Investigating Constant Multiplications with Debugging: An Analysis of Compiler Optimization and Integer Overflow Handling

Jiwon Hwang *Cybersecurity*
University of Maryland, College Park
jhwang97@umd.edu

***Abstract*—This paper investigates how modern compilers handle constant multiplication operations, specifically multiplication by 12, across different integer types and system architectures. We analyze the compilation process using debugging tools on both 32-bit and 64-bit systems, examining cases involving overflow conditions and various integer sizes. Through systematic testing and assembly-level analysis, we demonstrate significant differences in overflow handling between architectures and reveal compiler optimization strategies for constant multiplication. Our findings highlight the critical importance of understanding data type limitations and compiler behavior when dealing with large integer operations, particularly in security-sensitive applications where integer overflow vulnerabilities can lead to exploitable conditions.**

I. Introduction

A. Running environment

As running environment, two different systems were used for comparison.

- System A
  - OS: Ubuntu 16.04.1 LTS
  - Architecture: i686 (32-bit)
  - Compiler: gcc v5.4.0
  - Debugger: gdb v7.11.1
- System B
  - OS: Ubuntu 24.04.1 LTS
  - Architecture: x86_64(64-bit)
  - Compiler: gcc v13.2.0
  - Debugger: gdb v15.0.50

B. Purpose

The purpose of this report is to analyze code to gain a deeper understanding of how the compiler handles:

- Integer sizes and data type limitations
- Multiplication optimization strategies
- Integer overflow behavior across different architectures

to perform multiplication by a constant value, such as 12, and understand the security implications of overflow handling differences.

II. Methodology

A. Overview

A given code (See Appendix A) implements a function to multiply input number by 12. Testing various cases and analyzing the debugging process of these results will bring deeper understanding of how the compiler handles different data types and sizes and calculate the output step-by-step. The analysis focuses on assembly-level examination of compiler optimization strategies and overflow behavior.

B. Test Cases

Table I
Test Case Categories

	Unsigned integer	**Signed integer**
Normal sized	Case 1	Case 4
Over sized	Case 2	Case 5
Powers of Two	Case 3	Case 6

1) Six primary cases were considered based on existence of sign and size of integer values:
    a) Case 1: Unsigned, normal-sized integer (1)
    b) Case 2: Unsigned, over-sized integer (12345678910)
    c) Case 3: Unsigned, power of two integer (2)
    d) Case 4: Signed, normal-sized integer (-1)
    e) Case 5: Signed, over-sized integer (-12345678910)
    f) Case 6: Signed, power of two integer (-2)
2) Special cases were added for various types and scenarios:
    a) Case 7: Zero (0)
    b) Case 8: Very large constant (12345678901234567890)

C. Analysis Process

1) Compilation Analysis: The test program was compiled with debugging symbols and no optimization to preserve the multiplication process:

- Compilation flags: -g -O0
- Assembly code generation for analysis
- Register and memory layout examination

2) Debugging Methodology: Step-by-step debugging was performed using GDB with the following approach:

- Breakpoint placement at mulBy12 function
- Register state inspection at each assembly instruction
- Bit-level analysis of intermediate computations
- Overflow condition tracking

3) Cross-platform Comparison: Identical test cases were executed on both 32-bit and 64-bit systems to identify:

- Architectural differences in overflow handling
- Register usage patterns
- Intermediate computation differences

III. Results

As the following tables show, several errors which are different from expected output were detected. There were also significant gaps between the outputs of System A and B, particularly in overflow cases.

Table II
TEST CASES 1,2,3 : UNSIGNED INTEGERS

Case	1	2	3
Input	1	12345678910	2
Output			
Expected	12	148148146920	24
(Sys A) Actual	12	**-12**	24
(Sys B) Actual	12	**2119258856**	24

Table III
TEST CASES 4,5,6 : SIGNED INTEGERS

Case	4	5	6
Input	-1	-12345678910	-2
Output			
Expected	-12	-148148146920	-24
(Sys A) Actual	-12	**0**	-24
(Sys B) Actual	-12	**-2119258856**	-24

Table IV
TEST CASES 7,8 : SPECIAL CASES

Case	7	8
Input	0	12345678901234567890
Output		
Expected	0	148148146814814814680
(Sys A) Actual	0	**-12**
(Sys B) Actual	0	**-12**

