Secure Coding Lab 14 - Integers using Frama-C

INT30-C. Ensure that unsigned integer operations do not wrap

According to the C Standard, arithmetic operations involving unsigned integers can lead to wrapping, meaning that if a result exceeds the maximum value representable by the type, it wraps around using modulo arithmetic. This behavior can result in vulnerabilities, especially in security-critical code.

Key Points

1. ***Wrapping Behavior:****Unsigned integers wrap around when they exceed their max- imum value.*
2. ***Types of Operations:****Addition, subtraction, and multiplication are common op- erations that can lead to wrapping.*
3. ***Precondition and Postcondition Tests:****Implementing checks before and after operations to prevent wrapping.*

Addition of Unsigned Integers

Complaint Code - (Precondition Test)

- *This solution checks for potential wrapping before performing addition.*

Source Code 1: Addition C Code

```
#include <stdio.h>
#include <limits.h>

void func(int si_a, int si_b) {
    unsigned int isum;
    if ((si_a > 0 && si_b > INT_MAX - si_a) || (si_a < 0 && si_b < INT_MIN - si_a)) {
        printf("Error:uAdditionuwoulduoverflow.\n");
        return;
    } else {
        isum = si_a + si_b;
    }
    printf("Sum:u%d\n", isum);
}
```

```c
int main() {
    func(4000000000, 1000000000); // No wrap, handles error
    return 0;
}
```

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[variadic:typing] usig.c:12: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned int to int.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  isum ∈ {705032704}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 12 statements reached (out of 19): 63% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 1

Source Code 2: Addition C Code

```c
#include <stdio.h>
#include <limits.h>

void func(int si_a, int si_b) {
    int isum;
    if ((si_a > 0 && si_b > INT_MAX - si_a) || (si_a < 0 && si_b < INT_MIN - si_a)) {
        printf("Error:uAdditionuwoulduoverflow.\n");
        return;
    } else {
        isum = si_a + si_b;
    }
    printf("Sum:u%d\n", isum);
}

int main() {
    func(4000000000, 1000000000); // No wrap, handles error
    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  isum ∈ {705032704}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 12 statements reached (out of 19): 63% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 2

Subtraction of Unsigned Integers

Complaint Code - (Precondition Test)

- *This solution checks for potential wrapping before performing subtraction.*

Source Code 3: Subtraction C Code

```c
#include <stdio.h>
#include <limits.h>

void func(int si_a, int si_b) {
    unsigned int sdiff;
    if (si_a < si_b) {
        printf("Error:uSubtractionuwoulduunderflow.\n");
        return;
    } else {
        sdiff = si_a - si_b;
    }
    printf("Difference:u%d\n", sdiff);
}

int main() {
    func(10, 20); // Handles error
    return 0;
}
```

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[variadic:typing] usig.c:12: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned int to int.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_1
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 8 statements reached (out of 10): 80% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 3

Source Code 4: Subtraction C Code

```c
#include <stdio.h>
#include <limits.h>

void func(int si_a, int si_b) {
    int sdiff;
    if (si_a < si_b) {
        printf("Error:uSubtractionuwoulduunderflow.\n");
        return;
    } else {
        sdiff = si_a - si_b;
    }
    printf("Difference:u%d\n", sdiff);
}

int main() {
    func(10, 20); // Handles error
    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_1
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 8 statements reached (out of 10): 80% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 4

Multiplication of Unsigned Integers

Complaint Code - (Precondition Test)

- *This solution checks for potential wrapping before performing multiplication.*

Source Code 5: Multiplication C Code

```c
#include <stdio.h>
#include <limits.h>

void multiply(int si_a, int si_b) {
    if (si_a > 0 && si_b > INT_MAX / si_a) {
        printf("Error:uMultiplicationuwoulduoverflow.\n");
        return;
    } else if (si_a < 0 && si_b < INT_MIN / si_a) {
        printf("Error:uMultiplicationuwouldduunderflow.\n");
        return;
    }

    unsigned int product = si_a * si_b;
    printf("Product:u%d\n", product);
}

int main() {
    multiply(4000000000, 2); // Will handle overflow
    return 0;
}
```

