2010 IEEE 21st International Symposium on Software Reliability Engineering

As-If Infinitely Ranged Integer Model

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
rbd@cs.cmu.edu

Thomas Plum
Plum Hall, Inc.
Kamuela HI, USA
tplum@plumhall.com

Will Dormann
David Keaton
Robert C. Seacord
David Svoboda
Alex Volkovitsky
Timothy Wilson
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{wd, dmk, rcs, svoboda, twilson}@cert.org

***Abstract*— Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs. This paper presents the As-if Infinitely Ranged (AIR) Integer model for eliminating vulnerabilities resulting from integer overflow, truncation, and unanticipated wrapping. The AIR Integer model either produces a value equivalent to that obtained using infinitely ranged integers or results in a runtime-constraint violation. With the exception of wrapping (which is optional), this model can be implemented by a C99-conforming compiler and used by the programmer with little or no change to existing source code. Fuzz testing of libraries that have been compiled using a prototype AIR integer compiler has been effective in discovering vulnerabilities in software with low false positive and false negative rates. Furthermore, the runtime overhead of the AIR Integer model is low enough that typical applications can enable it in deployed systems for additional runtime protection.**

Keywords-software security; programming languages; empirical study

I. INTEGRAL SECURITY

The majority of software vulnerabilities result from coding errors. For example, 64% of the vulnerabilities in the National Vulnerability Database in 2004 resulted from programming errors [1]. The C and C++ languages are particularly prone to vulnerabilities because of the lack of type safety in these languages [2].

In 2007, MITRE reported that buffer overflows remain the number one issue as reported in operating system (OS) vendor advisories. It also reported that integer overflow, barely in the top 10 overall in the years preceding the report, was number two in OS vendor advisories [3].

Integer errors and vulnerabilities occur when programmers reason about infinitely ranged mathematical integers, while implementing their designs with the finite precision, integral data types supported by hardware and language implementations.

Integer values that originate from untrusted sources and are used in the following ways can easily result in vulnerabilities:

- as an array index in pointer arithmetic
- as a length or size of an object
- as the bound of an array (e.g., a loop counter)
- as an argument to a memory-allocation function

The remainder of this section describes integer behaviors that have resulted in real-world vulnerabilities.

A. *Signed Integer Overflow*

Signed integer overflow is undefined behavior in C, allowing implementations to silently wrap (the most common behavior), trap, or both. Because signed integer overflow produces a silent wraparound in most existing C and C++ implementations, some programmers assume that this is a well-defined behavior.

Conforming C and C++ compilers can deal with undefined behavior in many ways, such as ignoring the situation completely (with unpredictable results), translating or executing the program in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), or terminating a translation or execution (with the issuance of a diagnostic message). Because compilers are not obligated to generate code for undefined behaviors, those behaviors are candidates for optimization. For example, if GCC Version 4.1.1 detects an expression that depends on overflow, it may eliminate the expression as well as any dependent code.

B. *Unsigned Integer Wrapping*

Although unsigned integer wrapping is well-defined by the C standard as having modulo behavior, unexpected wrapping has led to numerous software vulnerabilities. A real-world example of vulnerabilities resulting from unsigned integer wrapping occurs in memory allocation. Wrapping can occur in `calloc()` and other memory allocation functions when the size of a memory region is being computed.[1]

For example, the following code fragments may lead to wrapping vulnerabilities:

```
C: p = calloc(sizeof(element_t), count);
```

[1] http://www.securityfocus.com/bid/5398

1071-9458/10 $26.00 © 2010 IEEE
DOI 10.1109/ISSRE.2010.29

IEEE computer society

```
C++: p = new ElementType[count];
```

The wrapping of calculations internal to these functions may result in too little storage being allocated and subsequent buffer overflows. This is mitigated by the draft standard *Programming Languages—C++, Final Committee Draft* which requires that a new expression throw an instance of `std::bad_array_new_length` on integer overflow [13]. Another well-known vulnerability resulting from unsigned integer wrapping occurred in the handling of the comment field in JPEG files [4].

C. Conversion Errors

Integer conversions, both implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data [5].

The only integer-type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness. Conversion of an integer to a smaller type results in truncation of the high-order bits.

Consequently, conversions from an integer with greater precision to an integer type with lesser precision can result in truncation, if the resulting value cannot be represented in the smaller type. Conversions to an integer of the same precision but different signedness can lead to misinterpreted data.

II. AIR Integer Model

To bring program behavior into greater agreement with the mathematical reasoning commonly used by programmers, the As-if Infinitely Ranged (AIR) Integer model guarantees that either integer values are equivalent to those obtained using infinitely ranged integers or a runtime exception occurs. The resulting system is easier to analyze because undefined behaviors have been defined and because the analyzer (either a tool or human) can safely assume that integer operations result in an as-if infinitely ranged value or trap. The model applies to both signed and unsigned integers, although either may be enabled or disabled per compilation unit using compiler options. Although the AIR Integer model is applicable to C++, our current focus is on C.

Traps are implemented using the existing hardware traps (such as divide-by-zero) or by invoking a runtime-constraint handler. Most functions defined by the bounds-checking interfaces annex of the C1X draft standard [9] include as part of their specification a list of runtime constraints, violations of which can be consistently handled at runtime.

Whether a program traps for given inputs depends on the exact optimizations carried out by a particular compiler version. If required, a programmer can implement a custom runtime-constraint handler to set a flag and continue (using the indeterminate value that was produced). In the future, an implementation that also supports C++ might throw an exception or invoke `terminate()`, rather than invoke a runtime-constraint handler. Alternatively, the runtime-constraint handler can throw an exception. We have not attempted to evaluate these, or other, alternatives for C++.