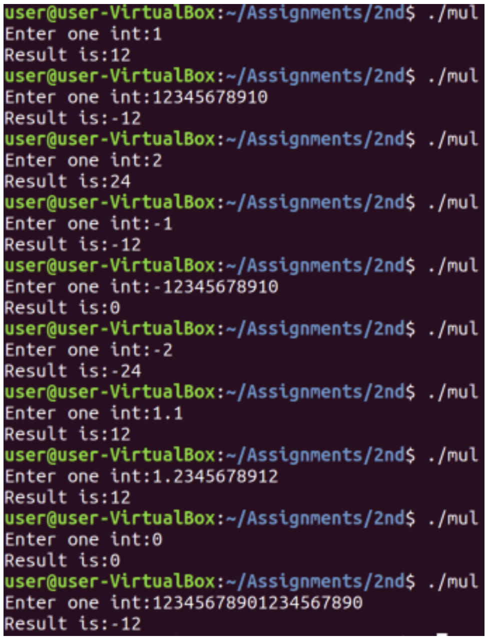
A. Error Cases

Among those test cases, there were several critical overflow errors:

1) Case 2: *-12 (System A) and 2119258856 (System B)* for *Unsigned, Over-sized integer*
2) Case 5: *0 (System A) and -2119258856 (System B)* for *Signed, Over-sized integer*
3) Case 8: *-12* for *Very large integer* on both systems

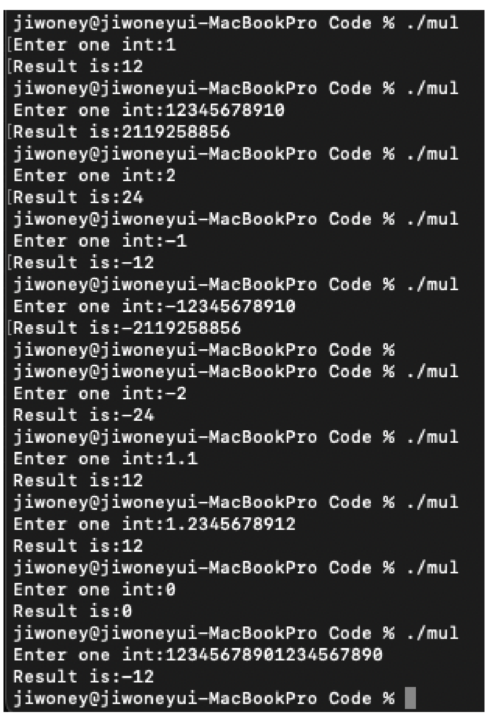
These errors demonstrate significant architectural differences in overflow handling and highlight potential security vulnerabilities.

B. System Output Comparison



```
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:1
Result is:12
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:12345678910
Result is:-12
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:2
Result is:24
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:-1
Result is:-12
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:-12345678910
Result is:0
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:-2
Result is:-24
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:1.1
Result is:12
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:1.2345678912
Result is:12
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:0
Result is:0
user@user-VirtualBox:~/Assignments/2nd$ ./mul
Enter one int:12345678901234567890
Result is:-12
```

Figure 1. Result of Appendix A in System A



```
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:1
Result is:12
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:12345678910
Result is:2119258856
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:2
Result is:24
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:-1
Result is:-12
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:-12345678910
Result is:-2119258856
jiwoney@jiwoneyui-MacBookPro Code %
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:-2
Result is:-24
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:1.1
Result is:12
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:1.2345678912
Result is:12
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:0
Result is:0
jiwoney@jiwoneyui-MacBookPro Code % ./mul
Enter one int:12345678901234567890
Result is:-12
jiwoney@jiwoneyui-MacBookPro Code %
```

Figure 2. Result of Appendix A in System B

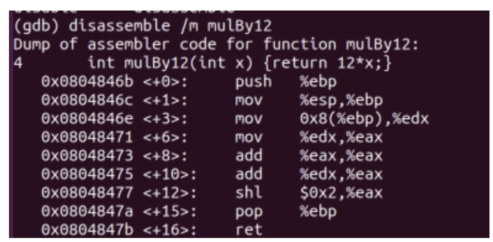
IV. Discussion

A. Multiplication Optimization Process

Breaking down the multiplication process into detailed steps with intermediate results provides better insight regarding register interaction and compiler optimization strategies, especially focusing on the mulBy12 function implementation.

