Undefined Behavior: What Happened to My Code?

Xi Wang Haogang Chen Alvin Cheung Zhihao Jia[†]
Nickolai Zeldovich M. Frans Kaashoek
MIT CSAIL [†]*Tsinghua University*

Abstract

System programming languages such as C grant compiler writers freedom to generate efficient code for a specific instruction set by defining certain language constructs as undefined behavior. Unfortunately, the rules for what is undefined behavior are subtle and programmers make mistakes that sometimes lead to security vulnerabilities. This position paper argues that the research community should help address the problems that arise from undefined behavior, and not dismiss them as esoteric C implementation issues. We show that these errors do happen in real-world systems, that the issues are tricky, and that current practices to address the issues are insufficient.

1 Introduction

A difficult trade-off in the design of a systems programming language is how much freedom to grant the compiler to generate efficient code for a target instruction set. On one hand, programmers prefer that a program behaves identically on all hardware platforms. On the other hand, programmers want to get high performance by allowing the compiler to exploit specific properties of the instruction set of their hardware platform. A technique that languages use to make this trade-off is labeling certain program constructs as *undefined behavior*, for which the language imposes no requirements on compiler writers.

As an example of undefined behavior in the C programming language, consider integer division with zero as the divisor. The corresponding machine instruction causes a hardware exception on x86 [17, 3.2], whereas PowerPC silently ignores it [15, 3.3.38]. Rather than enforcing uniform semantics across instruction sets, the C language defines division by zero as undefined behavior [19, 6.5.5], allowing the C compiler to choose an efficient implementation for the target platform. For this specific example, the compiler writer is not forced to produce an exception when a C program divides by zero, which allows the C compiler for the PowerPC to use the instruction that does not produce an exception. If the C language had insisted on an exception for division by zero, the C compiler would have to synthesize additional instructions to detect division by zero on PowerPC.

Some languages such as C/C++ define many constructs as undefined behavior, while other languages, for example Java, have less undefined behavior [7]. But the existence of undefined behavior in higher-level languages such as Java shows this trade-off is not limited to low-level system languages alone.

C compilers trust the programmer not to submit code that has undefined behavior, and they optimize code under that assumption. For programmers who accidentally use constructs that have undefined behavior, this can result in unexpected program behavior, since the compiler may remove code (e.g., removing an access control check) or rewrite the code in a way that the programmer did not anticipate. As one summarized [28], "permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose."

This paper investigates whether bugs due to programmers using constructs with undefined behavior happen in practice. Our results show that programmers do use undefined behavior in real-world systems, including the Linux kernel and the PostgreSQL database, and that some cases result in serious bugs. We also find that these bugs are tricky to identify, and as a result they are hard to detect and understand, leading to programmers brushing them off incorrectly as "GCC bugs." Finally, we find that there are surprisingly few tools that aid C programmers to find and fix undefined behavior in their code, and to understand performance implications of undefined behavior. Through this position paper, we call for more research to investigate this issue seriously, and hope to shed some

APSys '12, July 23-24, 2012, Seoul, S. Korea

```
if (!msize)
    msize = 1 / msize;  /* provoke a signal */
```

Figure 1: A division-by-zero misuse, in `lib/mpi/mpi-pow.c` of the Linux kernel, where the entire code will be optimized away.

light on how to treat the undefined behavior problem more systematically.

2 Case Studies

In this section, we show a number of undefined behavior cases in real-world systems written in C. For each case, we describe what C programmers usually expect, how representative instruction sets behave (if the operation is non-portable across instruction sets), and what assumptions a standard-conforming C compiler would make. We demonstrate unexpected optimizations using two popular compilers, GCC 4.7 and Clang 3.1, on Linux/x86-64 with the default optimization option `-O2` only, unless noted otherwise.

2.1 Division by Zero

As mentioned earlier, at the instruction set level, x86 raises an exception for a division by zero [17, 3.2], while MIPS [22, A.6] and PowerPC [15, 3.3.38] silently ignore it. A division by zero in C is undefined behavior [19, 6.5.5], and a compiler can thus simply assume that the divisor is always non-zero.

Figure 1 shows a division-by-zero misuse in the Linux kernel. From the programmer's comment it is clear that the intention is to signal an error in case `msize` is zero. When compiling with GCC, this code behaves as intended on an x86, but not on a PowerPC, because it will not generate an exception. When compiling with Clang, the result is even more surprising. Clang assumes that the divisor `msize` must be non-zero—on any system—since otherwise the division is undefined. Combined with this assumption, the zero check `!msize` becomes always false, since `msize` cannot be both zero and non-zero. The compiler determines that the whole block of code is unreachable and removes it, which has the unexpected effect of removing the programmer's original intention of guarding against the case when `msize` is zero.