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[variadic:typing] usig.c:14: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned int to int.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function multiply:
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 11 statements reached (out of 18): 61% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 5

Source Code 6: Multiplication C Code

```c
#include <stdio.h>
#include <limits.h>

void multiply(int si_a, int si_b) {
    if (si_a > 0 && si_b > INT_MAX / si_a) {
        printf("Error:uMultiplicationuwoulduoverflow.\n");
        return;
    } else if (si_a < 0 && si_b < INT_MIN / si_a) {
        printf("Error:uMultiplicationuwoulduunderflow.\n");
        return;
    }

    int product = si_a * si_b;
    printf("Product:u%d\n", product);
}

int main() {
    multiply(4000000000, 2); // Will handle overflow
    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function multiply:
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 11 statements reached (out of 18): 61% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 6

INT31-C. Ensure that integer conversions do not re- sult in lost or misinterpreted data

Integer conversions in C can lead to issues such as truncation, sign errors, and unex- pected behavior when handling different integer types. This assignment explores these concepts through examples of both noncompliant and compliant code, illustrating the potential pitfalls and safe practices.

The C Standard specifies how integer conversions should be handled. Errors can occur when:

- *Converting from a larger integer type to a smaller one, leading to truncation.*
- *Converting from signed to unsigned types, potentially misinterpreting negative val- ues.*
- *Using unsafe functions like memset with inappropriate arguments.*

Compliant Code - (Unsigned to Signed)

- *Before casting, the code checks if u_a is within the valid range of signed char. This prevents unsafe conversions by validating the range.*

Source Code 7: Unsigned to Signed C Code

```
#include <limits.h>
#include <stdio.h>

void compliant_func(void) {
    long int a = LONG_MAX;   // Change unsigned long to signed long
    signed char sc;

    // Check if 'a' is within the range of signed char
    if (a <= SCHAR_MAX && a >= SCHAR_MIN) {
        sc = (signed char)a; // Safe conversion within range
        printf("Compliantusigneducharuvalue:u%d\n", sc);
    } else {
        printf("Error:u Valueutooulargeuorutoousmalluforusigneduchar.\n");
    }
}

int main(void) {
    compliant_func();
    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function compliant_func:
  a ∈ {9223372036854775807}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 8 statements reached (out of 12): 66% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 7

Compliant Code - Handling time_t Return Values

- *To ensure the comparison is valid, the return value of time() should be compared against -1 cast to the time_t type*
- *This compliant solution ensures that the comparison behaves correctly regardless of whether time_t is signed or unsigned.*

Source Code 8: Handling time_t Return Values C Code

```c
#include <time.h>
#include <stdio.h>

void check_time(void) {
    time_t now = time(NULL);
    if (now != (time_t)-1) {    // Proper comparison
        printf("Currentutime:u%ld\n", (long)now);
    } else {
        printf("Erroruretrievingutime.\n");
    }
}

int main(void) {
    check_time();
    return 0;
}
```

- *It eliminates the risk of logical errors when handling the time return value.*

```
❯ frama-c -eva atim.c
[kernel] Parsing atim.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function time
[eva] using specification for function printf_va_1
[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function check_time:
  now ∈ [--..--]
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 8 statements reached (out of 8): 100% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     3 valid     0 unknown     0 invalid      3 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 8

Compliant Code - Handling Memory Functions (memset)

- *The code uses 0 as the second argument for memset, which is safe and valid.*

Source Code 9: Handling Memory Functions (memset) C Code