The compiler implements multiplication by 12 using the mathematical identity: $12x = 4 \times 3x$, which translates to the following assembly operations:

1) **Step 1: Setup Stack Frame** - Establishes the function's stack frame
2) **Step 2: Load Parameter into Register** - The parameter x is loaded into the edx register
3) **Step 3: Copy Parameter to eax** - The value of x is copied into eax register
4) **Step 4: Double the Value** - Adds eax to itself, effectively computing 2x
5) **Step 5: Add Original Value** - Adds the original x value, resulting in 3x
6) **Step 6: Shift Left by 2 bits** - Multiplies by 4, achieving the final result of 12x
7) **Step 7: Restore and Return** - Restores stack frame and returns the result



```
(gdb) disassemble /m mulBy12
Dump of assembler code for function mulBy12:
4	    int mulBy12(int x) {return 12*x;}
   0x0804846b <+0>:	push   %ebp
   0x0804846c <+1>:	mov    %esp,%ebp
   0x0804846e <+3>:	mov    0x8(%ebp),%edx
   0x08048471 <+6>:	mov    %edx,%eax
   0x08048473 <+8>:	add    %eax,%eax
   0x08048475 <+10>:	add    %edx,%eax
   0x08048477 <+12>:	shl    $0x2,%eax
   0x0804847a <+15>:	pop    %ebp
   0x0804847b <+16>:	ret
```

Figure 3. General Process of Appendix A

B. Detailed Overflow Analysis

1) **Case 2 Analysis** - Unsigned Over-sized Integer (12345678910)

   System A (32-bit):

   a) Step 3: eax = 0x7FFFFFFF (input truncated to INT_MAX due to 32-bit limitation)
   b) Step 4: eax = 0x7FFFFFFF + 0x7FFFFFFF = 0xFFFFFFFE (overflow to -2)
   c) Step 5: eax = 0xFFFFFFFE + 0x7FFFFFFF = 0x17FFFFFFD → 0x7FFFFFFD (32-bit overflow)
   d) Step 6: eax = 0x7FFFFFFD « 2 = 0x1FFFFFFF4 → 0xFFFFFFF4 (-12 in two's complement)

