

Understanding Integer Overflow in C/C++¹

WILL DIETZ, University of Illinois at Urbana-Champaign
PENG LI and JOHN REGEHR, University of Utah
VIKRAM ADVE, University of Illinois at Urbana-Champaign

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, LLVM, Python, BIND, and OpenSSL; many of these have since been fixed.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features; D.3.0 [Programming Languages]: General—Standards; D.2.5 [Software Engineering]: Testing and Debugging—Diagnostics; D.2.4 [Software Engineering]: Software/Program Verification—Validation

General Terms: Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: integer overflow, integer wraparound, undefined behavior

1. INTRODUCTION

Integer numerical errors in software applications can be insidious, costly, and exploitable. These errors include overflows, value-losing conversions (e.g., a truncating 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). We refer to all these types of errors collectively as “overflow errors.” 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 [MITRE Corporation 2002] and Firefox [MITRE Corporation 2010], 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” [Christey et al. 2011].

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, and may naïvely use signed types in intentional wraparound operations.² If such uses were rare, compiler-based overflow detection would be a reasonable way to detect integer overflow errors. 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.

¹This is an extended version of a paper that was presented at the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, June 2012.

²In fact, in the course of our work, we have found that even experts writing *safe integer libraries* or *tools to detect integer errors* sometimes make mistakes due to the subtleties of C/C++ semantics for numerical operations.

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 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 [Brumley et al. 2007], 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 8, that paper only tangentially addresses the third point above. We study all of these questions systematically.

This article expands upon our previous work [Dietz et al. 2012] making 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.³ 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 6).

In addition, this article makes the following new contributions: First, we conducted a new large-scale experiment in which we automatically tested for overflows in 1172 of the top 10,000 Debian packages. We found that 35% of these packages triggered an overflow during execution of their test suites, and 16% invoked overflows with *undefined* semantics. *These results provide strong new evidence that a substantial fraction of arbitrary C/C++ software packages, including many well-tested ones, contain serious overflow errors.* Second, we improved IOC by implementing and evaluating recoverability and two optimizations described in Section 4.4. Third, we integrated IOC into mainline Clang, a widely used production C/C++ compiler (e.g., it is the primary system compiler C/C++ compiler on Mac OS and iOS). The features of IOC are available via the `-fsanitize=*` family of options, e.g., `-fsanitize=integer` and `-fsanitize=shift`. This deployment led to improvements in diagnostics and the addition of new features such as error deduplication to meet user needs. We describe this experience in Section 5, contributing the discussion of our end-to-end development of an effective bug-detection tool that can now be easily used by LLVM/Clang users. Fourth, this article contains additional discussion of relevant details including hardware support for integer arithmetic (Section 2), language rules governing implementation-defined behavior (Section 3.1), and an explanation of the usual arithmetic conversions (Section 3.2). Fifth, we evaluated the performance overhead imposed by IOC under different checking policies on integer-intensive CPU benchmarks as well as security-sensitive server software. These results are presented in Section 7.

³ IOC is available as part of the production version of the Clang compiler for C/C++ [Clang 2011] as of version 3.3. Usage information and list of available checks available at <http://clang.llvm.org/docs/UsersManual.html>. The original IOC as described in Section 4 is also available at <http://embed.cs.utah.edu/ioc/>.

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 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, 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 sophisticated techniques and/or rely on manual intervention (e.g., annotations) to distinguish intentional from unintentional overflows.

2. HARDWARE INTEGER ARITHMETIC

Integer overflow bugs in C and C++ find their roots in features and limitations of the underlying hardware platform. On modern processors, an n -bit unsigned integer is represented in such a way that its value is

$$\sum_{i=0}^{n-1} x_i 2^i$$

where x_i is the value of the i th bit. The range of an n -bit unsigned integer is $0 \dots 2^n - 1$. Signed integers on all modern platforms use a *two's complement* representation where the value of an integer is:

$$-x_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

An advantage of two's complement is that signed and unsigned addition can be performed using the same operation. The same is true for subtraction and multiplication. Historically, this was advantageous because fewer instructions needed to be implemented. Also, unlike the *ones' complement* and *sign-magnitude* representations, two's complement has only one representation for zero. A drawback of two's complement is that its range, $-2^{n-1} \dots 2^{n-1} - 1$, is asymmetric. Thus, there is a representable value, -2^{n-1} , that does not have a representable additive inverse—a fact that programmers can and do forget.

When an n -bit addition or subtraction operation on unsigned or two's complement integers overflows, the result “wraps around,” effectively subtracting 2^n from, or adding 2^n to, the true mathematical result. Equivalently, the result can be considered to occupy $n + 1$ bits; the lower n bits are placed into the result register and the highest-order bit is placed into the processor's carry flag.

Unlike addition and subtraction, the result of an integer multiplication can wrap around many times. Consequently, a processor typically places the result of an n -bit multiplication into a location that is $2n$ bits wide, such as a pair of registers. Division instructions commonly have an analogous structure where the dividend is $2n$ bits wide, the divisor is n bits, and the quotient and remainder are both n bits. Two's complement division (unlike addition, subtraction, and multiplication) is not performed by the same operation as unsigned division of the same width. For example, the x86 architecture has both the `idiv` instruction for two's complement integer division and `div` for unsigned integer division.

Table I. Examples of C/C++ integer operations and their results

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

Processors intended for digital signal processing, including dedicated DSPs and also SIMD extensions to general-purpose processors, support *saturating* semantics for overflowing integer operations. Saturating arithmetic never wraps around, but rather “sticks” at the maximum or minimum representable value. In some domains saturation is a reasonable solution to the problem of integer overflow, but since saturating semantics are not used by C or C++, we will not consider it further.

Processor-level left-shift instructions typically place zeros into vacated bit positions. Right shift instructions come in logical and arithmetic varieties, where a logical shift places zeros into vacated bit positions and an arithmetic shift fills vacated bits with the value contained in the (unshifted) most significant bit, in order to avoid changing the sign of a two’s complement integer. Finally, many architectures support *rotate* instructions that treat the value being shifted as a circular buffer: bits that “fall off” one end of the register are used to fill in vacated bit positions at the other end. Rotating semantics are not available to C/C++ programs and we will not consider them further.