An *observation point* occurs at an output, including a volatile object access. AIR integers do not *require* a trap for every integer overflow or truncation error. In the AIR Integer model, it is acceptable to delay catching an incorrectly represented value until an observation point is reached or just before it causes *a critical undefined behavior* [9]. The trap may occur any time between the overflow or truncation and the output or critical undefined behavior. This model improves the ability of compilers to optimize, without sacrificing safety and security.

Critical undefined behavior is a means of differentiating between behaviors that can perform an out-of-bounds store and those that cannot. An out-of-bounds store is defined in the C1X draft standard titled *Programming Languages—C, Committee Draft* as an (attempted) access that, at runtime and for a given computational state, would modify (or, for an object declared volatile, fetch) one or more bytes that lie outside the bounds permitted by the C1X draft standard. The critical undefined behaviors are defined in the (normative) Annex L, "Analyzability" [9].

In the AIR Integer model, when an observation point is reached and before any critical undefined behavior occurs, any integer value in the output is correctly represented ("as-if infinitely ranged"), provided that traps have not been disabled and no traps have been raised. Optimizations are encouraged, provided the model is not violated.

All integer operations are included in the model. Pointer arithmetic (which results in a pointer) is not part of the AIR Integer model but can be checked by Safe Secure C/C++ methods [12].

A. Implementation Methods

The AIR Integer model permits a wide range of implementation methods, some of which might apply to different environments and implementations:

- Overflow or truncation can set a flag that compiler-generated code will test later.
- Overflow or truncation can immediately invoke a runtime-constraint handler.
- The testing of flags can be performed at an early point (such as within the same full expression) or be delayed (subject to some restrictions).

For example, in the following code

```
i = k + 1;
j = i * 3;
if (m < 0) a[i] = . . .;
```

the variable `j` does not need to be checked within this code fragment (but may need to be checked later). The variable `i` does not need to be checked unless and until the expression `a[i]` is evaluated.

Compilers may choose a single, cumulative, integer exception flag in some cases and one flag per variable in others, depending on what is most efficient in terms of speed and storage for the particular expressions involved. For example, in the following code

```
x++;  y++;  z++;
printf("%d", x);
```

the call to `printf()` is an observation point for the variable `x`. Any of the operations `x++`, `y++`, or `z++` can

result in an overflow. Consequently, it is necessary to test the value of the exception flag prior to the observation point (the call to `printf()`) and invoke the runtime-constraint handler if the exception flag is set:

```
// compiler clears integer exception flags
  x++;  y++;  z++;
  if (/* integer exception flags are set */)
    runtime_constraint_handler();
  printf("%d", x);
```

If only a single exception flag is used, one or more of the variables may contain an incorrectly represented value, but we cannot know which one. Consequently, the runtime-constraint handler will be invoked if any of the increment operations resulted in an overflow. In that case, it may be preferable for the compiler to generate a separate exception flag for `x` so that the runtime-constraint handler only has to be invoked if `x++` overflows.

Portably, if the code reaches an observation point without invoking a runtime-constraint handler, a programmer can only assume that all observable integer values are represented correctly. If a runtime-constraint error occurs, all integer values that have been modified since the last observation point contain indeterminate values. In cases where the programmer wants to rerun the calculation using a higher or arbitrary-precision integer, the programmer would need to recalculate the values for all indeterminate values.

Ideally, while we would like to eliminate implementation-defined behavior in the AIR Integer model, sufficient latitude must be provided for compiler implementers to optimize the resulting executable.

B. *Undefined Behavior*

One of the goals of the AIR Integer model is to eliminate previously undefined behaviors by providing optional predictable semantics for areas of C that are presently undefined (at some optimization cost). Changes from the existing, unbounded, undefined behavior that pose serious implementation problems in practice were not adopted under the model.

The following cases receive special handling in the AIR Integer model.

1) *Multiplicative Operators*

There is no defined infinite-precision result for division by zero. Processors typically trap, but that may not be universal. The AIR Integer model requires trapping.

When integers are divided, the result of the `/` operator is the algebraic quotient with any fractional part discarded. If the quotient of `a/b` is representable, the expression `(a/b)*b + a%b` is equal to `a`. Otherwise, the behavior of both `a/b` and `a%b` is undefined by C99, but processors commonly trap. For example, when using the IA-32 `idiv` instruction, dividing `INT_MIN` by `-1` results in a division error and generates an interrupt on vector 0 because the signed result (quotient) is too large for the destination [14]. The AIR Integer model requires trapping if the quotient is not representable.

The ISO/IEC JTC1/SC22/WG14 C standards committee discussed the behavior of `INT_MIN % -1` on the WG14 reflector and at the April 2009 Markham meeting [10]. The committee agreed that, mathematically, `INT_MIN % -1` equals 0. However, instead of producing the mathematically correct result, some architectures may trap. For example, implementations targeting the IA-32 architecture use the `idiv` instruction to determine the remainder. Consequently, `INT_MIN % -1` results in a division error and generates an interrupt on vector 0.

At the same Markham meeting, the committee decided that requiring C programs to produce 0 would render some compilers noncompliant with the standard and that adding this corner case could add a significant overhead. Consequently, the C1X draft standard [9] has been amended to state explicitly that if `a/b` is not representable, `a%b` is undefined.

