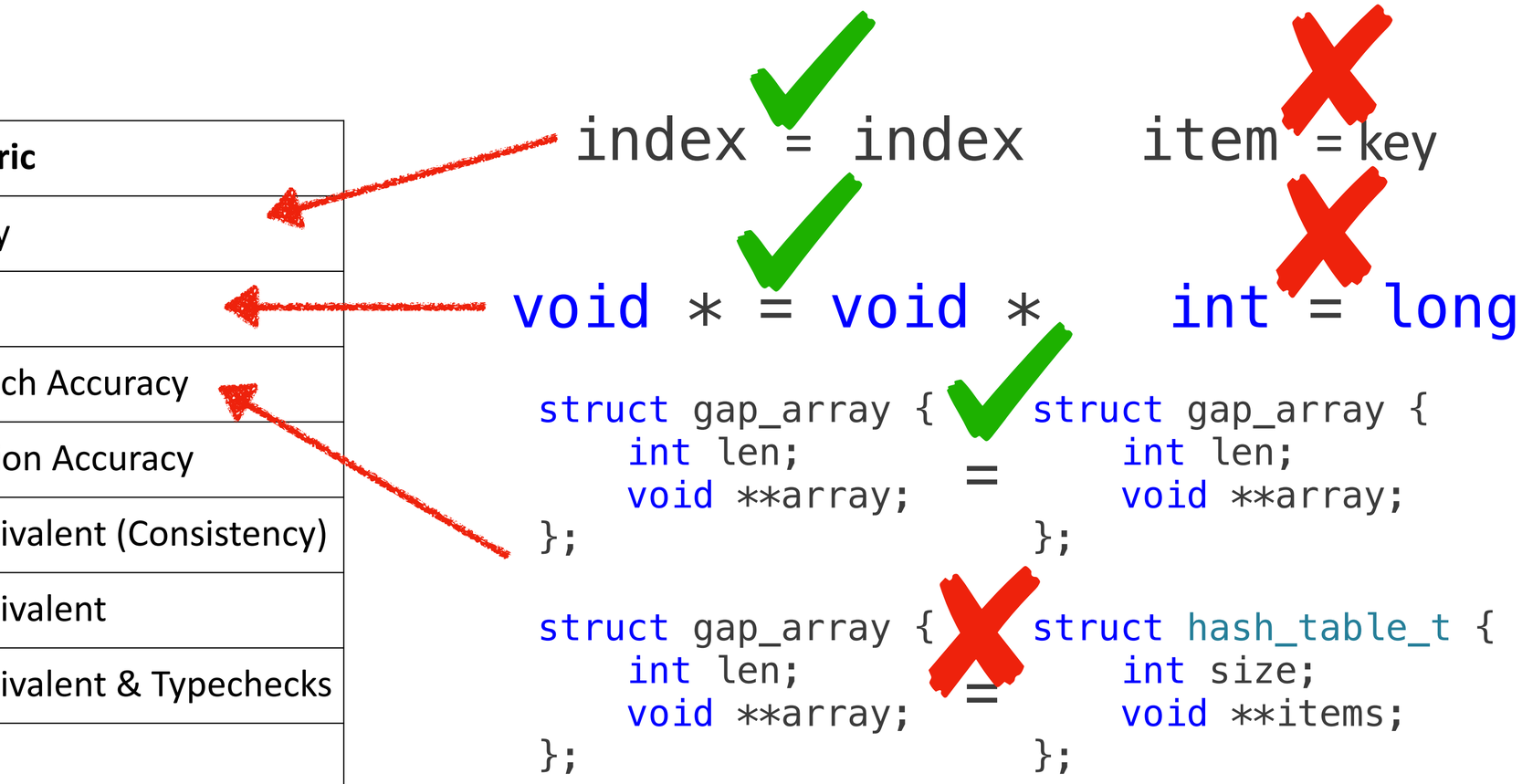
*loops introduce cycles of dependencies; therefore, this is true for a dynamic trace but technically not statically

Our metrics evaluate both correctness and improvements made to the code.

	Metric
Improvements	Variable Name Accuracy
	Variable Type Accuracy
	Variable UDT Exact-Match Accuracy
	Variable UDT Composition Accuracy
Correctness	Dependency-based Equivalent (Consistency)
	Dependency-based Equivalent
	Dependency-based Equivalent & Typechecks
	Passes exebench tests

Our metrics evaluate both correctness and improvements made to the code.

	Metric
Improvements	Variable Name Accuracy
	Variable Type Accuracy
	Variable UDT Exact-Match Accuracy
	Variable UDT Composition Accuracy
Correctness	Dependency-based Equivalent (Consistency)
	Dependency-based Equivalent
	Dependency-based Equivalent & Typechecks
	Passes exebench tests



Our metrics evaluate both correctness and improvements made to the code.

	Metric
Improvements	Variable Name Accuracy
	Variable Type Accuracy
	Variable UDT Exact-Match Accuracy
	Variable UDT Composition Accuracy
Correctness	Dependency-based Equivalent (Consistency)
	Dependency-based Equivalent
	Dependency-based Equivalent & Typechecks
	Passes exebench tests


```
struct gap_array {          =   struct hash_table_t {
    int len;                        int size;
    void **array;                   void **items;
};                              };

struct gap_array {          =   struct hash_table_t {
    int len;                        int size;
    void **array;                   int **items;
};                              };
```

Our metrics evaluate both correctness and improvements made to the code.