3. INTEGER ARITHMETIC IN C AND C++

There is tension in the design of C and C++ between the desire for portability on one hand, and the desire for highly predictable and efficient execution of low-level code on the other hand. Therefore, in spite of C’s reputation for being a simple “portable assembly language,” its integer semantics are quite different from those provided by real assembly languages. The same is true of C++. The most important differences include the following:

- Many integer operations are not entirely portable across platforms due to *implementation-defined behaviors*.
- C/C++ perform pervasive implicit conversions between bitwidths and between signed and unsigned types.
- A number of arithmetic operations have *undefined behavior*.
- Since the carry and overflow flags are not exposed in the language, it is not straightforward for programmers to write code that detects overflow before or after it happens.

The rest of this section explores the consequences of the first three of these issues; the last is discussed in Section 4.2. Table I contains some concrete examples of the results of evaluating some C/C++ expressions.

3.1. Implementation-Defined Behaviors

An implementation-defined behavior in C/C++ is one where individual implementations are given freedom to make a choice, but this choice must be consistent and documented. A few of C/C++'s integer-related implementation-defined behaviors would break most non-trivial C programs if they deviated from the expected values (given in parentheses):

- The number of bits in a byte (eight)
- The representation used for signed integers (two's complement)
- The result of converting an integer-typed value into a narrower signed integer-typed value, where the value cannot be represented in the new type (truncation, with the potential for converting a positive number into a negative number)
- The result of performing a bitwise operation on values with signed integer types (it is as if the operands were converted to unsigned integers of the same width, the bitwise operation was performed on them, and the result was cast back to the signed type)

Other behaviors do tend to vary across platforms and compilers. These include the signedness of the `char` type and the size of the `short`, `int`, and `long` types (expressed as a multiple of the size of a `char`). The `int` type in C99 is not required to hold values in excess of 32,767.

3.2. The Usual Arithmetic Conversions

Most integer operators in C/C++ require that both operands have the same type and, moreover, that this type is not narrower than an `int`. The collection of rules that accomplishes this is called *the usual arithmetic conversions*. The full set of rules encompasses both floating point and integer values; here we will discuss only the integer rules. First, both operands are *promoted*:

If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called the integer promotions. All other types are unchanged by the integer promotions.

If the promoted operands have the same type, the usual arithmetic conversions are finished. If the operands have different types, but either both are signed or both are unsigned, the narrower operand is converted to the type of the wider one.

If the operands have different types and one is signed and the other is unsigned, then the situation becomes slightly more involved. If the unsigned operand is narrower than the signed operand, and if the type of the signed operand can represent all values of the type of the unsigned operand, then the unsigned operand is converted to signed. Otherwise, the signed operand is converted to unsigned.

These rules can interact to produce counterintuitive results. Consider this function:

```
int compare (void) {
    long a = -1;
    unsigned b = 1;
    return a > b;
}
```

For a C/C++ implementation that defines `long` to be wider than `unsigned`, such as GCC for x86-64, this function returns zero. However, for an implementation that defines `long` and `unsigned` to have the same width, such as GCC for x86, this function returns one. The issue is that on x86-64, the comparison is between two signed integers, whereas on x86, the comparison is between two unsigned integers, one of which is very large. Some compilers are capable of warning about code like this.

3.3. Unsigned Overflow

According to the C99 standard:

A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

Thus, the semantics for unsigned overflow in C/C++ are precisely the same as the semantics of processor-level unsigned overflow as described in Section 2. As shown in Table I, `UINT_MAX+1` must evaluate to zero in a conforming C and C++ implementation.

Unsigned overflow tends to lead to two kinds of problems in C/C++ programs. First, it is often the case that developers are simply not expecting values to overflow. For example, one of the infamous Therac-25 bugs [Leveson and Turner 1993] occurred when an 8-bit variable, which used a non-zero value to signal an error, wrapped around to zero, causing a crucial checking function to be bypassed. Second, since the sizes of integer types are implementation-defined, overflow will occur at different values on different platforms. For example, GCC evaluates `0UL-1` to $2^{64} - 1$ on x86-64 and to $2^{32} - 1$ on x86.

3.4. Signed Overflow and Other Undefined Behaviors

If a signed arithmetic operation overflows, for example by evaluating `INT_MAX+1`, the behavior of the C implementation is *undefined*. 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:⁴

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

Other operations with undefined behavior include shifting a value by at least as many bit positions as the bitwidth of the promoted type of the operand, shifting by a negative number of bit positions, and division by zero. Undefined behavior represents a significant departure from the semantics of processor-level operations, and our experience is that many developers fail to appreciate the full consequences of this. The rest of this section examines these consequences.

3.4.1. 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.⁵ 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 zero for the first value (line 6) and a one 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.

3.4.2. 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

⁴<http://catb.org/jargon/html/N/nasal-demons.html>

⁵<http://code.google.com/p/nativeclient/issues/detail?id=245>

```

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 }

```

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

overflows. For example, in 2005 a principal PostgreSQL developer was annoyed that his code was broken by a recent version of GCC:⁶

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 6.5 we describe a time bomb in `SafeInt` [LeBlanc 2004]: a library that is itself intended to help developers avoid undefined integer overflows. This operation had been compiled by GCC (and other compilers) into code that behaved correctly. However, a subsequent version of GCC (4.7) exposed 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 [LeBlanc 2011].

3.4.3. Illusion of Predictability. Some compilers, at some optimization levels, have predictable behavior for some undefined operations. For example, at low optimization levels, C and C++ compilers typically give two’s complement semantics to signed overflows. It is, however, very unwise to rely on this, as the behavior can change when the compiler, the compiler version, or the compiler flags are changed.

3.4.4. Informal Dialects. Some compilers support stronger semantics than are mandated by the standard. For example, both GCC and Clang 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.

3.4.5. 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 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; compiler vendors are unwilling to impose this overhead on their

⁶<http://archives.postgresql.org/pgsql-hackers/2005-12/msg00635.php>

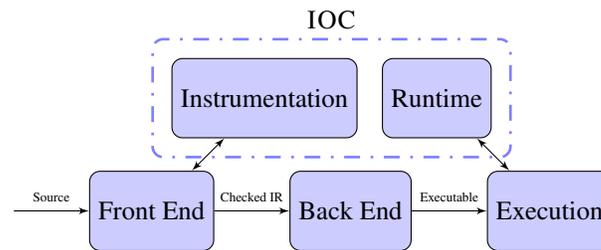


Fig. 1. Architecture of IOC

users. The C standards committee has recognized the problem and C11 explicitly makes this case undefined.

