Appeared in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, June 2012.

Understanding Integer Overflow in C/C++

Will Dietz,[*] Peng Li,[†] John Regehr,[†] and Vikram Adve[*]
[*]*Department of Computer Science*
University of Illinois at Urbana-Champaign
{wdietz2,vadve}@illinois.edu
[†]*School of Computing*
University of Utah
{peterlee,regehr}@cs.utah.edu

***Abstract*—Integer overflow bugs in C and C++ programs are difficult to track down and may lead to fatal errors or exploitable vulnerabilities. Although a number of tools for finding these bugs exist, the situation is complicated because not all overflows are bugs. Better tools need to be constructed—but a thorough understanding of the issues behind these errors does not yet exist. We developed IOC, a dynamic checking tool for integer overflows, and used it to conduct the first detailed empirical study of the prevalence and patterns of occurrence of integer overflows in C and C++ code. Our results show that intentional uses of wraparound behaviors are more common than is widely believed; for example, there are over 200 distinct locations in the SPEC CINT2000 benchmarks where overflow occurs. Although many overflows are intentional, a large number of accidental overflows also occur. Orthogonal to programmers' intent, overflows are found in both well-defined and undefined flavors. Applications executing undefined operations can be, and have been, broken by improvements in compiler optimizations. Looking beyond SPEC, we found and reported undefined integer overflows in SQLite, PostgreSQL, SafeInt, GNU MPC and GMP, Firefox, GCC, LLVM, Python, BIND, and OpenSSL; many of these have since been fixed. Our results show that integer overflow issues in C and C++ are subtle and complex, that they are common even in mature, widely used programs, and that they are widely misunderstood by developers.**

***Keywords*-integer overflow; integer wraparound; undefined behavior**

I. Introduction

Integer numerical errors in software applications can be insidious, costly, and exploitable. These errors include overflows, underflows, lossy truncations (e.g., a cast of an `int` to a `short` in C++ that results in the value being changed), and illegal uses of operations such as shifts (e.g., shifting a value in C by at least as many positions as its bitwidth). These errors can lead to serious software failures, e.g., a truncation error on a cast of a floating point value to a 16-bit integer played a crucial role in the destruction of Ariane 5 flight 501 in 1996. These errors are also a source of serious vulnerabilities, such as integer overflow errors in OpenSSH [1] and Firefox [2], both of which allow attackers to execute arbitrary code. In their 2011 report MITRE places integer overflows in the "Top 25 Most Dangerous Software Errors" [3].

Detecting integer overflows is relatively straightforward by using a modified compiler to insert runtime checks. However, reliable detection of overflow *errors* is surprisingly difficult because overflow behaviors are not always bugs. The low-level nature of C and C++ means that bit- and byte-level manipulation of objects is commonplace; the line between mathematical and bit-level operations can often be quite blurry. Wraparound behavior using unsigned integers is legal and well-defined, and there are code idioms that deliberately use it. On the other hand, C and C++ have undefined semantics for signed overflow and shift past bitwidth: operations that are perfectly well-defined in other languages such as Java. C/C++ programmers are not always aware of the distinct rules for signed vs. unsigned types in C, and may naïvely use signed types in intentional wraparound operations.[1] If such uses were rare, compiler-based overflow detection would be a reasonable way to perform integer error detection. If it is not rare, however, such an approach would be impractical and more sophisticated techniques would be needed to distinguish *intentional* uses from *unintentional* ones.

Although it is commonly known that C and C++ programs contain numerical errors and also benign, deliberate use of wraparound, it is unclear how common these behaviors are and in what patterns they occur. In particular, there is little data available in the literature to answer the following questions:

1) How common are numerical *errors* in widely-used C/C++ programs?
2) How common is use of intentional wraparound operations with signed types—which has undefined behavior—relying on the fact that today's compilers may compile these overflows into *correct* code? We refer to these overflows as "time bombs" because they remain latent until a compiler upgrade turns them into observable errors.
3) How common is *intentional* use of well-defined

[1] In fact, in the course of our work, we have found that even experts writing *safe integer libraries* or *tools to detect integer errors* are not always fully aware of the subtleties of C/C++ semantics for numerical operations.

wraparound operations on unsigned integer types?

Although there have been a number of papers on tools to detect numerical errors in C/C++ programs, *no previous work we know of has explicitly addressed these questions, or contains sufficient data to answer any of them.* The closest is Brumley et al.'s work [4], which presents data to motivate the goals of the tool and also to evaluate false positives (invalid error reports) due to intentional wraparound operations. As discussed in Section V, that paper only tangentially addresses the third point above. We study all of these questions systematically.

This paper makes the following primary contributions. First, we developed Integer Overflow Checker (IOC), an open-source tool that detects both undefined integer behaviors as well as well-defined wraparound behaviors in C/C++ programs.[2] IOC is an extension of the Clang compiler for C/C++ [5]. Second, we present the first detailed, empirical study—based on SPEC 2000, SPEC 2006, and a number of popular open-source applications—of the prevalence and patterns of occurrence of numerical overflows in C/C++ programs. Part of this study includes a manual analysis of a large number of *intentional* uses of wraparound in a subset of the programs. Third, we used IOC to discover previously unknown overflow errors in widely-used applications and libraries, including SQLite, PostgreSQL, BIND, Firefox, OpenSSL, GCC, LLVM, the SafeInt library, the GNU MPC and GMP libraries, Python, and PHP. A number of these have been acknowledged and fixed by the maintainers (see Section IV).

The key findings from our study of overflows are as follows: First, all four combinations of intentional and unintentional, well-defined and undefined integer overflows occur frequently in real codes. For example, the SPEC CINT2000 benchmarks had over 200 distinct occurrences of intentional wraparound behavior, for a wide range of different purposes. Some uses for intentional overflows are well-known, such as hashing, cryptography, random number generation, and finding the largest representable value for a type. Others are less obvious, e.g., inexpensive floating point emulation, signed negation of `INT_MIN`, and even ordinary multiplication and addition. We present a detailed analysis of examples of each of the four major categories of overflow. Second, overflow-related issues in C/C++ are very subtle and we find that even experts get them wrong. For example, the latest revision of Firefox (as of Sep 1, 2011) contained integer overflows *in the library that was designed to handle untrusted integers safely* in addition to overflows in its own code. More generally, we found very few mature applications that were completely free of integer numerical errors. This implies that there is probably little hope of eliminating overflow errors in large code bases without sophisticated tool support. However, these tools cannot simply distinguish errors from benign operations by checking rules from the ISO language standards. Rather, tools will have to use highly sophisticated techniques and/or rely on manual intervention (e.g., annotations) to distinguish intentional and unintentional overflows.

