Symbolic Transformations Assignment

Date : 28 – 09 - 2025

Raj Shekhar - 240545002004

Objective : Create and analyze the 5 symbolic constructs in a C code and compare their assembly output For -O0 and -O2 optimization levels. In the generated assembly code, there are noticeable insights into the compiler's optimization process and the practical differences between these constructs.

```c
#include <stdio.h>
#define DEFINE_CONST 20
#define DEFINE_MACRO(x) ((x) * 2)

const int CONST_INT = 45;
enum Status
{
    PERCENTAGE_ERROR = -4,
    STATUS_OK = 200
};
static const int STATIC_CONST_INT = 90;

int main()
{
    int var1 = DEFINE_CONST;
    int var2 = DEFINE_MACRO(var1);
    int var3 = CONST_INT;
    int var4 = STATUS_OK;
    int var5 = STATIC_CONST_INT;
    int result = var1 + var2 + var3 + var4 + var5 - PERCENTAGE_ERROR;
    // Expected: 10 + (10 * 2) + 40 + 200 + 40 - (-4) = 399
    printf("Result: %d\n", result);
    return 0;
}
```

Fig1: Source Code

Introduction : Symbolic execution (or symbolic transformation) in reverse engineering is a technique used to analyze a program with its inputs as symbolic values. Instead of executing the program with actual data, symbolic execution explores all possible execution paths by replacing program inputs with symbols.

This allows analysts to identify the specific inputs that trigger different code paths, understand complex behavior in obfuscated code, and bypass anti-analysis techniques.

Assembly Generation :

1. No Optimization (-O0) : *gcc -S -O0 -o symbolic_O0.s symbolic_tranformation.c*

This provides a direct, literal translation of the C code, making it easy to trace the logic.

2. Level 2 Optimization (-O2) : *gcc -S -O2 -o symbolic_O2.s symbolic_transformation.c*

This is a common optimization level that instructs the compiler to aggressively optimize for performance.

Symbolic Constructs Analysis :

1. Macro Constants (#define DEFINE_CONST 20)

int var1 = DEFINE_CONST
For -O0:

```
movl    $20, -4(%rbp)
```

- Direct substitution with literal value
- No symbol table entry

Optimized away, Pre-computed result
For -O2:

```
movl    $399, %edx
```

- Completely eliminated through constant propagation
- Value folded into final result

2. Function-like Macros (#define DEFINE_MACRO(x) ((x) * 2))

x * 2 via addition, & Store var2
For -O0:

```
movl    -4(%rbp), %eax
addl    %eax, %eax
movl    %eax, -8(%rbp)
```

- Macro expanded inline
- Arithmetic operation performed at runtime

For -O2:
- Completely eliminated - value computed at compile time
- No runtime computation

3. Const Variables (const int CONST_INT = 45)

var3 = CONST_INT
For -O0:

```
movl    $45, -12(%rbp)
```

- Treated like regular variable initialization
- Memory allocation and assignment

For -O2:
- Eliminated through constant folding
- Value incorporated into final computation

4. Enum Constants (STATUS_OK = 200)

var4 = STATUS_OK
For -O0:

```
movl    $200, -16(%rbp)
```

- Enum treated as integer literal
- No type safety in assembly

For -O2:
- Constant folded away
- Enum distinction lost in optimization

5. Static const Variables (static const int STATIC_CONST_INT = 90)

var5 = STATIC_CONST_INT
For -O0:

```
movl    $90, -20(%rbp)
```

- Similar to regular const but with static storage
- Local scope in translation unit

For -O2:
- Eliminated like other constants

-O0 Output (Simplified):

```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $20, -4(%rbp)        # var1 = DEFINE_CONST
    movl    -4(%rbp), %eax
    addl    %eax, %eax           # var2 = DEFINE_MACRO(var1)
    movl    %eax, -8(%rbp)
    movl    $45, -12(%rbp)       # var3 = CONST_INT
    movl    $200, -16(%rbp)      # var4 = STATUS_OK
    movl    $90, -20(%rbp)       # var5 = STATIC_CONST_INT
    # ... computation of result ...
    movl    $399, %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
```

-O2 Output (Simplified):

```
main:
    subq    $8, %rsp
    movl    $399, %esi          # Pre-computed result
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, %eax
    addq    $8, %rsp
    ret
```

Key Differences :

1. Constant Folding : -O2 pre-computes the entire expression :
   20 + 40 + 45 + 200 + 90 - (-4) = 399

2. Dead Code Elimination : All intermediate variables are removed

3. Stack Usage : -O0 uses 24+ bytes for variables. -O2 uses minimal stack.

4. Symbol Resolution :
- Macros: Pure text substitution (preprocessor)
- Const / Enum: Treated as constants (compiler)
- All resolved at compile time, no runtime symbol lookup

5. Performance Impact : -O2 eliminates all memory operations and arithmetic, reducing to a single constant load.

Conclusion :

Therefore, the symbolic constructs behave identically from an optimization perspective and they all enable constant folding when their values are known at compile time, allowing the optimizer to compute the final result without any runtime computation.

As a result, the above analysis reveals a key principle of modern compilers : the distinction between symbolic constructs is often more for the benefit of the programmer (improving readability, type safety, and scope management) than for the final performance of the program.