The AIR Integer model requires that `a % -1` equals 0 for all values of `a` or, alternatively, trapping is performed. This violates the literal interpretation of "as-if infinite range" but reflects a concession to practical implementation issues.

2) *Shifts*

Shifting by a negative number of bits or by more bits than exist in the operand is undefined behavior in C99 and, in almost every case, indicates a bug (logic error). Signed left shifts of negative values or cases where the result of the operation is not representable in the type are undefined in C99 and implementation-defined in C90. Processors may reduce the shift amount modulo some quantity larger than the width of the type. For example, 32-bit shifts are implemented using the following instructions on IA-32:

```
sa[rl]l   %cl, %eax
```

The `sa[rl]l` instructions take the least significant 5 bits from `%cl` to produce a value in the range [0, 31] and then shift `%eax` that many bits. 64-bit shifts become the two-instruction sequence:

```
sh[rl]dl %cl, %eax, %edx
sa[rl]l  %cl, %eax
```

where `%eax` stores the least significant bits in the double word to be shifted and `%edx` stores the most significant bits.

In the AIR Integer model, shifts by negative amounts or amounts outside the width of the type trap because the results are not representable without overflow; consistent with guideline INT34-C of the *CERT C Secure Coding Standard*: "Do not shift a negative number of bits or more bits than exist in the operand" [5].

In the AIR Integer model, signed left shifts on negative values must not trap if the result is representable without overflow. If the value is not representable in the type, the implementation must trap. For example, `a << b == a * 2`^{b} if `b >= 0`, and `a * 2`^{b} is representable without overflow in the type. For right shifts, a >> b == a / 2^{b} if `a >= 0, b >= 0`, and 2^{b} is representable without overflow in the type. If `a < 0, b >= 0`, and 2^{b} is representable without overflow in the type, `a >> b == -1 + (a + 1) / 2`^{b}. Unsigned left shifts never trap under the AIR

Integer model because unsigned left shifts are generally perceived by programmers as losing data, and a large amount of existing code assumes modulo behavior. For example, in the following code from the JasPer image processing library,[2] Version 1.900.1, `tmpval` has `uint_fast32_t` type:

```
while (--n >= 0) {
  c = (tmpval >> 24) & 0xff;
  if (jas_stream_putc(out, c) == EOF) {
    return -1;
  }
  tmpval = (tmpval << 8) & 0xffffffff;
}
```

The modulo behavior of `tmpval` is assumed in the left shift operation.

3) Fussy Overflows

One problem with trapping is *fussy overflows*, which are overflows in intermediate computations that do not affect the resulting value. For example, on two's complement architectures, the following code

```
int x = /* nondeterministic value */;
x = x + 100 – 1000;
```

overflows for values of `x > INT_MAX - 100` but underflows during the subsequent subtraction, resulting in a correct as-if infinitely ranged integer value.

In this case, most compilers will perform constant folding to simplify the above expression to `x – 900`, eliminating the possibility of a fussy overflow. However, there are situations where this will not be possible, for example:

```
int x = /* nondeterministic value */;
int y = /* nondeterministic value */;
x = x + 100 – y;
```

Because this expression cannot be optimized, a fussy overflow may result in a trap, and a potentially successful operation may be converted into an error condition.

C. Enabling and Disabling Unsigned Integer Wrapping

The default behavior under the AIR Integer model is to trap unsigned integer wrapping.

Unsigned integer semantics are problematic because unsigned integer wrapping poses a significant security risk but is well-defined by the C standard. Also, in legacy code, the wrapping behavior can be necessary to ensure correct behavior. Consequently, it is necessary to provide mechanisms to enable and disable wrapping for unsigned integers.

It is theoretically possible to introduce new identifiers, such as `__wrap` and `__trap`, to be used as named attributes to enable or disable wrapping for individual integer variables, both signed and unsigned. They could be implemented as variable attributes in GCC or using `__declspec` or a similar mechanism in Microsoft Visual Studio. Enabling or disabling wrapping and trapping per variable has implications for the type system: for example, what happens when you combine a wrapping variable with a trapping variable? It also has implications for type safety: for example, what happens when you pass a trapping variable as an argument to a function that accepts a wrapping parameter?

Because of these added complications, the AIR Integer model only supports enabling or disabling unsigned integer wrapping per compilation unit.

Compiler options can be provided to enable or disable wrapping for all unsigned integer variables per compilation unit. Existing code that depends on modulo behavior for unsigned integers should be isolated in a separate compilation unit and compiled with wrapping disabled.

When an unsigned integer defined in one compilation unit compiled with wrapping semantics is combined with another unsigned integer defined in a separate compilation unit with trapping semantics, the resulting value has the default behavior of the compilation unit in which the operation occurs.

Because a large number of exploitable software vulnerabilities result from unsigned integer wrapping, we strongly recommend that the trap behavior be the default for all new code and, for as much legacy code as possible, consistent with adequate testing and code review.

D. The Usual Arithmetic Conversions

In cases where a compilation unit is compiled with wrapping disabled for unsigned integers, operations can take place between signed integers with trapping semantics and unsigned integers with wrapping semantics. In these cases, the semantics of the resulting variable (trapping or wrapping) depends on the integer promotions and the usual arithmetic conversions defined by C99. In cases where the resulting variable is a signed integer type, trapping semantics apply; in cases where the resulting value is an unsigned integer type, wrapping semantics are used.

E. Integer Constants

In C99, it is a constraint violation if the value of a constant is outside the range of representable values for its type. A C99-conforming implementation must produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or other translation unit contains a violation of any constraint.