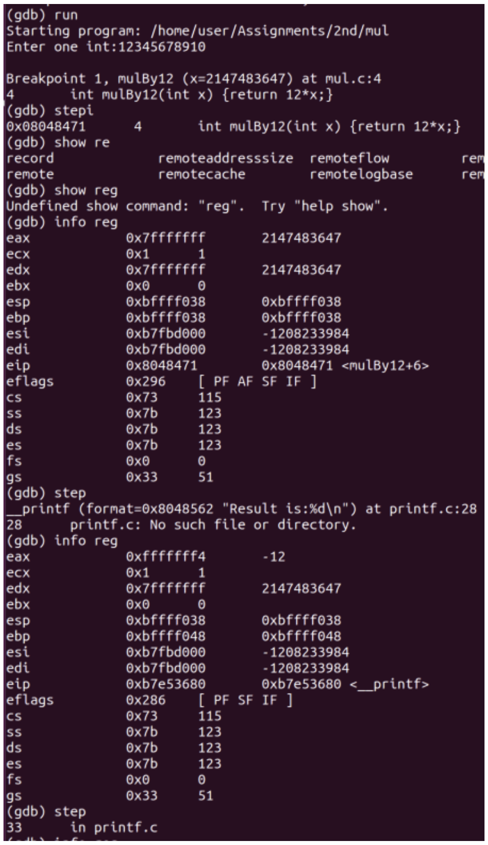


Figure 4. Case 2 Result of Appendix A in System A

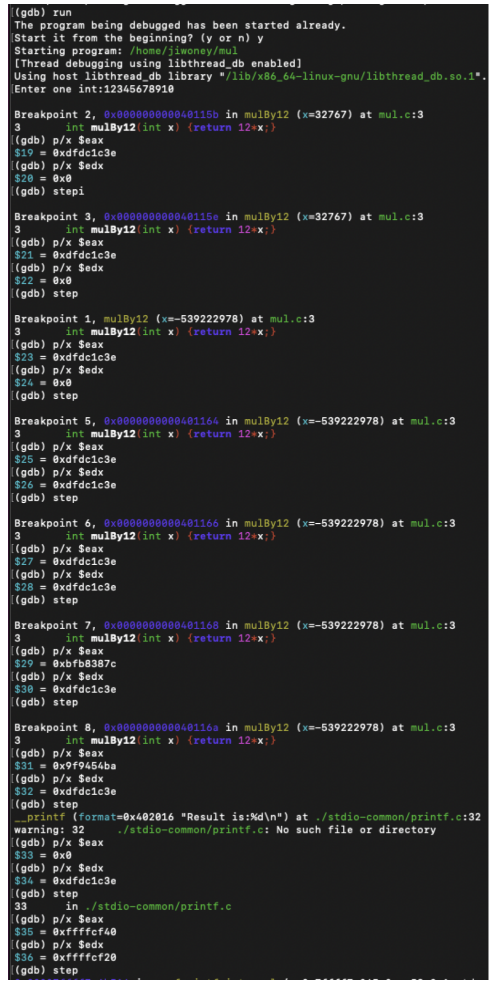


Figure 5. Case 2 Result of Appendix A in System B

System B (64-bit):

a) Step 3: eax = 0xDFDC1C3E (different truncation due to 64-bit processing)
b) Step 4: eax = 0xDFDC1C3E + 0xDFDC1C3E = 0x1BFB8387C → 0xBFB8387C (64-bit overflow)
c) Step 5: eax = 0xBFB8387C + 0xDFDC1C3E = 0x19F9454BA → 0x9F9454BA
d) Step 6: eax = 0x9F9454BA « 2 = 0x7E5152E8 = 2119258856

2) **Case 5 Analysis** - Signed Over-sized Integer (-12345678910)

System A (32-bit):

a) Step 3: eax = 0x80000000 (INT_MIN = -2147483648)
b) Step 4: eax = 0x80000000 + 0x80000000 = 0x100000000 → 0x00000000 (overflow to zero)
c) Steps 5-6: All subsequent operations on zero result in zero