```c
#include <string.h>
#include <stdio.h>

signed int *init_memory(signed int *array, size_t n) {
    return memset(array, 0, n); // Safe to set to zero
}

int main(void) {
    unsigned int array[10]; // Explicitly using signed int
    init_memory(array, sizeof(array));
    printf("Firstuelementu(initializedutouzero):u%d\n", array[0]); // Should print 0
    return 0;
}
```

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[kernel:typing:incompatible-types-call] usig.c:10: Warning:
  expected 'int *' but got argument of type 'unsigned int *': array
[variadic:typing] usig.c:11: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned int to int.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] FRAMAC_SHARE/libc/string.h:151:
  cannot evaluate ACSL term, unsupported ACSL construct: logic function memset
[eva] using specification for function printf_va_1
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function init_memory:
  array[0..9] ∈ {0}
[eva:final-states] Values at end of function main:
  array[0..9] ∈ {0}
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 6 statements reached (out of 6): 100% coverage.
  ------------------------------------------------------------------------
  Some errors and warnings have been raised during the analysis:
    by the Eva analyzer:      0 errors    0 warnings
    by the Frama-C kernel:    0 errors    1 warning
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     2 valid     0 unknown     0 invalid      2 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 9

Source Code 10: Handling Memory Functions (memset) C Code

```c
#include <string.h>
#include <stdio.h>

signed int *init_memory(signed int *array, size_t n) {
    return memset(array, 0, n); // Safe to set to zero
}

int main(void) {
    signed int array[10];  // Explicitly using signed int
    init_memory(array, sizeof(array));
    printf("Firstuelementu(initializedutouzero):u%d\n", array[0]); // Should print 0
    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] FRAMAC_SHARE/libc/string.h:151:
  cannot evaluate ACSL term, unsupported ACSL construct: logic function memset
[eva] using specification for function printf_va_1
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function init_memory:
  array[0..9] ∈ {0}
[eva:final-states] Values at end of function main:
  array[0..9] ∈ {0}
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 6 statements reached (out of 6): 100% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     2 valid     0 unknown     0 invalid      2 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 10

INT32-C. Ensure that operations on signed integers do not result in overflow

Signed integer overflow leads to undefined behavior in C, meaning compilers can handle it in various ways, potentially introducing vulnerabilities. This document aims to highlight the importance of preventing overflow in signed integer operations, providing compliant and noncompliant code examples, and detailing the reasoning behind these solutions.

Importance of Handling Overflow

Operations on signed integers can overflow if the resulting value exceeds the maximum or minimum representable values for the type. This is particularly critical in operations like

Division

- *Division is between two operands of arithmetic type.*
- *Overflow can occur during two's complement signed integer division when the div- idend is equal to the minimum (negative) value for the signed integer type and the divisor is equal to −1.*

- *Division operations are also susceptible to divide-by-zero errors.*

Compliant Code - Division

This compliant solution eliminates the possibility of divide-by-zero errors or signed overflow.

Source Code 11: Division C Code

```c
#include <stdio.h>
#include <limits.h>

void func(signed long s_a, signed long s_b) {
    unsigned long result;

    // Check for division by zero and overflow for signed long division
    if (s_b == 0 || (s_a == LONG_MIN && s_b == -1)) {
        printf("Error:u Divisionubyuzerouoruoverflow\n");
    } else {
        result = s_a / s_b;
        printf("Result:u%ld\n", result);
    }
}

int main() {
    func(-2147483648, -1); // Example that may cause overflow
    func(100, 5);          // Safe division

    return 0;
}
```

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[variadic:typing] usig.c:12: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned long to long.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  result ∈ {20; 2147483648}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 10 statements reached (out of 15): 66% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 11