For constant expressions, the AIR Integer model requires the compiler to use arbitrary-precision, signed arithmetic to evaluate an integer constant expression (even an unsigned one) and then issue a fatal diagnostic if the final result does not fit the appropriate type.

For example, the expression

```
((unsigned)0 - 1)
```

produces a constraint violation and should result in a fatal diagnostic if compiled.

[2] http://www.ece.uvic.ca/~mdadams/jasper/

F. Expressions Involving Integer Variables and Constants

Because of macro expansion, another common case in C programs involves expressions that include some number of variables and some number of constant values such as

```
V1 + 1u + V2 - 2u
```

In this case, the compiler can reorder the expressions and reduce to a single constant value, for example

```
V1 + V2 - 1u
```

regardless of whether it is compiled with trapping enabled or disabled for unsigned integer values.

G. Runtime-Constraint Handling

Implementations are free to detect any case of undefined behavior and treat it as a runtime-constraint violation by calling the runtime-constraint handler. This license comes directly from the definition of undefined behavior. Consequently, the AIR implementation uses the runtime-constraint mechanisms defined by ISO/IEC TR 24731-1 [11] and by the C1X draft standard [9] for handling integer exception conditions.

H. Optimizations

An important consideration in adopting a new integer model is the effect on compiler optimization and vice versa. C-language experts are accustomed to evaluating the CPU cost of various proposals. A typical approach is to compare the CPU cost of solving the problem in the compiler versus the (zero) cost of not doing so. Consequently, we take this approach to demonstrate that solving the problem does not introduce a large amount of overhead. However, for the AIR Integer model, it is also useful to compare the CPU cost of analyzable, generated code versus the CPU cost of the programmer's extra program logic added to the intrinsic CPU cost of the optimized construct. This comparison justifies putting a greater burden on the compiler when compiling otherwise insecure constructs in analyzable mode.

Regardless, performance is always an issue when evaluating new models, and it is important to preserve existing optimizations while discovering new ones. Consequently, the AIR Integer model does not prohibit any optimizations that are permitted by the C standard but does require a diagnostic any time the compiler performs an optimization based on

- signed overflow wrapping
- unsigned wrapping
- signed overflow not occurring (although value-range analysis cannot guarantee it will not)
- unsigned wrapping not occurring (although value-range analysis cannot guarantee it will not)

For example, AIR integers allow optimizations based on algebraic simplification without a diagnostic:

```
(signed) (a * 10) / 10
```

This expression can be optimized to `a`. There is no need to preserve the possibility of trapping `a * 10`.

The expression

```
(a - 10) + (b - 10)
```

can be optimized to

```
(a + b) - 20
```

While `(a + b)` could produce a trap, there is also a possibility that either `(a - 10)` or `(b - 10)` could result in a trap in the original expression. If the application can be sure that each output is represented correctly, its ability to determine whether a trap might have occurred by a different strategy is unimportant.

Optimizations that assume that integer overflow does not trap require a diagnostic because that assumption is inconsistent with the AIR Integer model. For example, certain optimizations operate on the basis that a loop must terminate by exactly reaching the limit `n`, and therefore the number of iterations can be determined by an exact division with no remainder such as

```
for (i = 0; i != n; i += 3)
```

This loop should also be diagnosed because it violates rule MSC21-C of the *CERT C Secure Coding Standard*: "Use inequality to terminate a loop whose counter changes by more than one" [5].

Diagnostics are also required for optimizations on pointer arithmetic that assume wrapping cannot occur.

I. The `rsize_t` Type

The `rsize_t` type defined as part of the bounds-checking interfaces annex of the C1X draft standard [9] can be used in a complementary fashion to AIR integers and is consequently subsumed as part of the overall solution. Functions that accept parameters of type `rsize_t` diagnose a constraint violation if the values of those parameters are greater than `RSIZE_MAX`. Extremely large sizes frequently indicate that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect errors. For machines with large address spaces, C1X recommends that `RSIZE_MAX` be defined as the smaller of these two, even if this limit is smaller than the size of some legitimate, but very large, objects:

- the size of the largest object supported
- `SIZE_MAX >> 1)`

The *CERT C Secure Coding Standard* recommends using `rsize_t` or `size_t` for all integer values representing the size of an object (INT01-C) [5].

III. Related Work

This section describes existing and contemplated alternative approaches to the problem of integral security in C and explains why they don't adequately address the issues.

A. The GCC `-ftrapv` Flag

GCC provides an `-ftrapv` compiler option that provides limited support for detecting integer overflows at runtime. The GCC runtime system generates traps for signed overflow on addition, subtraction, and multiplication operations for programs compiled with the `-ftrapv` flag. This trapping is accomplished by invoking existing, portable

library functions that test an operation's post-conditions and call the C library `abort()` function when results indicate that an integer error has occurred [2].

The `-ftrapv` option is known to have substantial problems. The `__addvsi3()` function requires a function call and conditional branching, which can be expensive on modern hardware. An alternative implementation tests the processor overflow condition code, but it requires assembly code and is non-portable. Furthermore, the GCC `-ftrapv` flag only works for a limited subset of signed operations and always results in an `abort()` when a runtime overflow is detected. Discussions for how to trap signed integer overflows in a reliable and maintainable manner are ongoing within the GCC community.

B. Pre-Condition Testing

Another approach to eliminating integer exception conditions is to test the values of the operands before an operation to prevent overflow and wrapping from occurring. This testing is especially important for signed integer overflow, which is undefined behavior and may result in a trap on some architectures (e.g., a division error on IA-32). The complexity of these tests varies significantly.