Table I
EXAMPLES OF C/C++ INTEGER OPERATIONS AND THEIR RESULTS

Expression	Result
`UINT_MAX+1`	`0`
`LONG_MAX+1`	undefined
`INT_MAX+1`	undefined
`SHRT_MAX+1`	`SHRT_MAX+1` if `INT_MAX`>`SHRT_MAX`, otherwise undefined
`char c = CHAR_MAX; c++`	varies[1]
`-INT_MIN`	undefined[2]
`(char)INT_MAX`	commonly `-1`
`1<<-1`	undefined
`1<<0`	`1`
`1<<31`	commonly `INT_MIN` in ANSI C and C++98; undefined in C99 and C++11[2,3]
`1<<32`	undefined[3]
`1/0`	undefined
`INT_MIN%-1`	undefined in C11, otherwise undefined in practice

[1] The question is: Does `c` get "promoted" to `int` before being incremented? If so, the behavior is well-defined. We found disagreement between compiler vendors' implementations of this construct.
[2] Assuming that the `int` type uses a two's complement representation
[3] Assuming that the `int` type is 32 bits long

II. Overflow in C and C++

Mathematically, n-bit two's complement arithmetic is congruent, modulo 2^n, to n-bit unsigned arithmetic for addition, subtraction, and the n least significant bits in multiplication; both kinds of arithmetic "wrap around" at multiples of 2^n. On modern processors, integer overflow is equally straightforward: n-bit signed and unsigned operations both have well-defined behavior when an operation overflows: the result wraps around and condition code bits are set appropriately. In contrast, integer overflows in C/C++ programs are subtle due to a combination of complex and counter-intuitive rules in the language standards, non-standards-conforming compilers, and the tendency of low-level programs to rely on non-portable behavior. Table I contains some C/C++ expressions illustrating cases that arise in practice. There are several issues; to clarify them we make a top-level distinction between *well-defined* (albeit perhaps non-portable) and *undefined* operations.

A. Well-Defined Behaviors

Some kinds of unsigned integer arithmetic uses well-defined and portable wraparound behavior, with two's complement semantics [6]. Thus, as Table I indicates, `UINT_MAX+1` must evaluate to zero in every conforming C and C++ implementation. nOf course, even well-defined semantics can lead to logic errors, for example if a developer naïvely assumes that $x + 1$ is larger than x.

[2] IOC is available at http://embed.cs.utah.edu/ioc/

Listing 1. Source for `overflow.c` referred to in the text

```
1 int foo (int x) {
2   return (x+1) > x;
3 }
4
5 int main (void) {
6   printf ("%d\n", (INT_MAX+1) > INT_MAX);
7   printf ("%d\n", foo(INT_MAX));
8   return 0;
9 }
```

Many unsigned integer overflows in C and C++ are well-defined, but *non-portable*. For example `0U-1` is well-defined and evaluates to `UINT_MAX`, but the actual value of that constant is *implementation defined*: it can be relied upon, but only within the context of a particular compiler and platform. Similarly, the `int` type in C99 is not required to hold values in excess of 32,767, nor does it have to be based on a two's complement representation.

B. Undefined Behaviors

Some kinds of integer overflow are undefined, and these kinds of behavior are especially problematic. According to the C99 standard, undefined behavior is

> "behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements."

In Internet parlance:[3]

> "When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose."

Our experience is that many developers fail to appreciate the full consequences of this. The rest of this section examines these consequences.

1) Silent Breakage: A C or C++ compiler may exploit undefined behavior in optimizations that silently break a program. For example, a routine refactoring of Google's Native Client software accidentally caused `1<<32` to be evaluated in a security check.[4] The compiler—at this point under no particular obligation—simply turned the safety check into a nop. Four reviewers failed to notice the resulting vulnerability.

Another illuminating example is the code in Listing 1. In this program, the same computation (`(INT_MAX+1)` `>INT_MAX`) is performed twice with two different idioms. Recent versions of GCC, LLVM, and Intel's C compiler, invoked at the `-O2` optimization level, all print a 0 for the first value (line 6) and a 1 for the second (line 7). In other words, each of these compilers considers `INT_MAX+1` to be both larger than `INT_MAX` and also not larger, at the same optimization level, depending on incidental structural features of the code. The point is that when programs execute undefined operations, optimizing compilers may silently break them in non-obvious and not necessarily consistent ways.

2) Time Bombs: Undefined behavior also leads to *time bombs*: code that works under today's compilers, but breaks unpredictably in the future as optimization technology improves. The Internet is rife with stories about problems caused by GCC's ever-increasing power to exploit signed overflows. For example, in 2005 a principal PostgreSQL developer was annoyed that his code was broken by a recent version of GCC:[5]

> It seems that gcc is up to some creative reinterpretation of basic C semantics again; specifically, you can no longer trust that traditional C semantics of integer overflow hold ...

This highlights a fundamental and pervasive misunderstanding: the compiler was not "reinterpreting" the semantics but rather was beginning to take advantage of leeway explicitly provided by the C standard.

In Section IV-E we describe a time bomb in SafeInt [7]: a library that is itself intended to help developers avoid undefined integer overflows. This operation, *until recently*, was reliably compiled by GCC (and other compilers) into code that did not have observable errors. However, the upcoming version of GCC (4.7) exposes the error, presumably because it optimizes the code more aggressively. We discovered this error using IOC and reported it to the developers, who fixed it within days [8].

3) Illusion of Predictability: Some compilers, at some optimization levels, have predictable behavior for some undefined operations. For example, C and C++ compilers typically give two's complement semantics to signed overflow when aggressive optimizations are disabled. It is, however, unwise to rely on this behavior, because it is not portable across compilers or indeed across different versions of the same compiler.

4) Informal Dialects: Some compilers support stronger semantics than are mandated by the standard. For example, both GCC and Clang (an LLVM-based C/C++/Objective-C compiler) support a `-fwrapv` command line flag that forces signed overflow to have two's complement behavior. In fact, the PostgreSQL developers responded to the incident above by adding `-fwrapv` to their build flags. They are now, in effect, targeting a non-standard dialect of C.

5) Non-Standard Standards: Some kinds of overflow have changed meaning across different versions of the standards. For example, `1<<31` is implementation-defined in ANSI C and C++98, while being explicitly undefined by C99 and C11 (assuming 32-bit ints). Our experience is that awareness of this particular rule among C and C++ programmers is low.