Source Code 12: Division C Code

```c
#include <stdio.h>
#include <limits.h>

void func(signed long s_a, signed long s_b) {
    signed long result;

    // Check for division by zero and overflow for signed long division
    if (s_b == 0 || (s_a == LONG_MIN && s_b == -1)) {
        printf("Error:u Divisionubyuzerouoruoverflow\n");
    } else {
        result = s_a / s_b;
        printf("Result:u%ld\n", result);
    }
}

int main() {
    func(-2147483648, -1); // Example that may cause overflow
    func(100, 5);          // Safe division

    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  result ∈ {20; 2147483648}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 10 statements reached (out of 15): 66% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 12

Remainder

- *The remainder operator provides the remainder when two operands of integer type are divided.*
- *Because many platforms implement remainder and division in the same instruction, the remainder operator is also susceptible to arithmetic overflow and division by zero.*

Compliant Code - Remainder

This compliant solution also tests the remainder operands to guarantee there is no possibility of an overflow

Source Code 13: Remainder C Code

```
#include <stdio.h>
#include <limits.h>

void func(signed long s_a, signed long s_b) {
    unsigned long result;

    // Check for division by zero or overflow in signed long division
    if (s_b == 0 || (s_a == LONG_MIN && s_b == -1)) {
        printf("Error:u Divisionubyuzerouoruoverflow\n");
    } else {
        result = s_a % s_b;
        printf("Remainder:u%ld\n", result);
    }
}

int main() {
    func(-2147483648, -1); // Example that may cause overflow
    func(100, 7);              // Safe remainder

    return 0;
}
```

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[variadic:typing] usig.c:12: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned long to long.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  result ∈ {0; 2}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 10 statements reached (out of 15): 66% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 13

Source Code 14: Remainder C Code

```c
#include <stdio.h>
#include <limits.h>

void func(signed long s_a, signed long s_b) {
    signed long result;

    // Check for division by zero or overflow in signed long division
    if (s_b == 0 || (s_a == LONG_MIN && s_b == -1)) {
        printf("Error:u Divisionubyuzerouoruoverflow\n");
    } else {
        result = s_a % s_b;
        printf("Remainder:u%ld\n", result);
    }
}

int main() {
    func(-2147483648, -1); // Example that may cause overflow
    func(100, 7);              // Safe remainder

    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  result ∈ {0; 2}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 10 statements reached (out of 15): 66% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 14

Left-Shift Operator

- *The left-shift operator takes two integer operands. The result of E1 « E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros.*
- *If E1 has a signed type and nonnegative value, and*

$$E_1 \times 2^{E2}$$

is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

- *In almost every case, an attempt to shift by a negative number of bits or by more bits than exist in the operand indicates a logic error.*

Compliant Code - Left-Shift Operator

This compliant solution eliminates the possibility of overflow resulting from a left-shift operation

Source Code 15: Left-Shift Operator C Code

```c
#include <stdio.h>
#include <limits.h>

void func(signed long si_a, signed long si_b) {

    unsigned long result;

    // Check for left shift overflow:
    // - Ensure si_b is within the valid range
    // - Ensure si_a is within a range that doesn't overflow when shifted
    if (si_b < 0 || si_b >= sizeof(signed long) * 8 || si_a > (LONG_MAX >> si_b)) {
        printf("Error:uLeftushiftuoverflow\n");
    } else {
        result = si_a << si_b;
        printf("LeftuShiftuResult:u%ld\n", result);
    }
}

int main() {
    func(1, 33); // Example that may cause overflow (on systems where long is 32 bits)
    func(4, 2);    // Safe left shift

    return 0;
}
```

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[variadic:typing] usig.c:15: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned long to long.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  result ∈ {16; 8589934592}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 10 statements reached (out of 14): 71% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 15