A pre-condition test for detecting wrapping when adding two unsigned integers is relatively simple. However, a strictly conforming test to ensure that a signed multiplication operation does not result in an overflow is significantly more involved. Examples of these pre-condition tests and others are shown in the *CERT C Secure Coding Standard* [5]:

- INT30-C. Ensure that unsigned integer operations do not wrap
- INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data
- INT32-C. Ensure that operations on signed integers do not result in overflow

Detecting an overflow in this manner can be relatively expensive, especially if the code is strictly conforming. Frequently, these checks must be in place before suspect system calls that may or may not perform similar checks prior to performing integer operations. Redundant testing by the caller and by the called is a style of defensive programming that has been largely discredited within the C and C++ community. The usual discipline in C and C++ is to require validation only on one side of each interface.

Furthermore, branches can be expensive on modern hardware, so programmers and implementers work hard to keep branches out of inner loops. This expense argues against requiring the application programmer to pretest all arithmetic values to prevent rare occurrences such as overflow. Preventing runtime overflow by program logic is sometimes easy, sometimes complicated, and sometimes extremely difficult. Clearly, some overflow occurrences can be diagnosed in advance by static-analysis methods. But no matter how good this analysis is, some code sequences still cannot be detected before runtime. In most cases, the resulting code is much less efficient than what a compiler could generate to detect overflow.

The underlying process of code generation may be immensely complicated. However, in general, it is best to avoid complexity in the code that end-user programmers are required to write.

C. Saturation Semantics

Verifiably in-range operations are often preferable to treating out-of-range values as an error condition because the handling of these errors has been shown to cause denial-of-service problems in actual applications (e.g., when a program aborts). The quintessential example of this incorrect handling is the failure of the Ariane 5 launcher that resulted from an improperly handled conversion error that caused the processor to be shut down [6].

A program that detects an imminent integer overflow may either trap or produce an integer result that is within the range of representable integers on that system. Some applications, particularly in embedded systems, are better handled by producing a verifiably in-range result because it allows the computation to proceed, thereby avoiding a denial-of-service attack. However, when continuing to produce an integer result in the face of overflow, the question of what integer result to return must be considered.

The saturation and modwrap algorithms and the technique of restricted-range usage produce integer results that are always within a defined range. This range is between the integer values `MIN` and `MAX` (inclusive), where `MIN` and `MAX` are two representable integers with `MIN < MAX`.

For saturation semantics, assume that the mathematical result of the computation is `result`. The value actually returned to the user is shown in **Error! Reference source not found.**.

TABLE I. SATURATION SEMANTICS

Range of Mathematical Result	Result Returned
`MAX < result`	`MAX`
`MIN <= result <= MAX`	`result`
`result < MIN`	`MIN`

D. Overflow Detection

C99 provides the `<fenv.h>` header to support the floating-point exception status flags and directed-rounding control modes required by IEC 60559 and similar floating-point state information. This support includes the ability to determine which floating-point exception flags are set.

A potential solution to handling integer exceptions in C is to provide an inquiry function (just as C provides for floating point) that interrogates the status flags being maintained by the (compiler-specific) assembler code that performs the various integer operations. If the inquiry function is called after an integer operation and returns a "no overflow" status, the value is reliably represented correctly.

At the assembler code level, the cost of detecting overflow is zero or nearly zero. Many architectures do not even have an instruction for "add two numbers but do NOT

set the overflow or carry bit;"[3] the detection occurs for free whether it is desired or not. But only the specific compiler code generator knows what to do with those status flags.

These inquiry functions may be defined, for example, by translating the `<fenv.h>` header into an equivalent `<ienv.h>` header that provides access to the integer exception environment. This header would support the integer exception status flags and other similar integer exception state information.

However, anything that can be performed by an `<ienv.h>` interface could be performed better by the compiler. For example, the compiler may choose a single, cumulative integer exception flag in some cases and one flag per variable in others, depending on what is most efficient in terms of speed and storage for the particular expressions involved. Additionally, the concept of a runtime-constraint handler is a new feature of C1X. Consequently, when designing `<fenv.h>`, the C standards committee defined an interface that put the entire burden on the programmer.

Floating-point code is different from integer code in that it includes concepts such as rounding mode, which need not be considered for integers. Additionally, floating point has a specific value, NaN (Not a Number), which indicates that an unrepresentable value was generated by an expression. Sometimes floating-point programmers want to terminate a computation when a NaN is generated; at other times, they want to print out the NaN because its existence conveys valuable information (and there might be one NaN in the middle of an array being printed out, with the rest of the values being valid results). Because of the combination of NaNs and the lack of runtime-constraint handlers, the programmer needed to be given more control.

In general, there is no NaI (Not an Integer) value, so there is no requirement to preserve such a value to allow it to be printed out. Therefore, the programmer does not need fine control over whether an integer runtime-constraint handler gets called after each operation. Without this requirement, it is preferable to keep the code simple and let the compiler do the work, which it can generally do more reliably and efficiently than individual application programmers.

E. *Runtime Integer Checking (RICH)*

Brumley and colleagues developed a static, program transformation tool, called RICH, that takes as input any C program and outputs object code that monitors its own execution to detect integer overflows and other bugs [8]. At compile time, RICH instruments the target program with runtime checks of all unsafe integer operations. At runtime, the inserted instrumentation checks each integer operation. When a check detects an integer error, it generates a warning and optionally terminates the program. When assessed using the industry standard SPECCPU2006, RICH succeeded in compiling 9 of the 12 SPECINT benchmarks at `O0` (the perl, gcc, and xalancbmk tests failed) with 220% overhead. The `O2` results were less interesting because RICH had errors on the first 9 benchmarks and only completed the last 3 benchmarks.