4. 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 [Clang 2011], the C/C++ frontend to LLVM [Lattner and Adve 2004]. 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.

This section discusses some of the key design choices made in the tool, including at what point in compilation to perform the instrumentation; how to perform the run-time overflow checks; the design of the run-time library; and, how to efficiently recover from a detected overflow operation and continue execution.

4.1. Where to Put the Instrumentation Pass?

The IOC 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), which is much simpler to transform than the AST and also allows the transformation to be more naturally expressed as a compiler pass. Unfortunately, this proved to be unreliable and unnecessarily complicated, because it requires a substantial amount of C-level type information to be recorded with the IR in order to support a correct transformation. For example, the IR does not distinguish between signed and unsigned integer types (since the two’s complement representation is equivalent for both), but this distinction is essential for correct checking of integer overflows in C code. Similarly, some important operations (such as signed to unsigned casts) do not exist at the IR level. Also, the original transformation was further complicated by the lack of a one-to-one mapping between IR and AST nodes. In short, it is much less error-prone to do the instrumentation in the frontend, where all the required information is naturally available, despite the greater complexity of the representation.

4.2. 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:

$$((s_2 > 0) \wedge (s_1 > (\text{INT_MAX} - s_2))) \vee ((s_2 < 0) \wedge (s_1 < (\text{INT_MIN} - s_2)))$$

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, using the flag checks has an uneven effect on performance. We evaluated this performance question experimentally, as follows.

4.2.1. Performance of the Overflow Checks. We studied the overall performance impact of IOC to answer two questions: How do the precondition test and the CPU flag postcondition test compare in terms of the overheads they introduce? And how much run-time overhead is introduced by using IOC?

To answer these questions, we compiled SPEC CPU 2006 in three ways: (1) A baseline compilation using Clang with optimization options set for maximum expected performance. (2) Checking for undefined integer overflows (shifts and arithmetic) using precondition checks. (3) Checking for undefined integer overflows (shifts and arithmetic) using the CPU flag postcondition test. 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.

For undefined behavior checking using precondition checks, slowdown relative to the baseline ranged from -0.5% – 191% , i.e., from a tiny accidental speedup to a threefold increase in run-

time. 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.

The explanation for the uneven benefits of the CPU flag checks 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 optimized by relatively few optimization passes, causing much of the potential performance gain due to this approach to be unrealized.

4.3. Runtime Library for Error Reporting

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, and Clang has outstanding error reporting.) 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 it can 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 invoked in execution environments where capturing their output would be difficult.

Finally, to avoid overwhelming users with repetitive 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.

4.4. Efficient Error Recovery

When an overflow check is triggered, it is often desirable to continue program execution, which we call *recovery*. When running IOC on a code base there are often multiple errors, many of which may be benign, and using recoverable checking to find these with a single run is useful. Unfortunately, our original implementation showed substantial slowdowns in error-free execution when making checks recoverable. To evaluate these overheads, we ran IOC with signed overflow and division-by-zero checks on the subset of SPEC CINT2006 benchmarks that did not trigger these errors dynamically. Adding more check types caused this subset to be too small to be interesting. With our initial implementation, the no-recovery case had an overhead of 15% on average while recoverable checking had an average overhead of 35%.

We used hardware performance counters, accessed via Intel’s VTune software [Intel 2013], to investigate the causes of the slowdowns. The first source of slowdown poor code layout, which degraded performance via bad cache interaction and frontend stalls. We resolved this issue by adding branch prediction metadata at the LLVM IR level, which the backend uses to guide block placement. By hinting that the checks are unlikely to be taken, the basic blocks containing calls to the runtime are not interspersed with the normal code. This issue did not arise with no-recovery checking because the code after the handler call is marked as unreachable, which similarly influences block layout.

Using performance counters, we verified that our fix resolved the layout problem. For example, in the SPEC benchmark 456.hammer, we observed that the number of instruction-starved cycles (cycles where the front-end did not issue an instruction) was reduced by 43% and CPI decreased from 0.685 to 0.614. Indeed, this resulted in a significant improvement for this benchmark: down from 179s to 142s (from 53% to 21% slowdown, relative to the no-recovery case). Averaged across the benchmarks, this optimization brought recovery overhead down to 25%, significantly less than the original 35%.

The second problem we identified was that, to enable recovery, failure-handling blocks for inserted checks must branch back into the program execution, which hurts register allocation performance in error-free paths. In particular, the compiler backend has to allocate registers with the additional constraints imposed by the parameter preparation and the registers clobbered by the runtime call. We resolved this by changing the runtime functions to use a special “cold” calling convention which indicates the call clobbers as little as is possible for the architecture. This was a difficult change because it requires making the runtime functions save/restore all registers on entry/exit with hand-written assembly and because, while LLVM already supports “coldcc” as part of the IR, it is treated the same as the normal C calling convention on all architectures. To handle the former we used hand-written assembly wrappers for the normal C functions, and plan on proposing a function attribute to cause the compiler to do this automatically (gcc has support for this). For the latter, we extended the LLVM x86 and x86_64 back ends to treat coldcc calls as not clobbering any registers.

Together, these changes reduced average overhead to 21% (only 6% over the no-recovery case), which is low enough to make checks recoverable by default. These optimizations would benefit any similar runtime check instrumentation: the first has already been adopted into mainline Clang for use in other sanitization checks, and there are plans to make use of the calling convention optimization once sufficient support has been added to LLVM.

5. IOC DEPLOYMENT

After the initial release of IOC, our findings and user feedback encouraged us to work to make IOC more widely available by working with the LLVM and Clang developers to include integer overflow checks in the official Clang code. During this process we made a number of functionality, efficiency, and usability changes resulting in the first widely-available easy-to-use tools to enable developers to better reason about integer overflows in the software they build and use. As a result of these efforts, integer checking support is now available in LLVM/Clang. The first release including our integer checking features was version 3.3 in June 2013.