Source Code 16: Left-Shift Operator C Code

```c
#include <stdio.h>
#include <limits.h>

void func(signed long si_a, signed long si_b) {

    signed long result;

    // Check for left shift overflow:
    // - Ensure si_b is within the valid range
    // - Ensure si_a is within a range that doesn't overflow when shifted
    if (si_b < 0 || si_b >= sizeof(signed long) * 8 || si_a > (LONG_MAX >> si_b)) {
        printf("Error:u  Leftushiftuoverflow\n");
    } else {
        result = si_a << si_b;
        printf("Leftu Shiftu Result:u%ld\n", result);
    }
}

int main() {
    func(1, 33); // Example that may cause overflow (on systems where long is 32 bits)
    func(4, 2);  // Safe left shift

    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  result ∈ {16; 8589934592}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 10 statements reached (out of 14): 71% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 16

Unary Negation

The unary negation operator takes an operand of arithmetic type. Overflow can occur during two's complement unary negation when the operand is equal to the minimum (negative) value for the signed integer type.

Compliant Code - Unary Negation

This compliant solution tests the negation operation to guarantee there is no possi- bility of signed overflow

Source Code 17: Unary Negation C Code

```c
#include <stdio.h>
#include <limits.h>

void func(signed long s_a) {

    unsigned long result;

    // Check for unary negation overflow for signed long
    if (s_a == LONG_MIN) {
        printf("Error:u Unaryunegationuoverflow\n");
    } else {
        result = -s_a;
        printf("Negatedu Result:u%ld\n", result);
    }
}

int main() {
    func(-2147483648); // Example that may cause overflow
    func(100);         // Safe negation

    return 0;
}
```

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[variadic:typing] usig.c:13: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned long to long.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  result ∈ {2147483648; 18446744073709551516}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 8 statements reached (out of 9): 88% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions       0 valid     0 unknown     0 invalid      0 total
    Preconditions    1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 17

Source Code 18: Unary Negation C Code

```c
#include <stdio.h>
#include <limits.h>

void func(signed long s_a) {

    signed long result;

    // Check for unary negation overflow for signed long
    if (s_a == LONG_MIN) {
        printf("Error:u Unaryunegationuoverflow\n");
    } else {
        result = -s_a;
        printf("Negatedu Result:u%ld\n", result);
    }
}

int main() {
    func(-2147483648); // Example that may cause overflow
    func(100);         // Safe negation

    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  result ∈ {-100; 2147483648}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 8 statements reached (out of 9): 88% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 18

Compliant Code - (Right Shift)

This compliant solution eliminates the possibility of shifting by greater than or equal to the number of bits that exist in the precision of the left operand

Source Code 19: Right Shift C Code

```
#include <inttypes.h>
#include  <limits.h>
#include  <stddef.h>
#include <stdio.h>

extern  size_t  popcount(uintmax_t);
#define PRECISION(x) (sizeof(x) * CHAR_BIT) // Calculate precision based on type size

void func(signed int si_a, signed int si_b) {
    unsigned int sresult = 0;

    // Check if the shift count exceeds the precision of signed int
    if (si_b < 0 || si_b >= PRECISION(INT_MAX)) {
        printf("Error:u Shiftucountuexceedsuprecisionuofusigneduint.\n");
    } else {
        // Perform right shift
        sresult = si_a >> si_b;
        printf("Resultuofurightushift:u%d\n",  sresult);
    }
}

int main() {
    signed int a = 10;
    signed int b = 2;   // Valid shift

    // Test compliant right shift
    func(a, b);    // Should show result

    return 0;
}
```

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[variadic:typing] usig.c:18: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned int to int.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  sresult ∈ {2}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  a ∈ {10}
  b ∈ {2}
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 11 statements reached (out of 14): 78% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 19

Source Code 20: Right Shift C Code

```c
#include <inttypes.h>
#include  <limits.h>
#include <stddef.h>
#include <stdio.h>

extern  size_t  popcount(uintmax_t);
#define PRECISION(x) (sizeof(x) * CHAR_BIT) // Calculate precision based on type size

void func(signed int si_a, signed int si_b) {
    signed int sresult = 0;

    // Check if the shift count exceeds the precision of signed int
    if (si_b < 0 || si_b >= PRECISION(INT_MAX)) {
        printf("Error:u Shiftucountuexceedsuprecisionuofusigneduint.\n");
    } else {
        // Perform right shift
        sresult = si_a >> si_b;
        printf("Resultuofurightushift:u%d\n",  sresult);
    }
}

int main() {
    signed int a = 10;
    signed int b = 2;   // Valid shift

    // Test compliant right shift
```

```
    func(a, b);    // Should show result

    return 0;
}
```