F. *Clang Implementation*

Chisnall implemented the AIR Integer model for Clang using the LLVM overflow-checked operations.[4] This implementation checks the flag immediately after any signed integer `+`, `-`, or `*` integer operation and jumps to a handler function if overflow occurred. Conditional jumps on overflow are cheap because the branch predictor will almost always guess correctly. By allowing a custom handler function, rather than aborting, Clang allows for calling `longjmp()` or some unwind library functions in cases where overflow occurred. This works well with the optimizer, which can eliminate the test for cases where the value can be proven to be in-range.

G. *GCC No-Undefined-Overflow*

Guenther has proposed a new no-undefined-overflow branch for GCC. Its goal is to make overflow behavior explicit per operation and to eliminate all constructs in the GIMPLE intermediate language (IL) that invoke undefined behavior. To support languages that have undefined semantics on overflowing operations such as C and C++, new unary and binary operators that implicitly encode value-range information about their operands are added to the middle end, noting that the operation does not overflow. These does-not-overflow operators transform the undefined behavior into a valid assumption, making the GIMPLE IL fully defined. Consequently, the front-end and value-range analysis must determine if operations overflow and generate the appropriate IL. Instructions such as `NEGATE_EXPR`, `PLUS_EXPR`, `MINUS_EXPR`, `MULT_EXPR`, and `POINTER_PLUS_EXPR` would have wrapping, no-overflow, and trapping variants.

The trapping variants are indicated by a `V` for overflow (e.g., `PLUSV_EXPR` is the trapping variant for `PLUS_EXPR`) and by `NV` for no overflow (e.g., `PLUSNV_EXPR`). The no-overflow variant also wraps if it overflows so that existing code continues to function.

The GCC no-undefined-overflow branch, when implemented, should greatly facilitate the implementation of the AIR Integer model within GCC.

IV. Performance & Efficacy Study

The AIR Integer model has been fully implemented in a proof-of-concept modification to the GCC compiler Version 4.5.0 for IA-32 processors to study the performance and efficacy of the AIR Integer model.[5] The AIR Integer model is enabled by the `-fanalyzable` compile-time option.

[3] However, the load effective address (LEA) instruction in Intel architectures is commonly used for integer addition and does *not* set status flags.

[4] http://article.gmane.org/gmane.comp.compilers.clang.devel/4469

[5] The prototype is available for download at http://www.cert.org/secure-coding/integralsecurity.html.

Generated executables automatically invoke a runtime-constraint handler when an integer operation fails to produce an as-if infinitely ranged value.

To diagnose integer overflow on an IA-32 processor, we must know whether the arguments are signed or unsigned so that the appropriate flag (carry or overflow) can be checked. The overflow flag indicates that overflow has occurred for signed operations, while the carry flag can be safely ignored. For unsigned computations, the opposite is true. Unfortunately, GCC's last internal representation, the register transfer language (RTL), has no way of storing the signedness of arguments to operations. Doing so requires inserting a flag into the RTL data structure, the `rtx`, which carries signedness information to the GCC back end, where translation to assembly code is performed. Upon translation, the proper signedness information is available to produce the correct RTL pattern.

Conditional jumps are added to RTL patterns containing arithmetic operations to invoke a runtime-constraint handler in the event that signed overflow or unsigned wrapping occurred, as shown in the following example:

```
   // arithmetic
jn[co] .LANALYZEXXX
   call constraint_handler
.LANALYZEXXX
   // code after arithmetic
```

Overflow checks were not added following signed division because these operations result in a division error on IA-32 and generate an interrupt on vector 0.

A. *Performance Study*

The performance of the prototype was assessed using the SPECINT2006 component of the industry standard SPECCPU2006 benchmarks, which provide a meaningful and unbiased metric. SPECINT2006 was compiled using a reference (unmodified) GCC compiler and the modified GCC compiler. The two binaries were executed, and the ratio of their runtime to a known baseline was used to compute a performance ratio.

Higher numbers for the control and analyzable rations in **Error! Reference source not found.** indicate better performance.

TABLE II. SPECINT2006 MACRO-BENCHMARK RUNS

Optimization Level	Control Ratio	Analyzable Ratio	% Slowdown
`-O0`	4.92	4.60	6.96
`-O1`	7.21	6.77	6.50
`-O2`	7.38	6.99	5.58

Because the benchmarks used in SPECINT2006 do overflow, the prototype was modified slightly so that the analysis is performed but programs do not abort on overflow.

Code insertions occur after all optimizations are performed by GCC, so the observed slowdown is not caused by disrupted optimizations. Instead, the slowdown results entirely from the cost of the conditional tests after each arithmetic operation.

Runtime performance could be further improved by eliminating unnecessary tests in cases where value-range analysis can prove that overflow or wrapping is not possible. This technique can be implemented as analyzer advice, where a front-end analyzer provides advice to a back-end compiler. For each input source file, our prototype accepts an analyzer advice file containing either a white list of operations that do not require testing or a black list of operations that do require it. We did not make use of this capability when generating our results because we wanted to establish a baseline prior to introducing any optimizations.

If, after optimization, AIR overhead remains unacceptable, the AIR Integer model can still provide some benefit for testing and then be disabled for deployed code.

B. *Efficacy Study*

For our efficacy study, the JasPer image processing and FFmpeg audio/video processing libraries were instrumented using the modified GCC compiler and fuzz tested.