A second kind of non-standardization occurs with constructs such as `INT_MIN%-1` which is—by our reading—well

[3] http://catb.org/jargon/html/N/nasal-demons.html

[4] http://code.google.com/p/nativeclient/issues/detail?id=245

[5] http://archives.postgresql.org/pgsql-hackers/2005-12/msg00635.php

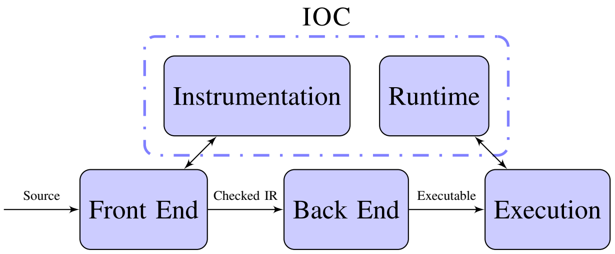

Figure 1. Architecture of IOC

defined in ANSI C, C99, C++98, and C++11. However, we are not aware of a C or C++ compiler that reliably returns the correct result, zero, for this expression. The problem is that on architectures including x86 and x86-64, correctly handling this case requires an explicit check in front of every % operation. The C standards committee has recognized the problem and C11 explicitly makes this case undefined.

III. Tool Design and Implementation

IOC, depicted in Fig. 1, has two main parts: a compile-time instrumentation transformation and a runtime handler. The transformation is a compiler pass that adds inline numerical error checks; it is implemented as a ∼1600 LOC extension to Clang [5], the C/C++ frontend to LLVM [9]. IOC's instrumentation is designed to be semantically transparent for programs that conform to the C or C++ language standards, except in the case where a user requests additional checking for conforming but error-prone operations, e.g., wraparound with unsigned integer types. The runtime library is linked into the compiler's output and handles overflows as they occur; it is ∼900 lines of C code.

A. *Where to Put the Instrumentation Pass?*

The transformation operates on the Abstract Syntax Tree (AST) late in the Clang front end—after parsing, type-checking, and implicit type conversions have been performed. This is an appropriate stage for inserting checks because full language-level type information is available, but the compiler has not yet started throwing away useful information as it does during the subsequent conversion into the flat LLVM intermediate representation (IR).

In a previous iteration of IOC we encoded the required high-level information into the IR (using IR metadata), allowing the transformation to be more naturally expressed as a compiler pass. Unfortunately, this proved to be unreliable and unnecessarily complicated, due to requiring a substantial amount of C-level type information in the IR in order to support a correct transformation. The original transformation was further complicated by the lack of a one-to-one mapping between IR and AST nodes. Also, some important operations (such as signed to unsigned casts) don't exist at the IR level. In short, it is much less error-prone to do the instrumentation in the frontend where all the required information is naturally available.

B. *Overflow Checks*

Finding overflows in shift operations is straightforward: operand values are bounds-checked and then, if the checks pass, the shift is performed. Checking for overflow in arithmetic operations is trickier; the problem is that a checked n-bit addition or subtraction requires $n+1$ bits of precision and a checked n-bit multiplication requires $2n$ bits of precision. Finding these extra bits can be awkward. There are basically three ways to detect overflow for an operation on two signed integers s_1 and s_2.

1) **Precondition test.** It is always possible to test whether an operation will wrap without actually performing the operation. For example, signed addition will wrap if and only if this expression is true:

$$((s_1 > 0) \land (s_2 > 0) \land (s_1 > (\texttt{INT\_MAX} - s_2))) \lor$$
$$((s_1 < 0) \land (s_2 < 0) \land (s_1 < (\texttt{INT\_MIN} - s_2)))$$

In pseudocode:

```
if (!precondition) then
  call failure handler
endif
result = s1 op s2
```

2) **CPU flag postcondition test.** Most processors contain hardware support for detecting overflow: following execution of an arithmetic operation, condition code flags are set appropriately. In the general case, it is problematic to inspect processor flags in portable code, but LLVM supports a number of intrinsic functions where, for example, an addition operation returns a structure containing both the result and an overflow flag. The LLVM backends, then, emit processor-specific code that accesses the proper CPU flag. In pseudocode:

```
(result, flag) = s1 checked_op s2
if (flag) then
  call failure handler
endif
```

3) **Width extension postcondition test.** If an integer datatype with wider bitwidth than the values being operated on is available, overflow can be detected in a straightforward way by converting s_1 and s_2 into the wider type, performing the operation, and checking whether the result is in bounds with respect to the original (narrower) type. In pseudocode:

```
result = extend(s1) op extend(s2)
if (result < MIN || result > MAX) then
  call failure handler
endif
```

IOC supports both the precondition test and the CPU flag postcondition test; width extension seemed unlikely to be better than these options due to the expense of emulating 64-bit and 128-bit operations. Initially we believed that the CPU flag postcondition checks would be far more efficient but this proved not to be the case. Rather, as shown in Section III-D, using the flag checks has an uneven effect on performance. The explanation can be found in the interaction between the overflow checks and the compiler's optimization passes. The precondition test generates far too many operations, but they are operations that can be aggressively optimized by LLVM. On the other hand, the LLVM intrinsics supporting the flag-based postcondition checks are recognized and exploited by relatively few optimization passes, causing much of the potential performance gain due to this approach to be unrealized.

C. Runtime Library

To produce informative error messages, IOC logs the source-code location corresponding to each inserted check, including the column number where the operator appeared. (Operating on the AST instead of the LLVM IR makes such logging possible.) Thus, users can disambiguate, for example, which shift operator overflowed in a line of code containing multiple shift operators. Also in service of readable error messages, IOC logs the types and values of the arguments passed to the operator; this is important for operators with multiple modes of failure, such as shift. For example, an error we found in OpenSSL was reported as:

```
<lhash.c, (464:20)> : Op: >>, Reason :
Unsigned Right Shift Error: Right operand is negative or is
greater than or equal to the width of the promoted left operand,
BINARY OPERATION: left (uint32): 4103048108 right (uint32): 32.
```

Based on the value of an environment variable, the IOC failure handler can variously send its output to `STDOUT`, to `STDERR`, to the syslog daemon, or simply discard the output. The syslog option is useful for codes that are sensitive to changes in their `STDOUT` and `STDERR` streams, and for codes such as daemons that are invoked in execution environments where capturing their output would be difficult.

Finally, to avoid overwhelming users with error messages, the fault handler uses another environment variable to specify the maximum number of times an overflow message from any particular program point will be printed.

D. Runtime Overhead of Integer Overflow Checking

To evaluate the effect of IOC on programs' runtime, we compiled SPEC CPU 2006 in four ways. First, a baseline compilation using Clang with optimization options set for maximum performance. Second, checking for undefined integer overflows (shifts and arithmetic) using precondition checks. Third, checking for undefined integer overflows (shifts and arithmetic) using the CPU flag postcondition test. Finally, checking for all integer overflows including unsigned overflow and value loss via sign conversion and truncation.