	Metric
Improvements	Variable Name Accuracy
	Variable Type Accuracy
	Variable UDT Exact-Match Accuracy
	Variable UDT Composition Accuracy
Correctness	Dependency-based Equivalent (Consistency)
	Dependency-based Equivalent
	Dependency-based Equivalent & Typechecks
	Passes exebench tests

Correct iff original and predicted code's program dependence graphs are isomorphic.

Underapproximation of correctness

Our metrics evaluate both correctness and improvements made to the code.

	Metric
Improvements	Variable Name Accuracy
	Variable Type Accuracy
	Variable UDT Exact-Match Accuracy
	Variable UDT Composition Accuracy
Correctness	Dependency-based Equivalent (Consistency)
	Dependency-based Equivalent
	Dependency-based Equivalent & Typechecks
	Passes exebench tests

Correct iff original and predicted code's program dependence graphs are isomorphic
AND function names are correct

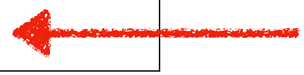
Our metrics evaluate both correctness and improvements made to the code.

	Metric
Improvements	Variable Name Accuracy
	Variable Type Accuracy
	Variable UDT Exact-Match Accuracy
	Variable UDT Composition Accuracy
Correctness	Dependency-based Equivalent (Consistency)
	Dependency-based Equivalent
	Dependency-based Equivalent & Typechecks
	Passes exebench tests

Correct iff original and predicted code's program dependence graphs are isomorphic
AND function names are correct
AND all non-UDT types are correct
AND all UDTs match compositionally.

Our metrics evaluate both correctness and improvements made to the code.

	Metric
Improvements	Variable Name Accuracy
	Variable Type Accuracy
	Variable UDT Exact-Match Accuracy
	Variable UDT Composition Accuracy
Correctness	Dependency-based Equivalent (Consistency)
	Dependency-based Equivalent
	Dependency-based Equivalent & Typechecks
	Passes exebench tests



Only applies to the exebench dataset

Our evaluation shows that our modeling approach offers significant improvement over prior work

		Exebench (Existing Dataset)			Realtype (Our Dataset)		
	Metric	Nova	LLM4Decompile	**Idioms (Ours)**	Nova	LLM4Decompile	**Idioms (Ours)**
Improvements	Variable Name Accuracy	12.9	14.7	**20.6**	4.5	3.4	**19.8**
	Variable Type Accuracy	41.8	45.5	**58.2**	13.0	13.4	**38.3**
	Variable UDT Exact-Match Accuracy	0.0	0.0	**20.7**	0.0	0.0	**6.4**
	Variable UDT Composition Accuracy	0.0	0.0	**34.7**	0.0	0.0	**15.0**
Correctness	Dependency-based Equivalent (Consistency)	24.8	27.9	**34.1**	16.6	10.6	**32.3**
	Dependency-based Equivalent	23.9	27.4	**33.7**	6.7	7.3	**21.6**
	Dependency-based Equivalent & Typechecks	13.4	17.6	**23.9**	0.9	3.3	**9.8**
	Passes exebench tests	37.5	46.3	**54.4**	-	-	-

Our evaluation shows that our modeling approach offers significant improvement over prior work

		Exebench (Existing Dataset)			Realtype (Our Dataset)		
	Metric	Nova	LLM4Decompile	**Idioms (Ours)**	Nova	LLM4Decompile	**Idioms (Ours)**
Improvements	Variable Name Accuracy	12.9	14.7	**20.6**	4.5	3.4	**19.8**
	Variable Type Accuracy	41.8	45.5	**58.2**	13.0	13.4	**38.3**
	Variable UDT Exact-Match Accuracy	0.0	0.0	**20.7**	0.0	0.0	**6.4**
	Variable UDT Composition Accuracy	0.0	0.0	**34.7**	0.0	0.0	**15.0**
Correctness	Dependency-based Equivalent (Consistency)	24.8	27.9	**34.1**	16.6	10.6	**32.3**
	Dependency-based Equivalent	23.9	27.4	**33.7**	6.7	7.3	**21.6**
	Dependency-based Equivalent & Typechecks	13.4	17.6	**23.9**	0.9	3.3	**9.8**
	Passes exebench tests	37.5	46.3	**54.4**	-	-	-

Our evaluation shows that our modeling approach offers significant improvement over prior work

		Exebench (Existing Dataset)			Realtype (Our Dataset)		
	Metric	Nova	LLM4Decompile	**Idioms (Ours)**	Nova	LLM4Decompile	**Idioms (Ours)**
Improvements	Variable Name Accuracy	12.9	14.7	**20.6**	4.5	3.4	**19.8**
	Variable Type Accuracy	41.8	45.5	**58.2**	13.0	13.4	**38.3**
	Variable UDT Exact-Match Accuracy	0.0	0.0	**20.7**	0.0	0.0	**6.4**
	Variable UDT Composition Accuracy	0.0	0.0	**34.7**	0.0	0.0	**15.0**
Correctness	Dependency-based Equivalent (Consistency)	24.8	27.9	**34.1**	16.6	10.6	**32.3**
	Dependency-based Equivalent	23.9	27.4	**33.7**	6.7	7.3	**21.6**
	Dependency-based Equivalent & Typechecks	13.4	17.6	**23.9**	0.9	3.3	**9.8**
	Passes exebench tests	37.5	46.3	**54.4**	-	-	-

Idioms full paper

I’m on the industry job market!

My website