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_2
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function func:
  sresult ∈ {2}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  a ∈ {10}
  b ∈ {2}
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 11 statements reached (out of 14): 78% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 20

INT35-C. Use correct integer precisions

Complaint Code

- *The compliant version uses a macro PRECISION to determine the actual precision of the unsigned integer type, ensuring that the exponent used in shifting operations is valid.*
- *This approach accounts for the number of bits actually used to represent values.*

Source Code 21: Complaint C Code

```
#include <stddef.h>
```

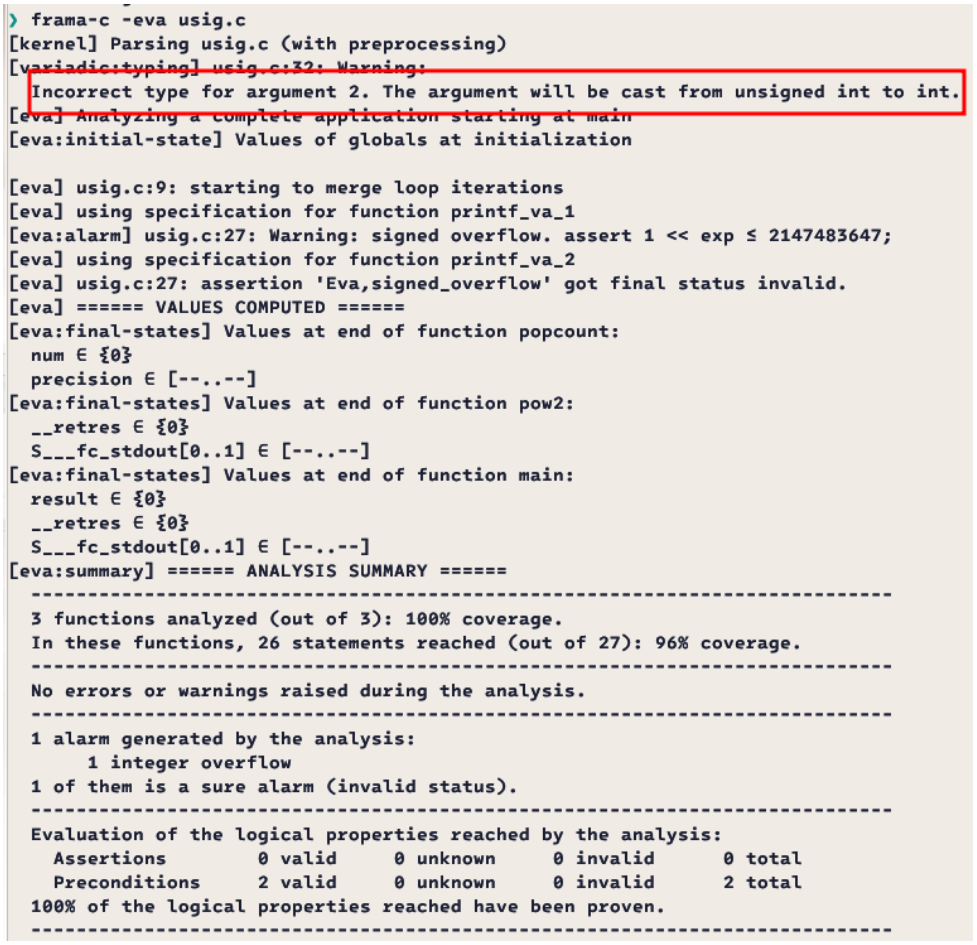
```c
# include <stdint.h>
# include <limits.h>
#include <stdio.h>

// Function to count set bits (popcount)
size_t popcount(uintmax_t num) {
    size_t precision = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            precision++;
        }
        num >>= 1;
    }
    return precision;
}

#define PRECISION(umax_value) popcount(umax_value)

// Function to compute 2^exp, where exp is a signed integer
signed int pow2(signed int exp) {
    if (exp < 0 || exp >= PRECISION(INT_MAX)) {
        // Handle error if exponent is negative or exceeds precision of signed int
        printf("Error:u Exponentuoutuofurange.\n");
        return 0; // Return 0 for this example
    }
    return 1 << exp; // Shift left by exp bits
}

int main() {
    unsigned int result = pow2(31); // Trying to compute 2^31 for signed int
    printf("Result:u%d\n", result);
    return 0;
}
```



```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[variadic:typing] usig.c:32: Warning:
  Incorrect type for argument 2. The argument will be cast from unsigned int to int.
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] usig.c:9: starting to merge loop iterations
[eva] using specification for function printf_va_1
[eva:alarm] usig.c:27: Warning: signed overflow. assert 1 << exp ≤ 2147483647;
[eva] using specification for function printf_va_2
[eva] usig.c:27: assertion 'Eva,signed_overflow' got final status invalid.
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function popcount:
  num ∈ {0}
  precision ∈ [--..--]
[eva:final-states] Values at end of function pow2:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  result ∈ {0}
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  3 functions analyzed (out of 3): 100% coverage.
  In these functions, 26 statements reached (out of 27): 96% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  1 alarm generated by the analysis:
       1 integer overflow
  1 of them is a sure alarm (invalid status).
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     2 valid     0 unknown     0 invalid      2 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 21

Source Code 22: Complaint C Code