We then ran the benchmarks on a 3.4 GHz AMD Phenom II 965 processor, using their "ref" inputs—the largest input data, used for reportable SPEC runs—five times and used the median runtime. We configured the fault handler to return immediately instead of logging overflow behaviors. Thus, these measurements do not include I/O effects due to logging, but they do include the substantial overhead of marshaling the detailed failure information that is passed to the fault handler.

Table II
TAXONOMY OF INTEGER OVERFLOWS IN C AND C++ WITH REFERENCES TO DETAILED DISCUSSION OF EXAMPLES

	undefined behavior e.g. signed overflow, shift error, divide by zero	**defined behavior** e.g. unsigned wrapround, signed wraparound with `-fwrapv`
intentional	*Type 1:* design error, may be a "time bomb" § IV-C3, IV-C9	*Type 2:* no error, but may not be portable § IV-C2, IV-C5, IV-C8
unintentional	*Type 3:* implementation error, may be a "time bomb" § IV-C4	*Type 4:* implementation error § IV-C1, IV-C6

For undefined behavior checking using precondition checks, slowdown relative to the baseline ranged from −0.5%–191%. In other words, from a tiny accidental speedup to a 3X increase in runtime. The mean slowdown was 44%. Using flag-based postcondition checks, slowdown ranged from 0.4%–95%, with a mean of 30%. However, the improvement was not uniform: out of the 21 benchmark programs, only 13 became faster due to the IOC implementation using CPU flags. Full integer overflow checking using precondition checks incurred a slowdown of 0.2%–195%, with a mean of 51%.

IV. INTEGER OVERFLOW STUDY

This section presents the qualitative and quantitative results of our study of overflow behaviors in C and C++ applications.

A. Limitations of the Study

There are necessarily several limitations in this kind of empirical study. Most important, because IOC is based on dynamic checking, bugs not exercised by our inputs will not be found. In this sense, our results likely understate the prevalence of integer numerical errors as well as the prevalence of intentional uses of wraparound in these programs. A stress testing strategy might uncover more bugs.

Second, our methodology for distinguishing intentional from unintentional uses of wraparound is manual and subjective. The manual effort required meant that we could only study a subset of the errors: we focused on the errors in the SPEC CINT2000 benchmarks for these experiments. For the other experiments, we study a wider range of programs.

B. A Taxonomy for Overflows

Table II summarizes our view of the relationship between different integer overflows in C/C++ and the correctness of

Listing 2. Well-defined but incorrect guard for memcpy in 164.gzip

```
/* (this test assumes unsigned comparison) */
if ( w - d  >= e)
{
  memcpy(slide + w, slide + d, e);
  w += e;
  d += e;
}
```

software containing these behaviors. Only Type 2 overflows do not introduce numerical errors into carefully written C/C++ software. Using IOC, we have found examples of software errors of Types 1, 3, and 4, as well as correct uses of Type 2. The section numbers in the table are forward references to discussions of bugs in the next section. We found many additional examples of each type of error, but lack space to discuss them in detail.

C. Wraparound and Overflow in SPEC CINT 2000

To investigate the prevalence of, and use-cases for, overflows and wraparounds, we examined SPEC CINT2000 in detail. The SPEC benchmark suites each contain a carefully selected set of C and C++ programs, designed to be representative of a wide range of real-world software (and many like GCC, bzip2, and povray, are taken from widely used applications). Moreover, since they are primary performance benchmarks for both compilers and architectures, these benchmarks have been compiled and tested with most optimizing compilers, making them especially good case studies.

We ran the SPEC benchmarks' "ref" data sets. Using IOC, we investigated every addition, subtraction, multiplication, and division overflow in an attempt to understand what it is that developers are trying to accomplish when they put overflows into their code.

Our findings are shown in Table III and described below. This benchmark suite consists of 12 medium-sized programs (2.5–222 KLOC), eight of which executed integer overflows while running on their reference input data sets.

Note: Real C code is messy. We have cleaned up the SPEC code examples slightly when we deemed this to improve readability and to not change the sense of the code.

1) 164.gzip: IOC reported eight wraparounds in this benchmark, all using well-defined unsigned operations. Of course, even well-defined operations can be wrong; we discuss an example of particular interest, shown in Listing 2. The POSIX `memcpy` function is undefined if its source and target memory regions overlap. To guard against invoking `memcpy` incorrectly, the code checks that `e` (number of bytes copied) is less than the distance between `w` and `d` (both offsets into a memory region). If this check fails, a slower memory copy that correctly handles overlap is invoked.

However, when $d \geq w$ an unsigned overflow occurs, resulting in an integer that is much greater than any potential value for `e`, causing the safety check to pass even when the source and target regions overlap. This overflow was