JasPer includes a software-based implementation of the codec specified in the JPEG-2000 Part-1 standard ISO/IEC 15444-1 [7]. FFmpeg is a multimedia library that supports the encoding and decoding of a wide range of video and audio formats that are supported by a range of container formats.[6] Both these libraries have been included in many widely deployed commercial and non-commercial applications. As a result, vulnerabilities in these libraries are quite severe because they can lead to widespread compromises.

Fuzz testing was performed using zzuf with a seed range of 0.001% to 1%.[7] JasPer was fuzzed for 17.5 hours, resulting in execution of the application 1,802,614 times. FFmpeg was fuzzed for 18.2 hours, resulting in execution of the application 978,060 times.

The JPEG-2000 decoding capabilities of JasPer were targeted in the fuzzing run. Code coverage details were obtained using GCC gcov.[8] 74.4% of the code in `jpc_dec.c` (which contains code for decoding JPEG-2000 streams) has been executed through the use of fuzzing.

The FFmpeg fuzzing run targeted Ogg Theora video decoding. Because of the large number of formats supported by FFmpeg, only 7.3% of the entire FFmpeg codebase was covered during the fuzzing run. However, 93.7% of the code in `vp3.c` (which is used for decoding Theora video) has been executed.

Fig. 1 and Fig. 2 show the defects discovered in JasPer and FFmpeg organized by severity and by the *CERT C Secure Coding Standard* rule that was violated.

[6] http://ffmpeg.org/

[7] http://caca.zoy.org/wiki/zzuf

[8] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

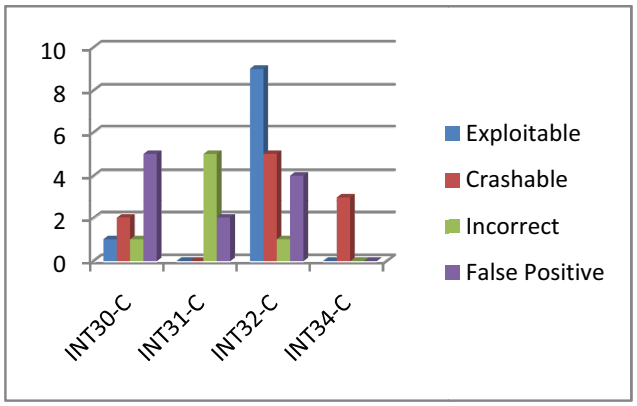

Figure 1: JasPer Defects

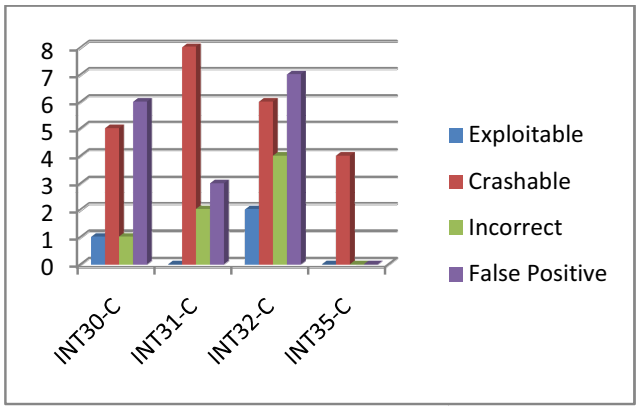

Figure 2: FFmpeg Defects

Violations of the following CERT C Secure Coding Standard rules were discovered:

- INT30-C. Ensure that unsigned integer operations do not wrap
- INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data
- INT32-C. Ensure that operations on signed integers do not result in overflow
- INT34-C. Do not shift a negative number of bits or more bits than exist in the operand
- INT35-C. Evaluate integer expressions in a larger size before comparing or assigning to that size

Each runtime constraint was classified as *exploitable*, *crashable*, *incorrect*, or a *false positive*. Exploitable defects are those that are believed likely to result in an attacker being able to execute arbitrary code. Crashable defects are those that result in a program crash but whose overall security impact otherwise appears limited to a denial-of-service condition. Incorrect defects result in incorrect program output or data corruption, but there is no possibility of crashing or exploiting the program. False positives are traps for overflows or truncations that are not errors because they are harmless for that particular implementation. Technically, these are still defects and may represent undefined behavior or rely on non-portable behaviors. For example, a left shift may be used to extract ASCII-character data packed into an `int` or `long`. While this is undefined behavior and a violation of guideline INT13-C: "Use bitwise operators only on unsigned operands" in the *CERT C Secure Coding Standard* [5], it may not constitute a defect for a given implementation.

Instrumented fuzz testing discovered 10 of a known 12 vulnerabilities in JasPer and had no code coverage for the other 2. For comparison, the Splint [9] static-analysis tool identified those 2. Of the 10 vulnerabilities discovered through fuzzing, Splint missed 4 and identified 6 but not for the reasons they are actually vulnerable. This is not surprising given that Splint issued 468 warnings for 2,000 lines of code.