```c
# include <stddef.h>
# include <stdint.h>
# include <limits.h>
#include <stdio.h>

// Function to count set bits (popcount)
size_t popcount(uintmax_t num) {
    size_t precision = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            precision++;
        }
        num >>= 1;
    }
    return precision;
}

#define PRECISION(umax_value) popcount(umax_value)

// Function to compute 2^exp, where exp is a signed integer
signed int pow2(signed int exp) {
    if (exp < 0 || exp >= PRECISION(INT_MAX)) {
        // Handle error if exponent is negative or exceeds precision of signed int
        printf("Error:uExponentuoutuofurange.\n");
        return 0; // Return 0 for this example
    }
    return 1 << exp; // Shift left by exp bits
```

```
}

int main() {
    signed int result = pow2(31); // Trying to compute 2^31 for signed int
    printf("Result:u%d\n", result);
    return 0;
}
```

```
> frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] sig.c:9: starting to merge loop iterations
[eva] using specification for function printf_va_1
[eva:alarm] sig.c:27: Warning: signed overflow. assert 1 << exp ≤ 2147483647;
[eva] using specification for function printf_va_2
[eva] sig.c:27: assertion 'Eva,signed_overflow' got final status invalid.
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function popcount:
  num ∈ {0}
  precision ∈ [--..--]
[eva:final-states] Values at end of function pow2:
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  result ∈ {0}
  __retres ∈ {0}
  S___fc_stdout[0..1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  3 functions analyzed (out of 3): 100% coverage.
  In these functions, 25 statements reached (out of 26): 96% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  1 alarm generated by the analysis:
       1 integer overflow
  1 of them is a sure alarm (invalid status).
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     2 valid     0 unknown     0 invalid      2 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 22

- *The compliant code properly checks the precision of the integer type before performing bitwise operations.*
- *The output demonstrates that the function can handle the operation correctly with- out errors.*