Division by zero may cause trickier problems when the compiler reorders a division [23]. Figure 2 shows a failed attempt to defend against a division by zero in PostgreSQL. When `arg2` is zero, the code invokes an error handling routine `ereport(ERROR, ...)` that internally signals an error and does not return to the call site. Therefore, the PostgreSQL programmer believed that the division by zero would never be triggered.

```
if (arg2 == 0)
    ereport(ERROR, (errcode(ERRCODE_DIVISION_BY_ZERO),
                    errmsg("division by zero")));
/* No overflow is possible */
PG_RETURN_INT32((int32) arg1 / arg2);
```

Figure 2: An unexpected optimization voids the division-by-zero check, in `src/backend/utils/adt/int8.c` of PostgreSQL. The call to `ereport(ERROR, ...)` will raise an exception.

However, the programmer failed to inform the compiler that the call to `ereport(ERROR, ...)` does not return. This implies that the division will *always* execute. Combined with the assumption that the divisor must be non-zero, on some platforms (e.g., Alpha, S/390, and SPARC) GCC moves the division *before* the zero check `arg2 == 0`, causing division by zero [5]. We found seven similar issues in PostgreSQL, which were noted as "GCC bugs" in source code comments.

2.2 Oversized Shift

Intuitively, a logical left or right shift of an n-bit integer by n or more bits should produce 0, since all bits from the original value are shifted out. Surprisingly, this is false at both the instruction set and the C language level. For instance, on x86, 32- and 64-bit shift instructions truncate the shift amount to 5 and 6 bits, respectively [17, 4.2], while for PowerPC, the corresponding numbers of truncation bits are 6 and 7 [15, 3.3.13.2]. As a result, shifting a 32-bit value 1 by 32 bits produces 1 on x86, since 32 is truncated to 0, while the result is 0 on PowerPC.

In C, shifting an n-bit integer by n or more bits is undefined behavior [19, 6.5.7]. A compiler can thus assume that the shift amount is at most $n-1$. Under this assumption, the result of left-shifting 1 is always non-zero, no matter what the shift amount is, and this can lead to unexpected program behavior.

As an illustration, consider the code fragment from the ext4 file system in Linux, shown in Figure 3. The code originally contained a security vulnerability where a division by zero could be triggered when mounting the file system [1, CVE-2009-4307].

Particularly, since `sbi->s_log_groups_per_flex` is read from disk, an adversary can craft an ext4 file system with that value set to 32. In that case, `groups_per_flex`, which is `1 << 32`, becomes 0 on PowerPC. A programmer discovered that it would be used as a divisor later; to avoid the division by zero, the programmer added the zero check `groups_per_flex == 0` [4].

As discussed earlier, Clang assumes that the left shift for calculating `groups_per_flex` is always non-zero. As a result, it concludes that the check is redundant and thus

```
groups_per_flex = 1 << sbi->s_log_groups_per_flex;
/* There are some situations, after shift the
   value of 'groups_per_flex' can become zero
   and division with 0 will result in fixpoint
   divide exception */
if (groups_per_flex == 0)
    return 1;
flex_group_count = ... / groups_per_flex;
```

Figure 3: A failed attempt to fix a division-by-zero due to oversized shift [4], in `fs/ext4/super.c` of the Linux kernel.

```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
    ...
}
```

Figure 4: A signed integer overflow check, in `fs/open.c` of the Linux kernel, which uses GCC's `-fno-strict-overflow` to prevent the check from being removed.

removes it. This essentially undoes the intent of the patch and leaves the code as vulnerable as the original.

2.3 Signed Integer Overflow

A common misbelief is that signed integer operations always silently wrap around on overflow using two's complement, just like unsigned operations. This is false at the instruction set level, including older mainframes that use one's complement, embedded processors that use saturation arithmetic [18], and even architectures that use two's complement. While most x86 signed integer instructions do silently wrap around, there are exceptions, such as signed division that traps for `INT_MIN`$/-1$ [17, 3.2]. On MIPS, signed addition and subtraction trap on overflow, while signed multiplication and division do not [22, A.6].

In C, signed integer overflow is undefined behavior [19, 6.5]. A compiler can assume that signed operations do not overflow. For example, both GCC and Clang conclude that the "overflow check" $x + 100 < x$ with a signed integer x is always false, since they assume signed overflow is impossible. Some programmers were shocked that GCC turned the check into a no-op, leading to a harsh debate between the C programmers and the GCC developers [2].