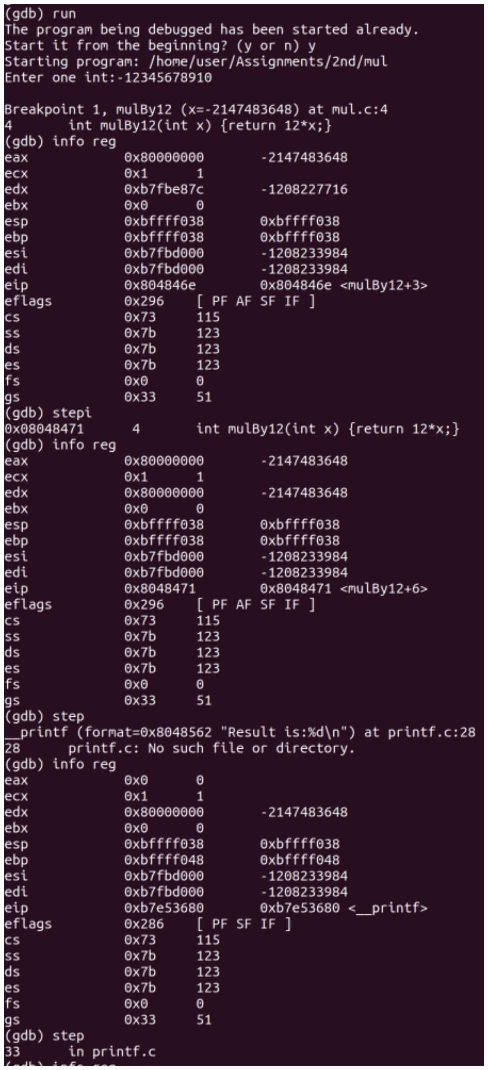


Figure 6. Case 5 Result of Appendix A in System A

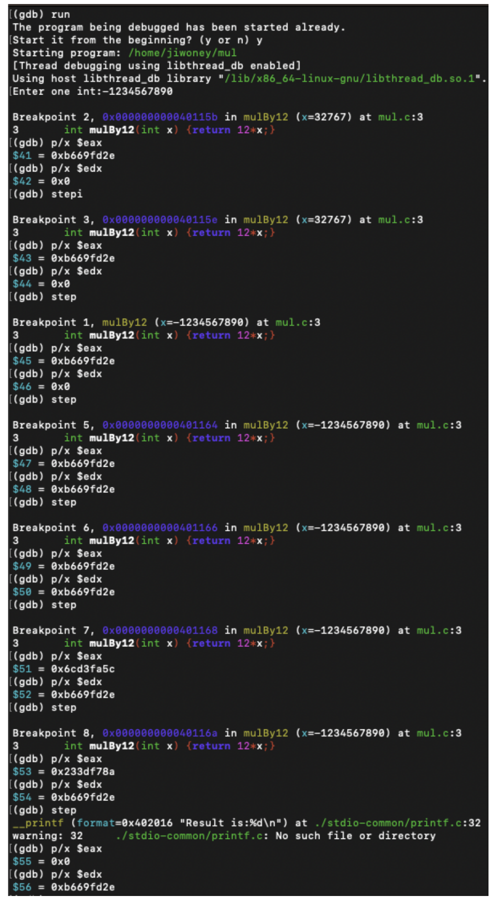


Figure 7. Case 5 Result of Appendix A in System B

System B (64-bit):

a) Step 3: eax = 0xB669FD2E (different handling of negative overflow)
b) Step 4: eax = 0xB669FD2E + 0xB669FD2E = 0x16CD3FA5C → 0x6CD3FA5C
c) Step 5: eax = 0x6CD3FA5C + 0xB669FD2E = 0x1233DF78A → 0x233DF78A
d) Step 6: eax = 0x233DF78A « 2 = 0x8CF7E5E28 → 0x8CF7E5E28 (-2119258856)

3) **Case 8 Analysis** - Very Large Constant
Both systems exhibit identical behavior for this extreme case, with the input being truncated to 0xFFFFFFFF and following the same computational path as Case 2, resulting in -12.