Table III
INTEGER WRAPAROUNDS[1] REPORTED IN SPEC CINT2000

Name	Location[2]	Op[3]	Description
164.gzip	bits.c(136:18)	$+_u$	Bit manipulation
164.gzip	bits.c(136:28)	$-_u$	Bit manipulation
164.gzip	deflate.c(540:21)	$-_u$	Unused
164.gzip	inflate.c(558:13)	$-_u$	Bit manipulation
164.gzip	inflate.c(558:22)	$-_u$	Bit manipulation
164.gzip	inflate.c(566:15)	$-_u$	Incorrect `memcpy` guard (Listing 2)
164.gzip	trees.c(552:25)	$+_u$	Type promotion
164.gzip	trees.c(990:38)	$-_u$	Type promotion
175.vpr	route.c(229:19)	$-_u$	Hash
175.vpr	util.c(463:34)	$*_u$	RNG[4](Listing 3)
175.vpr	util.c(463:39)	$+_u$	RNG[4]
175.vpr	util.c(484:34)	$*_u$	RNG[4]
175.vpr	util.c(484:39)	$+_u$	RNG[4]
176.gcc	combine.c × 6	$-_s$	Find `INT_MAX` (Listing 4)
176.gcc	cse.c × 5	$+_u$	Hash
176.gcc	expmed.c × 15	$\pm_{u,s}$	Bit manipulation
176.gcc	expmed.c(2484:13)	$*_u$	Inverse of $x \mod 2^n$
176.gcc	expmed.c(2484:18)	$-_u$	Inverse of $x \mod 2^n$
176.gcc	expmed.c(2484:21)	$*_u$	Inverse of $x \mod 2^n$
176.gcc	insn-emit.c(3613:5)	$+_u$	Range check
176.gcc	loop.c(1611:19)	$*_s$	Cost calculation bug (Listing 5)
176.gcc	m88k.c(127:44)	$-_u$	Bit manipulation (Listing 6)
176.gcc	m88k.c(128:20)	$+_u$	Bit manipulation
176.gcc	m88k.c(128:20)	$-_u$	Bit manipluation
176.gcc	m88k.c(888:13)	$+_u$	Range check
176.gcc	m88k.c(1350:38)	$+_u$	Range check
176.gcc	m88k.c(2133:9)	$+_u$	Range check
176.gcc	obstack.c(271:49)	$-_u$	Type promotion artifact
176.gcc	real.c(1909:35)	$-_u$	Emulating addition
176.gcc	real.c(2149:18)	$*_s$	Overflow check
176.gcc	rtl.c(193:16)	$+_u$	Allocation calc bug (Listing 7)
176.gcc	rtl.c(193:16)	$*_u$	Allocation calc bug
176.gcc	rtl.c(216:19)	$*_u$	Allocation calc bug
176.gcc	rtl.c(216:5)	$+_u$	Allocation calc bug
176.gcc	stor-layout.c(1040:7)	$-_s$	Find largest sint
176.gcc	tree.c(1222:15)	$*_s$	Hash
176.gcc	tree.c(1585:37)	$-_s$	Bit manipulation
176.gcc	varasm.c(2255:15)	$*_s$	Hash
186.crafty	evaluate.c(594:7)	$-_u$	Bit manipulation
186.crafty	evaluate.c(595:7)	$-_u$	Bit manipulation
186.crafty	iterate.c(438:16)	$*_s$	Statistic bug (`100*a/(b+1)`)
186.crafty	utility.c(813:14)	$+_u$	RNG[4]
197.parser	and.c × 6	$+_{u,s}$	Hash
197.parser	fast-match.c(101:17)	$+_u$	Hash
197.parser	fast-match.c(101:8)	$+_s$	Hash
197.parser	parse.c × 10	$+_{u,s}$	Hash
197.parser	prune.c × 7	$+_{u,s}$	Hash
197.parser	xalloc.c(68:40)	$*_u$	Compute `SIZE_MAX >> 1` (Listing 8)
197.parser	xalloc.c(70:19)	$+_u$	Compute `SIZE_MAX >> 1`
253.perlbmk	hv.c × 7	$*_u$	Hash
253.perlbmk	md5c.c × 68	$+_u$	Hash
253.perlbmk	pp.c(1958:14)	$-_u$	Missing cast
253.perlbmk	pp.c(1971:6)	$+_u$	Missing cast
253.perlbmk	regcomp.c(353:26)	$+_s$	Unused
253.perlbmk	regcomp.c(462:21)	$+_s$	Unused
253.perlbmk	regcomp.c(465:21)	$+_s$	Unused
253.perlbmk	regcomp.c(465:34)	$*_s$	Unused
253.perlbmk	regcomp.c(465:9)	$+_s$	Unused
253.perlbmk	regcomp.c(584:23)	$+_s$	Unused
253.perlbmk	regcomp.c(585:13)	$+_s$	Unused
253.perlbmk	sv.c(2746:19)	$-_u$	Type promotion artifact
254.gap	eval.c(366:34)	$*_s$	Overflow check requiring -fwrapv
254.gap	idents.c × 4	$*_u$	Hash
254.gap	integer.c × 28	$*_s$	Overflow check requiring -fwrapv
254.gap	integer.c × 4	$+_s$	Overflow check requiring -fwrapv
254.gap	integer.c × 4	$-_s$	Overflow check requiring -fwrapv
255.vortex	ut.c(1029:17)	$*_u$	RNG[4]

[1] Only Add, Sub, Mul, and Div errors were checked in this experiment. (No Div overflows were found)

[2] Source, and line:column. For space, we summarize frequent ones as '× n'.

[3] Operation Type(s), and Signed/Unsigned.

[4] (Pseudo-)Random Number Generation.

Listing 3. Correct wraparound in a random number generator in 175.vpr

```
#define IA 1103515245u
#define IC 12345u
#define IM 2147483648u

static unsigned int c_rand = 0;

/* Creates a random integer [0...imax] (inclusive) */
int my_irand (int imax) {
  int ival;
  /* c_rand = (c_rand * IA + IC) % IM; */
  c_rand = c_rand * IA + IC; // Use overflow to wrap
  ival = c_rand & (IM - 1);  /* Modulus */
  ival = (int) ((float) ival * (float) (imax + 0.999)
        / (float) IM);
  return ival;
}
```

Listing 4. Undefined overflow in 176.gcc to compute `INT_MAX`

```
/* (unsigned) <= 0x7fffffff is equivalent to >= 0. */
else if (const_op == ((HOST_WIDE_INT) 1 << (
      mode_width - 1)) - 1)
{
  const_op = 0, op1 = const0_rtx;
  code = GE;
}
```

Listing 5. Overflow in loop hoisting cost heuristic in 176.gcc

```
if (moved_once[regno])
{
  insn_count *= 2;
  ...
  if (already_moved[regno]
    || (threshold * savings * m->lifetime) >=
        insn_count
    || (m->forces && m->forces->done
      && n_times_used[m->forces->regno] == 1))
  {
    ...
```

Listing 6. Correct use of wraparound in bit manipulation in 176.gcc

```
#define POWER_OF_2_or_0(I) \
  (((I) & ((unsigned)(I) - 1)) == 0)

int
integer_ok_for_set (value)
     register unsigned value;
{
  /* All the "one" bits must be contiguous.
   * If so, MASK + 1 will be a power of two or zero.
      */
  register unsigned mask = (value | (value - 1));
  return (value && POWER_OF_2_or_0 (mask + 1));}
```

reported by IOC and while investigating the report we discovered this potential bug. Fortunately, the version of gzip used in this experiment is rather old (based on 1.2.4) and this issue has already been reported and fixed upstream[6] as of version 1.4. Note that this bug existed in gzip as of 1993 and wasn't fixed until 2010. Furthermore, the initial fix was overkill and was later fixed[7] to be the proper minimal condition to protect the `memcpy`. This illustrates the subtlety of overflow errors, and serves as a good example of well-defined overflows leading to logic errors. In terms of the taxonomy in Table II, this wraparound is Type 4.

2) 175.vpr: This benchmark had four unsigned wraparounds caused by two similar implementations of random number generation. As shown in Listing 3, the developers documented their intentional use of unsigned integer wraparound. These wraparounds are well-defined and benign, and represent an important idiom for high-performance code. They are Type 2.