5.1. User Interface

In the original version of IOC, users specified what operations to check by requesting checks for violations of a particular version of the C standard. This was found to be cumbersome and for the version of IOC deployed with Clang, we integrated our command line options with the `-fsanitize` family of options, providing users with a unified interface to a variety of dynamic checks. Example invocations include `-fsanitize=integer` (check all suspicious integer operations) and `-fsanitize=shift` (check only shift operations).

5.2. Improved Diagnostics

Diagnostics were improved to read more naturally and to match the formatting style of errors given statically by the Clang compiler. Here are two examples taken from ClamAV 0.97.6:

```
bytecode_vm.c:687:6: runtime error: signed integer overflow: 65535 * 65535 cannot be
represented in type 'int'
bytecode_vm.c:709:6: runtime error: left shift of 255 by 56 places cannot be represented in
type 'int64_t' (aka 'long')
```

These new diagnostics are designed to significantly improve the user experience by providing a natural explanation about what occurred including the operands that triggered the problem. While the same information was available in the diagnostics printed by the original IOC, it has been made

cleaner with attention given to details such as providing source-level types and more specific explanations.

5.3. New Features

Besides changing the interface, we also introduced a number of features to make IOC suitable for a larger variety of use cases. First, we added the ability to continue program execution (to “recover”) after a failing check. Making checks recoverable incurs a performance penalty, which we significantly reduced through the changes described in Section 4.4.

In addition to making checks recoverable, once a particular check has been triggered, often users are not interested in seeing the error again. This error deduplication was originally accomplished using a hash table, but this design proved to not scale well. To address this, we now emit a byte for each check which is used to track its triggered state which scales well and only requires a minor amount of additional space in the resulting binary.

Finally, often integer overflows are known to be intentional or the programmer has investigated it and determined it to be acceptable. To address these use cases while still being useful in reporting undesired integer overflows a whitelist functionality was introduced to enable users to specify certain files or functions that should not be checked.

5.4. Runtime Concerns

Inclusion in a production compiler required us to address a number of edge-cases in our runtime that we previously ignored in our research prototype. Examples include avoiding dependencies on `libc`, linking the IOC runtime automatically and transparently in a variety of build scenarios, ensuring correct execution in multithreaded environments, and ensuring the preservation of `errno` across calls into the runtime functions.

6. INTEGER OVERFLOW STUDY

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

6.1. 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 understate the prevalence of integer numerical errors as well as the prevalence of intentional uses of wraparound in these programs. A stress testing campaign, or use of a formal methods tool, would be very likely to uncover additional integer 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.

Finally, while some parts of our study are methodical, other parts were performed in an informal fashion, and hence our reporting of the results is also informal. For example, in many cases we lack links into bug reporting systems because the programs that we tested do not have bug reporting systems—the bugs we found were reported by email. In other cases, we lack a definitive answer about whether a particular bug was fixed because (1) the developers failed to confirm that they fixed the bug and (2) the changes across versions of programs are often large enough that manually tracking the status of bugs in the code is infeasible.

6.2. A Taxonomy for Overflows

Table II summarizes our view of the relationship between different integer overflows in C/C++ and the correctness of 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 II. Taxonomy of integer overflows in C and C++ with references to discussion of examples

	undefined behavior e.g. signed overflow, shift error, divide by zero	defined behavior e.g. unsigned wraparound, signed wraparound with <code>-fwrapv</code>
intentional	<i>Type 1:</i> design error, may be a “time bomb” § 6.3.3, 6.3.9	<i>Type 2:</i> no error, but may not be portable § 6.3.2, 6.3.5, 6.3.8
unintentional	<i>Type 3:</i> implementation error, may be a “time bomb” § 6.3.4	<i>Type 4:</i> implementation error § 6.3.1, 6.3.6

```

1 /* (this test assumes unsigned comparison) */
2 if (w - d >= e)
3 {
4     memcpy(slide + w, slide + d, e);
5     w += e;
6     d += e;
7 }

```

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

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.

6.3. 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 can be 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.

6.3.1. 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 reported by IOC and while investigating the report we discovered

Table III. Integer wraparounds reported in SPEC CINT2000

Name	Location ¹	Op ²	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	Random Number Generation (Listing 3)
175.vpr	util.c(463:39)	+ _u	Random Number Generation
175.vpr	util.c(484:34)	* _u	Random Number Generation
175.vpr	util.c(484:39)	+ _u	Random Number Generation
176.gcc	combine.c × 6	- _s	Find INT_MAX (Listing 4)
176.gcc	cse.c × 5	+ _u	Hash
176.gcc	expmed.c × 15	± _{u,s}	Bit manipulation
176.gcc	expmed.c(2484:13)	* _u	Inverse of $x \bmod 2^n$
176.gcc	expmed.c(2484:18)	- _u	Inverse of $x \bmod 2^n$
176.gcc	expmed.c(2484:21)	* _u	Inverse of $x \bmod 2^n$
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 manipulation
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	Random Number Generation
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	Random Number Generation

¹ Source, and line:column. For space, we summarize frequent ones as '× n'.² Operation Type(s), and Signed/Unsigned.

```

1 #define IA 1103515245u
2 #define IC 12345u
3 #define IM 2147483648u
4
5 static unsigned int c_rand = 0;
6
7 /* Creates a random integer [0...imax] (inclusive) */
8 int my_irand (int imax) {
9     int ival;
10    /* c_rand = (c_rand * IA + IC) % IM; */
11    c_rand = (c_rand * IA + IC); // Use overflow to wrap
12    ival = c_rand & (IM - 1); /* Modulus */
13    ival = (int) ((float) ival * (float) (imax + 0.999) / (float) IM);
14    return ival;
15 }

```

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

```

1 /* (unsigned) <= 0x7fffffff is equivalent to >= 0. */
2 else if (const_op == ((HOST_WIDE_INT) 1 <<< (mode_width - 1)) & 1)
3 {
4     const_op = 0, op1 = const0_rtx;
5     code = GE;
6 }

```

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

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 upstream⁷ as of version 1.4. Note that this bug existed in `gzip` as of 1993 and was not fixed until 2010. Furthermore, the initial fix was overkill and was later fixed⁸ 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.

6.3.2. 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.