An example of an exploitable vulnerability by fuzz testing the AIR-integer-instrumented JasPer library occurs in `jas_image_cmpt_create()` where `size` can easily overflow:

```
303: long size;
321: size = cmpt->width_ * cmpt->height_ *
cmpt->cps_;
322: cmpt->stream_ = (inmem) ?
       jas_stream_memopen(0, size) :
       cmpt->jas_stream_tmpfile();
```

In `jas_stream_memopen()`, a `bufsize` less than or equal to 0 is meaningful and indicates that a buffer has been allocated internally and is growable:

```
171: jas_stream_t *jas_stream_memopen(char
*buf, int bufsize)
205:   if (bufsize <= 0) {
206:     obj->bufsize_ = 1024
207:     obj->growable_ = 1;
208:   } else {
209:     obj->bufsize_ = bufsize;
210:     obj->growable_ = 0;
211:   }
```

If `size` overflows in `jas_image_cmpt_create()`, it becomes negative, causing `jas_stream_memopen()` to treat the buffer as if it were internally allocated and growable. Both signed integer overflows on line 321 are diagnosed in violation of INT32-C:

```
jas_image_cmpt_create
src/libjasper/base/jas_image.c:321
0x804c8d3 Signed integer overflow in
multiplication
0x804c8e3 Signed integer overflow in
multiplication
```

Two false positives in JasPer both involved signed left shift of an `unsigned char` cast to `int`. This false positive can be eliminated by casting the `int` value to `unsigned int`. While this is a false positive for vulnerability, it is undefined behavior and a violation of guideline INT13-C: "Use bitwise operators only on unsigned

[9] http://www.splint.org/

operands" in the *CERT C Secure Coding Standard*" [5]. In FFmpeg, harmless unsigned wrapping caused a false positive. While this is a false positive for vulnerability, it is a violation of rule INT30-C.

V. Conclusions

The AIR Integer model produces either a value that is equivalent to a value that would have been obtained using infinitely ranged integers or a runtime-constraint violation. AIR integers can be used for dynamic analysis or as a runtime protection scheme. In either case, no changes are required to the source code. Consequently, the AIR Integer model can be used with legacy systems by compiling C source code in analyzable mode.

At the `O2` optimization level, our compiler prototype showed only a 5.58% slowdown when running the SPECINT2006 macro-benchmark. Although that percentage represents the worst-case performance for AIR integers (because no optimizations were performed in placing checks), it is still low enough for typical applications to enable this feature in deployed systems. AIR integers have also been proven effective in discovering vulnerabilities, crashes, and other defects in the JasPer image processing library and the FFmpeg audio/video processing library during testing with dumb (mutation) fuzzing.

Acknowledgment

We would like to acknowledge the contributions of the following individuals to the research presented in this paper: Chad Dougherty, Ian Lance Taylor, Richard Guenther, David Chisnall, Tzi-cker Chiueh, Huijia Lin, Joseph Myers, Raunak Rungta, Alexey Smirnov, Rob Johnson, and David Brumley. We would also like to acknowledge the contribution of our technical editors and librarians: Pennie Walters, Edward Desautels, Alexa Huth, and Sheila Rosenthal. This research was supported by the U.S. Department of Defense (DoD), the U.S. Department of Homeland Security (DHS) National Cyber Security Division (NCSD), and CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 from the Army Research Office.

References

[1] Heffley, J. and Meunier, P. 2004. "Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?" Proc. 37th Annual Hawaii international Conference on System Sciences (HICSS'04) - Track 9 - Volume 9 (January 05 - 08, 2004). HICSS. IEEE Computer Society, Washington, DC, 90277.

[2] Seacord, R. C. Secure Coding in C and C++ (SEI Series in Software Engineering). Addison-Wesley Professional, 2005.

[3] Christy, Steve and Martin, Robert A. Vulnerability Type Distributions in CVE. Document Version: 1.1. May 22, 2007. http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf

[4] Solar Designer. JPEG COM Marker Processing Vulnerability in Netscape Browsers. OpenWall Project, July 2000. http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt

[5] Seacord, R. The CERT C Secure Coding Standard. 1st. Addison-Wesley Professional, 2008.

[6] Lions, J. L. ARIANE 5 Flight 501 Failure Report. Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[7] Michael D. Adams. JasPer Software Reference Manual (Version 1.900.0). December 2006. http://www.ece.uvic.ca/~mdadams/jasper/jasper.pdf

[8] Brumley, David; Chiueh, Tzi-cker; Johnson, Robert; Lin, Huijia; and Song, Dawn. "RICH: Automatically Protecting Against Integer-Based Vulnerabilities." Proc. NDSS, San Diego, CA, Feb. 2007. Reston, VA: ISOC, 2007. http://www.cs.berkeley.edu/~dawnsong/papers/efficient_detection_integer-based_attacks.pdf

[9] Jones, Larry. WG14 N1401 Committee Draft ISO/IEC 9899:201x. International Standards Organization, November 24, 2009.

[10] Hedquist, Barry. ISO/JTC1/SC22/WG14 AND INCITS PL22.11 MARCH/APRIL MEETING. MINUTES). Washington, D.C.: International Committee for Information Technology Standards, 2009. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1421.pdf

[11] International Standards Organization. ISO/IEC TR 24731. Extensions to the C Library, — Part I: Bounds-checking interfaces. Geneva, Switzerland: International Standards Organization, April 2006.

[12] Plum, Thomas and Keaton, David M. "Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool." Proc. Workshop on Software Security Assurance Tools, Techniques, and Metrics, Long Beach, CA, November 7-8, 2005. Washington, D.C.: U.S. National Institute of Standards and Technology (NIST), 2005. http://samate.nist.gov/docs/NIST_Special_Publication_500-265.pdf

[13] International Standards Organization. Programming Languages—C++, Final Committee Draft, ISO/IEC JTC1 SC22 WG21 N3092. International Standards Organization, March 2010.

[14] Intel, Corp. The IA-32 Intel Architecture Software Developer's Manual. Denver, CO: Intel Corp., 2004.