3) 176.gcc: This benchmark had overflows at 48 static sites, some undefined and some well-defined. Listing 4 shows code that tries to compute the largest representable signed integer. `HOST_WIDE_INT` is an `int` and `mode_width` is 32, making the expression equivalent to `(1 << 31) - 1`. This expression is undefined in two different ways. First, in C99 it is illegal to shift a "1" bit into or past the sign bit. Second—assuming that the shift operation successfully computes `INT_MIN`—the subtraction underflows. In our experience, this idiom is common in C and C++ code. Although compilers commonly give it the semantics that programmers expect, it should be considered to be a time bomb. A better way to compute `INT_MAX` is using unsigned arithmetic. This overflow is Type 1.

4) 176.gcc: Listing 5 shows an undefined overflow that may cause GCC to generate suboptimal code *even in the case where the signed overflow is compiled to a wraparound behavior*. The variable `insn_count` is used as a score in a heuristic that decides whether to move a register outside of a loop. When it overflows, this score inadvertently goes from being very large to being small, potentially affecting code generation. This overflow is Type 3.

5) 176.gcc: Listing 6 shows code that determines properties about the integer passed in at a bit level. In doing so, it invokes various arithmetic operations (subtraction, addition, and another subtraction in the `POWER_OF_2_or_0` macro) that wrap around. These are all on unsigned integers and are carefully constructed to test the correct bits in the integers, so all of these wraparounds are benign. This example is a good demonstration of safe bit-level manipulation of integers, a popular cause of wraparound in programs. This overflow is Type 2.

6) 176.gcc: In Listing 7 we see an allocation wrapper function that allocates a vector of n elements. It starts with 16 bytes and then adds $(n-1)*8$ more to fill out the array, since the beginning `rtvec_def` struct has room for 1 element by default. This works well enough (ignoring the type safety violations) for most values of n, but has curious

Listing 7. Wraparound in an allocation function in 176.gcc

```
/* Allocate a zeroed rtx vector of N elements */
rtvec rtvec_alloc (int n) {
  rtvec rt;
  int i;

  rt = (rtvec) obstack_alloc (rtl_obstack,
               sizeof (struct rtvec_def)
               + (( n - 1) * sizeof (rtunion)));
  ...
  return rt;
}
```

[6] http://git.savannah.gnu.org/gitweb/?p=gzip.git;a=commit;h=b9e94c93df914bd1d9eec9f150b2e4e00702ae7b

[7] http://git.savannah.gnu.org/gitweb/?p=gzip.git;a=commit;h=17822e2cab5e47d73f224a688be8013c34f990f7

Listing 8. Compute `SIZE_MAX >> 1` in 197.parser

```
void initialize_memory (void) {
  SIZET i, j;
  ...
  for (i=0, j=1; i < j; i = j, j = (2*j+1))
    largest_block = i;
  largest_block &= ALIGNMENT_MASK;
  // must have room for a nuggie too
  largest_block += -sizeof(Nuggie);
```

behavior when $n = 0$. Of course, since we are using a dynamic checker, it actually is called with $n = 0$ during a SPEC benchmarking run.

First, consider this code after the sizeof operators are resolved and the promotion rules are applied: `16 + ((unsigned)(n-1)) * ((unsigned)8)`. When $n = 0$, we immediately see the code casting -1 to unsigned, which evaluates to `UINT_MAX`, or $2^{32} - 1$. The result is then multiplied by eight, which overflows with a result of $2^{32} - 8$. Finally, the addition is evaluated, which produces the final result of 8 after wrapping around again.

Although the overflow itself is benign, its consequences are unfortunate. Only eight bytes are allocated but the `rtvec_def` structure is 16 bytes. Any attempt to copy it by value will result in a memory safety error, perhaps corrupting the heap. This is one of the more intricate well-defined but ultimately harmful overflows that we saw; it is Type 4.

7) 186.crafty: In this benchmark we found some Type 2 wraparounds in `evaluate.c` used to reason about a bitmap representation of the chessboard. Additionally, there is a Type 3 statistic miscalculation that seems like a minor implementation oversight.

8) 197.parser: This benchmark had a number of overflows, including undefined signed overflows in a hash table as indicated in Table III. Here we focus on an overflow in 197.parser's custom memory allocator, shown in Listing 8. This loop computes `SIZE_MAX`, setting `largest_block` to `SIZE_MAX >> 1`. Unsigned overflow is used to determine when `j` exceeds the capacity of `size_t` (note that `i = j` when the loop terminates). While `SIZE_MAX` wasn't introduced until C99, it's unclear why `sizeof` and a shift weren't used instead. This overflow is Type 2: well-defined and benign.

9) 254.gap: Most of the undefined signed overflows in the SPEC 2000 suite are currently latent: today's compilers do not break them by exploiting the undefinedness. 254.gap is different: today's compilers cause it to go into an infinite loop unless two's complement integer semantics are forced. From the LLVM developers' mailing list:[8]

> "This benchmark thinks overflow of signed multiplication is well defined. Add the `-fwrapv` flag to ensure that the compiler thinks so too."

We did not investigate the errors in this benchmark due to the complex and obfuscated nature of the code. However, as shown in Table III, our tool reported many sources of signed wraparound as expected. The signed overflows are Type 1, as they rely on undefined behavior and there is no mention of `-fwrapv` in the documentation or source code. Using `-fwrapv` would make this Type 2, but non-portable because it would be limited to compilers that support `-fwrapv`.

Table IV
EXPOSING TIME BOMBS IN SPEC CINT 2006 BY MAKING UNDEFINED INTEGER OPERATIONS RETURN RANDOM RESULTS. ✔ INDICATES THE APPLICATION CONTINUES TO WORK; ✖ INDICATES THAT IT BREAKS.

Benchmark	ANSI C / C++98	C99 / C++11
400.perlbench	✔	✔
401.bzip2	✔	✖
403.gcc	✖	✖
445.gobmk	✔	✔
464.h264ref	✔	✖
433.milc	✖	✖
482.sphix3	✔	✖
435.gromacs	✔	✔
436.cactusADM	✔	✖

10) Shift Overflows: In our examination of SPEC CINT2000 we also checked for shift errors, finding a total of 93 locations. Of these, 43 were `1 << 31` which is an idiom for `INT_MIN` that's legal in ANSI C, and another 38 were shifts with a negative left operand which is also legal in ANSI C. For space reasons, and because this behavior is fairly benign (and well-defined until C99), these are omitted from Table III and not discussed in detail.

Summary of Overflows in SPEC CINT2000: As shown in Table III, we found a total of 219 static sources of overflow in eight of the 12 benchmarks. Of these, 148 were using unsigned integers, and 71 were using signed integers ($\tilde{3}2\%$). Overall, the most common uses of overflow were for hashing (128), overflow check requiring fwrapv (37), bit manipulation (25), and random number generation (6). Finally, the vast majority of overflows found (both unsigned and signed) were not bugs, suggesting occurrence of integer overflow by itself is not a good indicator of a security vulnerability or other functional error.