INT36-C. Converting a pointer to integer or integer to pointer

- *Conversions between integers and pointers can have undesired consequences de- pending on the implementation.*
- *Do not convert an integer type to a pointer type if the resulting pointer is in- correctly aligned, does not point to an entity of the referenced type, or is a trap representation.*
- *Do not convert a pointer type to an integer type if the result cannot be represented in the integer type.*
- *The mapping between pointers and integers must be consistent with the addressing structure of the execution environment. Issues may arise, for example, on architec- tures that have a segmented memory model.*

Complaint Code

Source Code 23: Complaint C Code

```c
#include <stdint.h>
#include <stdio.h>

void f(void) {
    char *ptr = "Hello,uWorld!";
    intptr_t number = (intptr_t)ptr; // Cast to signed integer type
    printf("Number:u%ld\n", (long)number); // Print as signed long
}

int main() {
    f();
    return 0;
}
```

- *In this compliant version, the pointer is converted to uintptr_t, which is designed to safely hold pointer values without data loss.*
- *This ensures that the conversion is safe and can be reversed without issues.*

```
❯ frama-c -eva usig.c
[kernel] Parsing usig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva:alarm] usig.c:6: Warning:
  pointer downcast. assert (unsigned long)ptr ≤ 9223372036854775807;
[eva] using specification for function printf_va_1
[eva:garbled-mix:assigns] usig.c:7:
  The specification of function printf_va_1
  has generated a garbled mix of addresses
  for assigns clause __fc_stdout->__fc_FILE_data.
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function f:
  ptr ∈ {{ "Hello, World!" }}
  number ∈ {{ "Hello, World!" }}
  S___fc_stdout[0].__fc_FILE_id ∈ [--..--]
                [0].__fc_FILE_data ∈
                {{ garbled mix of &{"Hello, World!"}
                 (origin: Library function {usig.c:7}) }}
                [1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0].__fc_FILE_id ∈ [--..--]
                [0].__fc_FILE_data ∈
                {{ garbled mix of &{"Hello, World!"}
                 (origin: Library function {usig.c:7}) }}
                [1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 7 statements reached (out of 7): 100% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  1 alarm generated by the analysis:
       1 integer overflow
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 23

Source Code 24: Complaint C Code

```c
#include <stdint.h>
#include <stdio.h>

void f(void) {
    const char *ptr = "Hello,uWorld!"; // Use const char* for string literals
    uintptr_t number = (uintptr_t)ptr;  // Use uintptr_t for pointer to integer
        conversion
    printf("Number:u%lu\n", (unsigned long)number); // Print as unsigned long
}

int main() {
    f();
    return 0;
}
```

The output will correctly represent the pointer's address without data loss.

```
❯ frama-c -eva sig.c
[kernel] Parsing sig.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_1
[eva:garbled-mix:assigns] sig.c:7:
  The specification of function printf_va_1
  has generated a garbled mix of addresses
  for assigns clause __fc_stdout->__fc_FILE_data.
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function f:
  ptr ∈ {{ "Hello, World!" }}
  number ∈ {{ "Hello, World!" }}
  S___fc_stdout[0].__fc_FILE_id ∈ [--..--]
                [0].__fc_FILE_data ∈
                {{ garbled mix of &{"Hello, World!"}
                 (origin: Library function {sig.c:7}) }}
                [1] ∈ [--..--]
[eva:final-states] Values at end of function main:
  __retres ∈ {0}
  S___fc_stdout[0].__fc_FILE_id ∈ [--..--]
                [0].__fc_FILE_data ∈
                {{ garbled mix of &{"Hello, World!"}
                 (origin: Library function {sig.c:7}) }}
                [1] ∈ [--..--]
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  2 functions analyzed (out of 2): 100% coverage.
  In these functions, 7 statements reached (out of 7): 100% coverage.
  ------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ------------------------------------------------------------------------
  0 alarms generated by the analysis.
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        0 valid     0 unknown     0 invalid      0 total
    Preconditions     1 valid     0 unknown     0 invalid      1 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

Figure 24