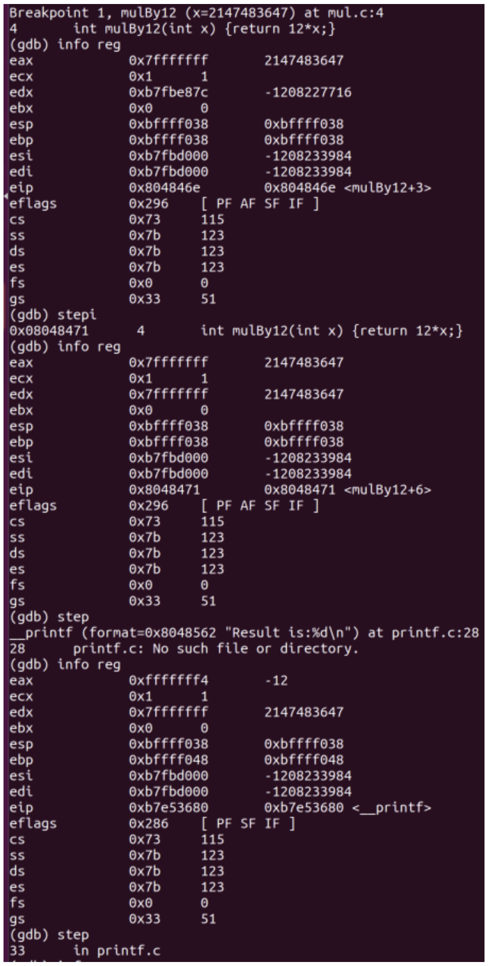


Figure 8. Case 8 Result of Appendix A in System A

Figure 9. General process of Appendix A in System B

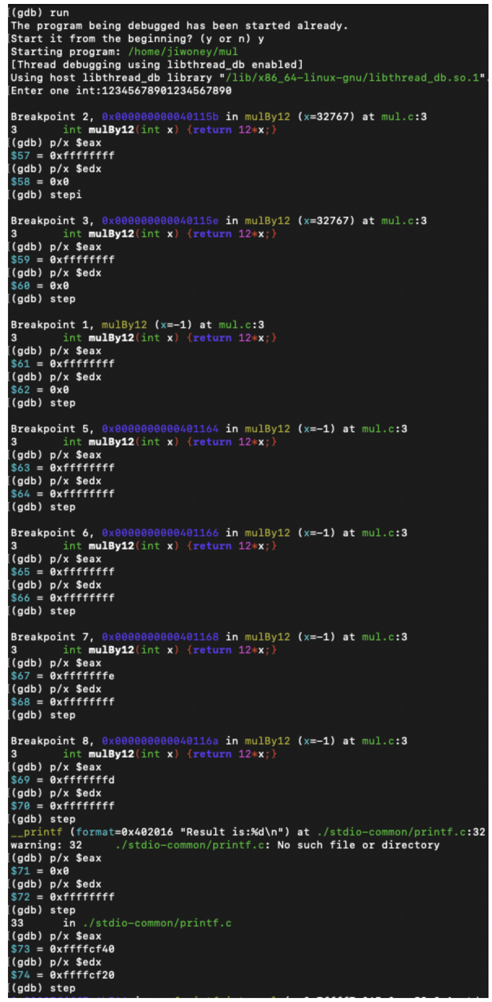


Figure 10. Case 8 Result of Appendix A in System B

C. Architectural Differences and Security Implications

The analysis reveals several critical differences between 32-bit and 64-bit systems:

1) **Register Size Impact:**
   - 32-bit systems: Use 32-bit registers (eax, edx) with stack-based parameter passing
   - 64-bit systems: Utilize 64-bit registers with register-based parameter passing, allowing for extended precision in intermediate calculations
2) **Overflow Behavior:**
   - 32-bit: Overflow occurs during intermediate steps, affecting final results
   - 64-bit: Extended precision delays overflow until final truncation
3) **Truncation Patterns:**
   - Over-sized inputs are truncated differently between architectures

- 32-bit systems truncate to maximum/minimum integer values
- 64-bit systems preserve more bits during intermediate computations

4) **Security Considerations:**
   - Architecture-dependent overflow behavior can lead to inconsistent security properties
   - Silent integer overflows may mask potential vulnerabilities
   - Cross-platform software may exhibit different behavior patterns

D. Compiler Optimization Analysis

The compiler's choice to implement multiplication by 12 as $(3x) \times 4$ rather than alternative decompositions (e.g., $8x + 4x$ or $16x - 4x$) has significant implications:

- Current strategy minimizes the number of operations
- Overflow timing is affected by the specific decomposition used
- Different optimization levels may produce different overflow characteristics

E. Limitations and Future Work

This study has several acknowledged limitations:

1) Analysis limited to multiplication by constant 12
2) Single compiler family (GCC) tested
3) No variation in optimization flags
4) Focus on integer overflow without considering floating-point arithmetic

Future research should investigate multiple constants, various compiler implementations, and different optimization levels to provide comprehensive understanding of these phenomena.

V. Conclusion

This investigation demonstrates that integer overflow behavior in constant multiplication operations varies significantly between system architectures. Key findings include:

1) Compiler optimization strategies for constant multiplication can interact with overflow conditions in architecture-dependent ways
2) 32-bit and 64-bit systems handle intermediate computations differently, leading to distinct overflow patterns
3) Understanding these architectural differences is crucial for developing secure, portable software
4) Silent integer overflows represent a significant security concern that requires careful consideration in cross-platform development

To mitigate potential issues, developers should:

- Use appropriate data types for expected value ranges
- Implement explicit overflow checking where necessary
- Consider using libraries designed for arbitrary precision arithmetic when dealing with large numbers
- Test software across multiple architectures to identify inconsistent behavior

The choice of data types and understanding of compiler behavior are critical when dealing with operations that may exceed standard type bounds, particularly in security-sensitive applications where integer overflow vulnerabilities can lead to exploitable conditions.

VI. Appendices

A. Appendix A: mul.c

```
/* Program to demonstrate arithmetic right
    shift */
//gcc -m32 -g mul.c -o mul -fno-stack-protector
     -no-pie -mpreferred-stack-boundary=2 -fno-
    pic -z execstack
#include <stdio.h>
int mulBy12(int x) {return 12*x;}
int main(){
int a;
printf("Enter one int:");
scanf("%d", &a);
printf("Result is:%d\n", mulBy12(a));
return 0;
}
```

References

[1] HackTricks, Introduction to x64, https://book.hacktricks.xyz/macos-hardening/macos-security-and-privilege-escalation/macos-apps-inspecting-debugging-and-fuzzing/introduction-to-x64