D. Latent Undefined Overflows: Harmless, or Time Bombs?

The presence of integer overflows that result in undefined behavior in a well-worn collection of software like SPEC CINT raises the question: *Do these overflows matter?* After all—with the notable exception of 254.gap—the benchmarks execute correctly under many different compilers. For each undefined overflow site in a benchmark program that executes correctly, there are two possibilities. First, the values coming out of the undefined operation might not matter. For example, a value might be used in a debugging printout, it might be used for inconsequential internal bookkeeping, or it might simply never be used. The second possibility is that these overflows are "time bombs": undefined behaviors whose results matter, but that happen—as an artifact of

[8] http://lists.cs.uiuc.edu/pipermail/llvm-commits/Week-of-Mon-20110131/115969.html

today's compiler technology—to be compiled in a friendly way by all known compilers.

To find the time bombs, we altered IOC's overflow handler to return a random value from any integer operation whose behavior is undefined by the C or C++ standard. This creates a high probability that the application will break in an observable way if its execution actually depends on the results of an undefined operation. Perhaps amusingly, when operating in this mode, IOC is still a standards-conforming C or C++ compiler—the standard places no requirements on what happens to a program following the execution of an operation with undefined behavior.

SPEC CINT is an ideal testbed for this experiment because it has an unambiguous success criterion: for a given test input, a benchmark's output must match the expected output. The results appear in Table IV. In summary, the strict shift rules in C99 and C++11 are routinely violated in SPEC 2006. A compiler that manages to exploit these behaviors would be a conforming implementation of C or C++, but nevertheless would create SPEC executables that do not work.

E. Integer Overflows in the Wild

To understand the prevalence of integer overflow behaviors in modern open-source C and C++ applications, we ran IOC on a number of popular applications and libraries. In all cases, we simply compiled the system using IOC and then ran its existing test suite (i.e., we typed "`make check`" or similar). For this part of our work, we focused on undefined behaviors as opposed to well-defined wraparounds. Also, we explicitly avoided looking for bugs based on the stricter C99 and C++11 shift rules; developer awareness of these rules is low and our judgment was that bug reports about them would be unwelcome.

1) SQLite: SQLite is a compact DBMS that is extremely widely used: it is embedded in Firefox, Thunderbird, Skype, iOS, Android and others. In March 2011 we reported 13 undefined integer overflows in the then-current version. Although none of these undefined behaviors were believed to be sources of bugs at the time, some of them could have been time bombs. The main developer promptly fixed these overflows and IOC found no problems in the next version.

IOC also found a lossy conversion from unsigned int to signed int that resulted in a negative value being used as an array index. This code was triggered when SQLite attempted to process a corrupted database file. The SQLite developer also promptly fixed this issue.[9]

2) SafeInt and IntegerLib: SafeInt [7] is a C++ class for detecting integer overflows; it is used in Firefox and also "used extensively throughout Microsoft, with substantial adoption within Office and Windows." We tested SafeInt and found 43 sites at which undefined overflows occurred, about half of which were negations of `INT_MIN`. The SafeInt developers were aware that their code performed this operation, but did not feel that it would have negative consequences. However, development versions of G++ do in fact exploit the undefinedness of `-INT_MIN` and we found that when SafeInt was built with this compiler, it returned incorrect results for some inputs. Basically, the G++ optimizer finally triggered this time bomb that had been latent in SafeInt for some time. We informed the developers of this issue and they promptly released a new version of SafeInt that contains no undefined integer behaviors.

Listing 9. An overflow in IntegerLib

```
1 int addsi (int lhs, int rhs) {
2   errno = 0;
3   if ((((lhs+rhs)^lhs)&((lhs+rhs)^rhs))
4       >> (sizeof(int)*CHAR_BIT-1)) {
5     error_handler("OVERFLOW ERROR", NULL, EOVERFLOW);
6     errno = EINVAL;
7   }
8   return lhs+rhs;
9 }
```

We tested another safe integer library, IntegerLib [10], which was developed by CERT. This library contains 20 sites at which undefined integer overflows occur. One of them is shown in Listing 9; it is supposed to check if arguments `lhs` and `rhs` can be added without overflowing. However, at Line 3 the arguments are added without being checked, a bug that results in undefined behavior. A reasonable solution for this case would be to cast the arguments to an unsigned type before adding them.

3) Other Codes: Six overflows in the GNU MPC library that we reported were promptly fixed. We reported 30 overflows in PHP; subsequent testing showed that 20 have been fixed. We reported 18 overflows in Firefox, 71 in GCC, 29 in PostgreSQL, 5 in LLVM, and 28 in Python. In all of these cases developers responded in a positive fashion, and in all cases except Firefox and LLVM we subsequently received confirmation that at least some of the overflows had been fixed. Finally, we reported nine undefined overflows in the GNU Multiple Precision Arithmetic Library, one in BIND, and one in OpenSSL. We received no response from the developers of these three packages.

Out of all the codes we tested, only three were completely free of undefined integer overflows: Kerberos, libpng, and libjpeg. All three of these packages have had security vulnerabilities in the past; undoubtedly the more recent versions that we tested have been subjected to intense scrutiny.

V. Prior Work

Integer overflows have a long and interesting history. The popular Pac-Man game, released in 1980, suffered from two known integer overflows that generate user-visible, and surprising, artifacts [11], [12]. More recently, as buffer overflows in C and C++ programs have been slowly brought under control, integer overflows have emerged as an im-

[9] http://www.sqlite.org/src/info/f7c525f5fc

portant root cause of exploitable vulnerabilities in Internet-facing programs [3], [13].

Solutions to integer overflow are almost as old as the problem. For example, the IBM 702 provided a hardware-based implementation of variable-precision integers more than 50 years ago [14]. MacLisp, in the 1960s, provided the first widely-available software implementation of arbitrary precision arithmetic. Even so, for a variety of reasons, today's low-level programming languages eschew well-known integer overflow solutions, forcing programmers to deal with modulo integers and undefined behaviors.

Although there has been extensive work, especially during the last decade or so, on tools and libraries for *mitigating* integer-based security vulnerabilities, *none of these tools have been used to understand the patterns of integer numerical overflows in real-world programs and benchmarks*, which is the main focus of our work. Instead, those efforts have focused primarily on developing new tools and libraries and evaluating their efficacy. In particular, none of these projects has specifically attempted to examine the prevalence of undefined behaviors, although there is data in some of these papers about specific bugs. Moreover, none of these projects has attempted to examine the prevalence of intentional wraparound behaviors, or the idioms for which they are used, except the limited data in the paper on RICH.