6.3.3. 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 $(1 \ll 31) - 1$. 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 overflows. 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

⁷<http://git.sv.gnu.org/gitweb/?p=gzip.git;a=commit;h=b9e94c93df914bd1d9eec9f150b2e4e00702ae7b>

⁸<http://git.sv.gnu.org/gitweb/?p=gzip.git;a=commit;h=17822e2cab5e47d73f224a688be8013c34f990f7>

```

1 if (moved_once[regno])
2 {
3     insn_count *= 2;
4     ...
5     if (already_moved[regno]
6         || (threshold * savings * m->lifetime) >= insn_count
7         || (m->forces && m->forces->done
8             && n_times_used[m->forces->regno] == 1))
9     {
10        ...

```

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

```

1 #define POWER_OF_2_or_0(I) \
2   (((I) & ((unsigned)(I) - 1)) == 0)
3
4 int
5 integer_ok_for_set (value)
6   register unsigned value;
7 {
8   /* All the "one" bits must be contiguous.
9    * If so, MASK + 1 will be a power of two or zero.*/
10  register unsigned mask = (value | (value - 1));
11  return (value && POWER_OF_2_or_0 (mask + 1));}

```

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

considered to be a time bomb. A better way to compute `INT_MAX` is using unsigned arithmetic. This overflow is Type 1.

6.3.4. 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.

6.3.5. 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.

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

```

1 /* Allocate a zeroed rtvec vector of N elements */
2 rtvec rtvec_alloc (int n) {
3   rtvec rt;
4   int i;
5
6   rt = (rtvec) obstack_alloc (rtl_obstack,
7                               sizeof (struct rtvec_def)
8                               + (( n - 1) * sizeof (rtunion)));
9   ...
10  return rt;
11 }

```

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

```

1 void initialize_memory (void) {
2   SIZET i, j;
3   ...
4   for (i=0, j=1; i < j; i = j, j = (2*j+1))
5     largest_block = i;
6   largest_block &= ALIGNMENT_MASK;
7   // must have room for a nuggie too
8   largest_block += -sizeof(Nuggie);

```

Listing 8: Computation of `SIZE_MAX >> 1` in 197.parser

First, consider this code after the `sizeof` operators are resolved and the promotion rules are applied: $16 + ((\text{unsigned})(n-1)) * ((\text{unsigned})8)$. 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.

6.3.7. 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.

6.3.8. 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` was not introduced until C99, it is unclear why `sizeof` and a shift were not used instead. This overflow is Type 2: well-defined and benign.

6.3.9. 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:

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	✓	✗

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:⁹

“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 the `-fwrapv` flag.

6.3.10. Shift Overflows. In our examination of SPEC CINT2000 we also checked for shift errors, finding a total of 93 locations that overflow. Of these, 43 were `1 << 31` which is an idiom for `INT_MIN` that is legal in ANSI C, and another 38 were left 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 (32%). 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.

6.4. 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 book-keeping, 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 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

⁹<http://lists.cs.uiuc.edu/pipermail/llvm-commits/Week-of-Mon-20110131/115969.html>

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 produce incorrect results because it would expose the latent errors lurking in these codes.

6.5. 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.

6.5.1. SQLite. SQLite is a compact DBMS that is extremely widely used: it is embedded in Firefox, Thunderbird, Skype, iOS, Android and other systems. 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 all of 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.¹⁰

6.5.2. SafeInt and IntegerLib. SafeInt [LeBlanc 2004] 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, newer 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.

We tested another safe integer library, IntegerLib [CERT 2006], 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 workaround for this case would be to cast the arguments to an unsigned type before adding them.

6.5.3. 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.

¹⁰<http://www.sqlite.org/src/info/f7c525f5fc>

```

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 }

```

Listing 9: An overflow in IntegerLib

Out of all the codes we tested, only three were completely free of undefined integer overflows, at least when tested with their own test suites: 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.

6.6. Automated Testing for Overflows

Building on our manual investigation of integer overflow in common C and C++ applications, we also conducted an automated investigation of integer overflows found in a large number of popular open-source software packages. Using a list of the most popular Debian packages, we built each package from source and recorded overflows that occurred during this process. We explain the details of our experiment below, and a summary of our findings is given in Table V.

6.6.1. Methodology. We built an automated testing system, making use of work by Ledru¹¹ to help coerce packages to use a custom compiler in their build process. We modified IOC to integrate well into our build automation, enabling us to capture a list of integer overflows that occurred while running each package with the test suite included with the package. The errors reported are only from the source for each package, and do not include dependencies which were provided from binary packages.

Using this system, we built the top 10,000 source packages from Debian as posted to the popcon website¹² as of Feb 20th, 2013. This list captures the top source packages used regularly by Debian users with the popularity-contest package installed. We used packages available in the more recent “sid” distribution of Debian, and leveraged Debian’s build infrastructure to build each package with IOC and to run any tests included as part of the Debian packaging. In this way we were able to look at a large number of packages automatically.

Most of the 10,000 packages we analyzed were not valid for purposes of this experiment: 1560 failed to complete the build process successfully when using our compiler, and of the remaining, only 3371 contained invocations of code built by our compiler. We determined this by modifying IOC to add to every translation unit processed a global constructor that writes a log entry when and if the resulting code is loaded. This log entry includes details of the command line used to run this software (as read from `/proc/self/cmdline`). Using this modification, we found many packages only loaded instrumented code as part of tests used during the `configure` process. To only count overflows reported in the software itself, we further filtered the set of valid packages (and the errors considered for each package) by those that load instrumented code outside of the `configure` step, leaving 1172 packages that contained C/C++ code that was successfully built and had some semblance of being tested.

¹¹<http://clang.debian.net/>

¹²http://popcon.debian.org/sourcemax/by_vote

Table V. Summary of Overflows in Top Debian Packages

Overflow Type	Count	% of Overflowing	% of All
Signed Overflow	74	17%	6%
Shift Errors	175	42%	14%
Signed Negation	18	4%	1%
Division Errors	1	0%	0%
Unsigned Overflow	356	85%	30%
Unsigned Negation	65	15%	5%
Any Undefined Overflow	198	47%	16%
Any Overflow	414	100%	35%
Total Packages	1173		

6.6.2. *Results.* A summary of our findings for these 1172 packages is shown in Table V, and a more detailed list of packages is available online¹³ or upon request.

We found that 413 (35%) of the packages triggered one of the recognized overflows, with 197 (16% overall) involving undefined integer behavior. Unsigned overflow was found to be the most common overflow type, representing 85% of the overflowing packages and 30% overall. These numbers are a lower bound on the true prevalence of overflows as they represent only those overflows that occurred during the packaging process and is further limited by the often minor amount of testing this involves.

Of the packages listed in the top 10,000, we found undefined overflows other than shift errors in 78 packages. Errors found in each are shown in Table VI, broken down by type of undefined behavior encountered. Division errors did not occur in any of these packages. After shift errors, the most common type of undefined integer behavior was signed multiplication (64), followed by signed addition (32), signed negation (17), and signed subtraction (15).

We have not yet filed bugs for these, but plan to do so after more investigation. Undefined behavior in these packages is particularly important to address due to the potentially high impact should these errors be the source of bugs or vulnerabilities.

In summary, we conducted a broad automated investigation of integer overflows in 1172 of the top Debian packages and found 35% triggered integer overflows and 16% invoked undefined integer behavior. Moreover, these errors were discovered using *only the relatively few test cases included with the packages during installation*, without any serious testing for overflows. This highlights the prevalence of integer overflows in real software, as well as how dangerously common undefined integer behavior is in software used every day by many users.

7. PERFORMANCE EVALUATION

As discussed in the previous section, incorrectly handling integers is a prevalent problem even in mature applications and it is all too easy for a skilled developer to introduce a dangerous overflow bug. Similar to how tools such as Valgrind are commonly used today to mitigate the prevalence of often-subtle memory safety errors, we believe tools like IOC are critical to helping developers understand and mitigate the insidious occurrences of integer overflows.

One important measure of such a tool's utility is the impact it has on performance. To evaluate the overheads a developer can expect to see when using IOC, in this section we report the overhead incurred on the SPEC CPU 2006 benchmarks and on two popular server applications, using a variety of checking configurations.

For all runs we used IOC as it is most commonly used: recoverable checks, detailed reporting enabled, deduplication enabled. One result of this is that repeated overflows at the same source location include the costs of marshaling the details of the failed check, calling into the runtime, using an atomic exchange operation to handle deduplication, and printing details about an error the first time it is encountered. This differs from the experiments done in Section 4.2.1 in that we are evaluating the performance impact a user might experience, not evaluating alternative check

¹³<http://wdtz.org/files/ioc-debian.log>

Table VI. Occurrences of Non-Shift Undefined Behavior in Debian Packages

Package	SAdd	SSub	SMul	SNeg	Package	SAdd	SSub	SMul	SNeg
anthy	✗		✗		libonig			✗	
apcalc		✗	✗		libversion-perl	✗		✗	
apr			✗		libzip			✗	
babl	✗				lighttpd			✗	
ccrypt			✗		llvm-3.1				✗
clamav			✗		lsof			✗	
curl			✗		m4	✗		✗	
dpkg			✗		matplotlib	✗	✗		
dvipsk-ja			✗		mono	✗			
eina	✗				mysql-5.5	✗	✗	✗	✗
erlang			✗	✗	ncbi-blast+			✗	
ffcall	✗				neko			✗	
flex-old					ocaml	✗	✗	✗	✗
ftnchek			✗		openafs			✗	
gap			✗		orc	✗	✗	✗	
gauche			✗	✗	pari	✗	✗		
gcc-4.6	✗	✗	✗	✗	pcb			✗	
gcc-4.7	✗	✗	✗	✗	pcre3	✗	✗	✗	
gcc-avr	✗	✗	✗		pdl			✗	
gcc-msp430		✗	✗		perl	✗	✗	✗	✗
gcj-4.6	✗	✗	✗	✗	pixmap	✗		✗	
gcj-4.7	✗	✗	✗	✗	postgresql-9.1	✗	✗	✗	✗
glib2.0	✗				protobuf-c			✗	✗
gnome-keyring			✗		psensor	✗			
gnuradio	✗				python2.6	✗		✗	
gpsd			✗		qhull			✗	
gsl				✗	racket	✗		✗	
gst-plugins-base0.10	✗		✗		ruby1.9.1			✗	✗
gst-plugins-base1.0	✗		✗		scm			✗	
guile-1.6			✗	✗	sqlite3			✗	
guile-1.8			✗		sqlite			✗	
guile-2.0			✗		texlive-bin	✗		✗	
haveged			✗		tla	✗		✗	✗
lcms2	✗		✗		vavoom				✗
libdap			✗		vim	✗		✗	
libgda5			✗		wireshark			✗	
libimager-perl			✗		zsh-beta			✗	
libmcrypt			✗		zsh			✗	
liboggz			✗						

implementations. These experiments were conducted on an otherwise idle 2.5 Ghz i7-2860QM with 16 GB RAM, with all dynamic scaling and multicore features disabled. Reported numbers are the median of at least three runs; we observed negligible variance within each set of runs.

7.1. Compiler

The compiler used for these experiments was chosen to be similar to that we expect users to have available. At the time of the experiment not all features from our research prototype had been incorporated in the mainstream version of Clang, so for these experiments we used a modified version of LLVM/Clang based on r170089 (Dec 13th, 2012). Since then, full support has been added.

7.2. Check configurations

The five check configurations used are as follows:

- **Full:** All checks supported: signed, unsigned, shift, division, and both implicit and explicit conversion checks.