The RICH paper presents two relevant pieces of data [4]. First, it classifies integer numerical errors from MITRE's CVE database [15] as overflow, underflow, signedness, or truncation errors. This classification does not show *how prevalent* numerical errors are across programs because the survey only looks at cases where overflows have already been reported, not a general collection of programs. Second, they briefly discuss some benign overflow behaviors that are flagged as errors by their tool, and discuss for what algorithms those overflows are used. That study provides limited data about the prevalence and patterns of intentional uses because their goal was different—to evaluate false positives from RICH. We study the empirical questions systematically and in more detail.

Other prior research on mitigating integer-based security vulnerabilities is more tangential to our work. We briefly discuss that work to illustrate the solutions available. The tools vary from static analysis and dynamic instrumentation to libraries with various strategies to mitigate the problem.

RICH is a compiler-based tool that instruments programs to detect signed and unsigned overflows in addition to lossy truncations and sign-conversions [4]. BRICK [16] detects integer overflows in compiled executables using a modified Valgrind [17]. The runtime performance is poor (50X slowdown) and the lack of C-level type information in executable code causes both false positives and false negatives. SmartFuzz [18] is also based on Valgrind, but goes further by using whitebox testing to generate inputs leading to good test coverage. IntScope [19] is a static binary analysis tool for integer vulnerabilities.

The As-if Infinitely Ranged (AIR) integer model [20] is an ambitious solution that is intended to be used online. It simply provides well-defined semantics for most of C/C++'s integer-related undefined behaviors. AIR provides a strong invariant—integer operations either produce representable results or else trap—while being carefully designed to minimally constrain the optimizer. An alternative online solution is provided by libraries such as SafeInt [7] and IntegerLib [10], where checked operations must be explicitly invoked and overflows explicitly dealt with. SafeInt, however, is quite easy to use because it exploits C++'s exceptions and operator overloading.

VI. Conclusion

We have conducted an empirical study of the prevalence and patterns of occurrence of integer overflows in C and C++ programs, both well-defined and undefined, and both intentional and inadvertent. We find that intentional uses of wraparound behaviors are much more common than is widely believed, e.g., over 200 distinct locations in SPEC CINT2000 alone. We identify a wide range of algorithms for which programmers use wraparound intentionally.

Unfortunately, we also observe that some of the intentional uses are written with signed instead of unsigned integer types, triggering undefined behaviors in C and C++. Optimizing compilers are free to generate arbitrary results for such code. In fact, we identified a number of lurking "time bombs" that happen to work correctly with some of today's compilers but may fail with future compiler changes, such as more aggressive optimizations. Finally, we identified a number of previously unknown numerical bugs in widely used open source software packages (and even in safe integer libraries!), many of which have since been fixed or acknowledged as bugs by the original developers. Even among mature programs, only a small fraction are free of integer numerical errors.

Overall, based on the locations and frequency of numerical errors, we conclude that there is widespread misunderstanding of the (highly complex) language rules for integer operations in C/C++, even among expert programmers. Our results also imply that tools for *detecting* integer numerical errors need to distinguish intentional from unintentional uses of wraparound operations—a challenging task—in order to minimize false alarms.

Acknowledgments

We thank Tennessee Carmel-Veilleux, Danny Dig, Ganesh Gopalakrishnan, Alex Groce, Mary Hall, Derek Jones, Swarup Sahoo, and the ICSE 2012 reviewers for their insightful comments on drafts of this paper. This research was supported, in part, by an award from DARPA's Computer Science Study Group, and by the Air Force Research Laboratory (AFRL).

References

[1] MITRE Corporation, "CVE-2002-0639: Integer overflow in sshd in OpenSSH," 2002, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0639.

[2] ——, "CVE-2010-2753: Integer overflow in Mozilla Firefox, Thunderbird and SeaMonkey," 2010, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2753.

[3] S. Christey, R. A. Martin, M. Brown, A. Paller, and D. Kirby, "2011 CWE/SANS Top 25 Most Dangerous Software Errors," MITRE Corporation, Tech. Report, September 2011, http://cwe.mitre.org/top25.

[4] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song, "RICH: Automatically protecting against integer-based vulnerabilities," in *Proc. of the Symp. on Network and Distributed Systems Security (NDSS)*, San Diego, CA, USA, Feb. 2007.

[5] "clang: a C language family frontend for LLVM," http://clang.llvm.org/ ; accessed 21-Sept-2011.

[6] ISO, *ISO/IEC 14882:2011: Programming languages — C++*. International Organization for Standardization, 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[7] D. LeBlanc, "Integer handling with the C++ SafeInt class," 2004, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp.

[8] ——, "Author's blog: Integer handling with the C++ SafeInt class," http://safeint.codeplex.com/.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of the 2004 Intl. Symp. on Code Generation and Optimization (CGO'04)*, Palo Alto, CA, USA, Mar. 2004.

[10] CERT, "IntegerLib, a secure integer library," 2006, http://www.cert.org/secure-coding/IntegerLib.zip.

[11] D. Hodges, "Why do Pinky and Inky have different behaviors when Pac-Man is facing up?" Dec. 2008, http://donhodges.com/pacman_pinky_explanation.htm ; accessed 21-Sept-2011.

[12] Wikipedia, "Pac-Man," 2011, http://en.wikipedia.org/w/index.php?title=Pac-Man&oldid=450692749#Split-screen ; accessed 21-Sept-2011.

[13] S. Christey and R. A. Martin, "Vulnerability type distributions in CVE," MITRE Corporation, Tech. Report, May 2007, http://cwe.mitre.org/documents/vuln-trends.html.

[14] Wikipedia, "Arbitrary-precision arithmetic," 2011, http://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic ; accessed 21-Sept-2011.

[15] MITRE Corporation, "Common Vulnerability and Exposures," http://cve.mitre.org/.

[16] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie, "Brick: A binary tool for run-time detecting and locating integer-based vulnerability," in *Proc. of the 4th Intl. Conf. on Availability, Reliability and Security*, Fukuoka, Japan, Mar. 2009, pp. 208–215.

[17] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," in *Proc. of the 3rd Workshop on Runtime Verification*, Boulder, CO, Jul. 2003.

[18] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary Linux programs," in *Proc. of the 18th USENIX Security Symposium*, 2009, pp. 67–82.

[19] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution," in *Proc. of the 16th Network and Distributed System Security Symp.*, San Diego, CA, USA, Feb. 2009.

[20] R. B. Dannenberg, W. Dormann, D. Keaton, R. C. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum, "As-if infinitely ranged integer model," in *Proc. of the 21st Intl. Symp. on Software Reliability Engineering (ISSRE 2010)*, San Jose, CA, USA, Nov. 2010, pp. 91–100.