Table VII. Performance overhead of IOC on the SPEC CPU 2006 benchmark suite

Benchmark Name	Baseline Time (s)	Full		Full-Shift		Full-Exp		Default		Undefined	
		Raw	O/H	Raw	O/H	Raw	O/H	Raw	O/H	Raw	O/H
400.perlbench	460.1	657.1	42.8%	669.7	45.6%	666.4	44.8%	551.7	19.9%	490.5	6.6%
401.bzip2	641.4	926.2	44.4%	940.4	46.6%	954.0	48.7%	842.7	31.4%	749.4	16.8%
403.gcc	361.7	407.6	12.7%	406.0	12.2%	402.8	11.4%	399.1	10.3%	379.1	4.8%
429.mcf	326.4	363.1	11.3%	363.3	11.3%	362.4	11.0%	361.5	10.8%	361.3	10.7%
445.gobmk	574.6	816.0	42.0%	815.6	41.9%	811.3	41.2%	734.5	27.8%	735.7	28.0%
456.hammer	821.1	1120.4	36.5%	1120.4	36.4%	1135.5	38.3%	1124.3	36.9%	1117.4	36.1%
458.sjeng	629.8	860.6	36.7%	859.7	36.5%	861.6	36.8%	851.9	35.3%	837.5	33.0%
462.libquantum	416.6	570.4	36.9%	602.3	44.6%	570.5	37.0%	569.0	36.6%	569.0	36.6%
464.h264ref	682.9	1515.9	122.0%	1163.0	70.3%	1513.9	121.7%	1507.9	120.8%	1503.0	20.1%
471.omnetpp	320.7	335.9	4.8%	339.2	5.8%	334.1	4.2%	334.0	4.2%	335.1	4.5%
473.astar	482.5	560.2	16.1%	572.0	18.5%	559.5	16.0%	552.4	14.5%	566.0	17.3%
483.xalancbmk	271.6	305.3	12.4%	305.4	12.4%	296.8	9.3%	292.3	7.6%	282.4	4.0%
433.milc	392.7	482.0	22.7%	481.7	22.7%	481.9	22.7%	481.1	22.5%	480.5	22.3%
444.namd	481.4	555.9	15.5%	556.1	15.5%	556.0	15.5%	557.3	15.8%	555.5	15.4%
447.dealII	376.6	574.2	52.5%	584.3	55.2%	570.8	51.6%	565.8	50.2%	369.6	-1.8%
450.soplex	271.2	318.7	17.5%	317.7	17.1%	316.8	16.8%	316.8	16.8%	317.4	17.0%
453.povray	211.4	234.7	11.0%	233.2	10.3%	235.7	11.5%	233.5	10.4%	235.7	11.5%
470.lbm	388.2	439.7	13.3%	439.8	13.3%	440.8	13.6%	439.8	13.3%	440.7	13.5%
482.sphinx3	691.7	928.9	34.3%	927.6	34.1%	826.4	19.5%	826.4	19.5%	828.4	19.8%
Geometric Mean			28.8%		27.8%		28.0%		24.6%		19.9%

- **Full-Shift:** All checks other than shift checking, as shifts are frequently misused which can lead to high overheads or a large number of false positives. In conjunction with Full, this also helps demonstrate the cost of checking these shifts.
- **Full-Explicit:** All check types other than explicit conversions. These are often less interesting because they are well-understood due to straightforward semantics and clear visibility at the source level.
- **Default:** Signed, unsigned, shift, and division checks. This configuration is important as it is the default set of checks enabled when integer checking is enabled on a recent version of Clang. Support for conversion checks has not yet been added at the time of writing, and the default checks may change to include value-losing implicit conversions in the future.
- **Undefined:** Only checks for undefined integer behavior: signed, shift, and division. Developers are generally most interested in seeing these kinds of errors reported.

7.3. SPEC CPU2006

The SPEC CPU benchmarks are CPU-bound, arithmetic-intensive programs, which means that they are likely to incur higher checking overheads than do many other common applications that are network-, disk- or UI-intensive. We instrumented the C/C++ benchmarks of SPEC CPU2006 with all five configurations of IOC, as well as a baseline configuration. Our results are shown in Fig. VII. The measurements used the `ref` inputs, as required for official performance measurements with SPEC. Each reported number is the median of at least three runs; with a relative standard deviation (i.e., the coefficient of variation) less than 0.08%. Depending on the checks enabled, the geometric mean overhead was between 20.0% (undefined only) and 28.8% (all checks enabled).

Comparing the Full and Default configurations, we observed that disabling all conversion checks improved performance by 4.2%. Removing the explicit conversion checks (Full vs. Full-Explicit) resulted in only 0.8% improvement. Disabling shift checks (Full vs. Full-Shift) only improved performance overall by 1%, but it had a significant impact on 464.h264ref because this program frequently misuses shift operations. Enabling only the undefined checks incurs an average overhead of less than 20%, with no benchmark incurring over 37% overhead. 464.h264ref saw a significant improvement here as well due to its use of unsigned overflow in tight loops. Overall, other than 464.h264ref, all overheads were less than 60%.

Table VIII. Mean response times of Apache’s httpd under different IOC configurations

Req. Size	Baseline	Full		Default		Undefined	
	Sec/Req	Raw	O/H	Raw	O/H	Raw	O/H
1 kB	0.780	0.807	3.461%	0.780	-0.077%	0.778	-0.256%
10 kB	2.528	2.491	-1.479%	2.515	-0.514%	2.520	-0.340%
100 kB	22.034	22.035	0.003%	22.035	0.002%	22.034	-0.001%
1 MB	224.633	224.648	0.007%	224.647	0.006%	224.653	0.009%
10 MB	2252.290	2253.480	0.052%	2253.170	0.039%	2252.890	0.026%

Table IX. Performance overhead of file copy using instrumented OpenSSH server

File Size	Baseline	Full		Full-Shift		Full-Explicit		Default		Undefined	
100 MB	3.09	3.23	4.54%	3.22	4.34%	3.22	4.24%	3.12	1.00%	3.10	0.26%
1 GB	20.73	22.06	6.40%	21.97	5.93%	22.00	6.13%	21.02	1.40%	20.63	-0.50%
10 GB	189.60	204.00	7.59%	203.10	7.12%	202.30	6.69%	190.60	0.53%	190.30	0.37%

7.4. Server Applications

In addition to the integer-intensive scenario presented by CINT2006, we evaluated the performance impact of IOC on two popular server applications: Apache httpd and OpenSSH’s sshd. For httpd, we measured average response time while under load for various file sizes using the three main checking configurations. Response times were measured over a gigabit link using the ab tool shipped with httpd, using 25 concurrent connections making as many requests as possible for 200 seconds. The experiment was repeated five times for each file size and configuration, with the mean values reported in Fig. VIII. The coefficients-of-variation for the 1 KB experiments were 2.5%; others were less than 0.12%. Experiments for file sizes over 100 KB were limited by the network, hence the lack of change across check configurations for these. However, even on the smaller file sizes the check configurations had only a minor impact, with the performance differences comparable to the standard deviation for the 1KB and 10KB runs.

With sshd, we measured the performance impact by copying files from an instrumented server across a gigabit network using scp. Fig. IX shows the median of five runs for each configuration and file size. Coefficients-of-variation for the 100 M experiments were less than 0.8%; the others were less than 1.6%. Under all configurations the server was CPU-bound (files were likely cached), while the client was not. Here the instrumentation slowed the transfer rates at most by 8%. No checks included in the “Default” or “Undefined” configurations triggered dynamically while conducting our scp experiments.

7.5. Performance Summary

We observed minimal performance impact on server applications regardless of the check configuration. Even on the CPU-intensive SPEC benchmarks, average overheads were relatively low (20.0% to 28.8%), making IOC suitable for use in nightly testing or as a debugging aid. We expect these overheads can be reduced more by further improving LLVM’s handling of overflow intrinsics in its optimization passes and by removing or simplifying checks through use of a range analysis. Regardless, we believe the overheads are already sufficiently low to make IOC usable for most C/C++ software today.

8. 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 surprising, user-visible artifacts [Hodges 2008; Wikipedia 2011b]. More recently, as buffer overflows in C and C++ programs have been slowly brought under control, integer overflows have emerged as an important root cause of exploitable vulnerabilities in Internet-facing programs [Christey et al. 2011; Christey and Martin 2007].

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 [Wikipedia 2011a]. 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 [Brumley et al. 2007] presents two relevant pieces of data. First, it classifies integer numerical errors from MITRE's CVE database [MITRE Corporation 2013] as overflow, signedness, and 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 [Brumley et al. 2007]. BRICK [Chen et al. 2009] detects integer overflows in compiled executables using a modified Valgrind [Nethercote and Seward 2003]. 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 [Molnar et al. 2009] is also based on Valgrind, but goes further by using whitebox testing to generate inputs leading to good test coverage. IntScope [Wang et al. 2009] is a static binary analysis tool for integer vulnerabilities.

The As-if Infinitely Ranged (AIR) integer model [Dannenberg et al. 2010] is an ambitious solution that is intended to be used in deployed software. It simply provides well-defined semantics for most of C/C++'s integer-related undefined behaviors. AIR provides a strong invariant—every integer operation either produces the mathematically correct result or else traps—while being carefully designed to minimally constrain the optimizer. An alternative online solution is provided by libraries such as SafeInt [LeBlanc 2004] and IntegerLib [CERT 2006], 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.

9. 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 observe that some of the intentional uses are written with signed instead of unsigned integer types, triggering undefined behaviors in C and C++. 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

- BRUMLEY, D., CHIUEH, T., JOHNSON, R., LIN, H., AND SONG, D. 2007. RICH: Automatically protecting against integer-based vulnerabilities. In *Proc. of the Symp. on Network and Distributed Systems Security (NDSS)*. San Diego, CA, USA.
- CERT. 2006. IntegerLib, a secure integer library. <http://www.cert.org/secure-coding/IntegerLib.zip>.
- CHEN, P., WANG, Y., XIN, Z., MAO, B., AND XIE, L. 2009. 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, 208–215.
- CHRISTEY, S. AND MARTIN, R. A. 2007. Vulnerability type distributions in CVE. Tech. report, MITRE Corporation. May. <http://cwe.mitre.org/documents/vuln-trends.html>.
- CHRISTEY, S., MARTIN, R. A., BROWN, M., PALLER, A., AND KIRBY, D. 2011. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. Tech. report, MITRE Corporation. September. <http://cwe.mitre.org/top25>.
- CLANG. 2011. clang: a C language family frontend for LLVM. <http://clang.llvm.org/>; accessed 21-Sept-2011.
- DANNENBERG, R. B., DORMANN, W., KEATON, D., SEACORD, R. C., SVOBODA, D., VOLKOVITSKY, A., WILSON, T., AND PLUM, T. 2010. As-if infinitely ranged integer model. In *Proc. of the 21st Intl. Symp. on Software Reliability Engineering (ISSRE 2010)*. San Jose, CA, USA, 91–100.
- DIETZ, W., LI, P., REGEHR, J., AND ADVE, V. 2012. Understanding integer overflow in *c/c++*. In *Proceedings of the 2012 International Conference on Software Engineering*. ICSE 2012. IEEE Press, Piscataway, NJ, USA, 760–770.
- HODGES, D. 2008. Why do Pinky and Inky have different behaviors when Pac-Man is facing up? http://donhodes.com/pacman_pinky_explanation.htm; accessed 21-Sept-2011.
- INTEL. 2013. Intel VTune Amplifier XE 2013. <http://software.intel.com/en-us/intel-vtune-amplifier-xe>.
- LATTNER, C. AND ADVE, V. 2004. 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.
- LEBLANC, D. 2004. Integer handling with the C++ SafeInt class. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp>.
- LEBLANC, D. 2011. Author’s blog: Integer handling with the C++ SafeInt class. <http://safeint.codeplex.com/>.
- LEVESON, N. G. AND TURNER, C. S. 1993. An investigation of the Therac-25 accidents. *Computer* 26, 7, 18–41.
- MITRE CORPORATION. 2002. CVE-2002-0639: Integer overflow in sshd in OpenSSH. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0639>.
- MITRE CORPORATION. 2010. CVE-2010-2753: Integer overflow in Mozilla Firefox, Thunderbird and SeaMonkey. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2753>.
- MITRE CORPORATION. 2013. Common Vulnerability and Exposures. <http://cve.mitre.org/>.
- MOLNAR, D., LI, X. C., AND WAGNER, D. A. 2009. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proc. of the 18th USENIX Security Symposium*. 67–82.
- NETHERCOTE, N. AND SEWARD, J. 2003. Valgrind: A program supervision framework. In *Proc. of the 3rd Workshop on Runtime Verification*. Boulder, CO.
- WANG, T., WEI, T., LIN, Z., AND ZOU, W. 2009. 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.

WIKIPEDIA. 2011a. Arbitrary-precision arithmetic. http://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic; accessed 21-Sept-2011.

WIKIPEDIA. 2011b. Pac-Man. <http://en.wikipedia.org/w/index.php?title=Pac-Man&oldid=450692749#Split-screen>; accessed 21-Sept-2011.