Figure 4 shows another example from the `fallocate` system call implementation in the Linux kernel. Both `offset` and `len` are from user space, which is untrusted, and thus need validation. Note that they are of the signed integer type `loff_t`.

```
int vsnprintf(char *buf, size_t size, ...)
{
    char *end;
    /* Reject out-of-range values early.
       Large positive sizes are used for
       unknown buffer sizes. */
    if (WARN_ON_ONCE((int) size < 0))
        return 0;
    end = buf + size;
    /* Make sure end is always >= buf */
    if (end < buf) { ... }
    ...
}
```

Figure 5: A pointer wraparound check, in `lib/vsprintf.c` of the Linux kernel, which uses GCC's `-fno-strict-overflow` to prevent the check from being removed.

The code first rejects negative values of `offset` and `len`, and checks whether `offset + len` exceeds some limit. According to the comment "Check for wrap through zero too," the programmer clearly realized that the addition may overflow and bypass the limit check. The programmer then added the overflow check `offset + len < 0` to prevent the bypass.

However, GCC is able to infer that both `offset` and `len` are non-negative at the point of the overflow check. Along with the assumption that the signed addition cannot overflow, GCC concludes that the sum of two non-negative integers must be non-negative. This means that the check `offset + len < 0` is always false and GCC removes it. Consequently, the generated code is vulnerable: an adversary can pass in two large positive integers from user space, the sum of which overflows, and bypass all the sanity checks. The Linux kernel uses GCC's `-fno-strict-overflow` to disable such optimizations.

2.4 Out-of-Bounds Pointer

A pointer holds a memory address. Contrary to some expectations, an n-bit pointer arithmetic operation does not always yield an address wrapped around modulo 2^n. Consider the x86 family [17]. The limit at which pointer arithmetic wraps around depends on the memory model, for example, 2^{16} for a near pointer, 2^{20} for a huge pointer on 8086, and 2^{64} for a flat 64-bit pointer on x86-64.

The C standard states that when an integer is added to or subtracted from a pointer, the result should be a pointer to the same object, or just one past the end of the object; otherwise the behavior is undefined [19, 6.5.6]. By this assumption, pointer arithmetic never wraps, and the compiler can perform algebraic simplification on pointer comparisons.

However, some programs rely on this undefined behavior to do bounds checking. Figure 5 is a code snip-

```
unsigned int
tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    if (!tun)
        return POLLERR;
    ...
}
```

Figure 6: An invalid null pointer check due to null pointer dereference, in `drivers/net/tun.c` of the Linux kernel, which uses GCC's `-fno-strict-overflow` to prevent such checks from being removed.

pet from the Linux kernel. The check `end < buf` assumes that when `size` is large, `end` (i.e., `buf + size`) will wrap around and become smaller than `buf`. Unfortunately, both GCC and Clang will simplify the overflow check `buf + size < buf` to `size < 0` by eliminating the common term `buf`, which deviates from what the programmer intended. Specifically, on 32-bit systems Clang concludes that `size < 0` cannot happen because the preceding check already rejects any negative `size`, and eliminates the entire branch.

An almost identical bug was found in Plan 9's `sprint` function [10]. CERT later issued a vulnerability note against GCC [3]. The Linux kernel uses GCC's `-fno-strict-overflow` to disable such optimizations.

2.5 Null Pointer Dereference

GCC, like most other C compilers, chooses memory address 0 to represent a null pointer. On x86, accessing address 0 usually causes a runtime exception, but it can also be made legitimate by memory-mapping address 0 to a valid page. On ARM, address 0 is by default mapped to hold exception handlers [20].

In C, dereferencing a null pointer is undefined behavior [19, 6.5.3]. Compilers can thus assume that all dereferenced pointers are non-null. This assumption sometimes leads to undesirable behavior.

Figure 6 shows an example from the Linux kernel. The code dereferences `tun` via `tun->sk`, and only afterward does it validate that `tun` is non-null. Given a null `tun`, it was expected that this null-check-after-dereference bug would either cause a kernel oops as a result of the `tun->sk` dereference, or return an error code due to the null pointer check (e.g., when address 0 is mapped). Neither was considered a serious vulnerability.

However, an unexpected optimization makes this bug exploitable. When GCC sees the dereference, it assumes that `tun` is non-null, and removes the "redundant" null pointer check. An attacker can then continue to run the rest of the function with `tun` pointing to address 0, leading to privilege escalation [9]. The Linux kernel started using GCC's `-fno-delete-null-pointer-checks` to disable such optimizations.

```
struct iw_event {
    uint16_t len;  /* Real length of this stuff */
    ...
};
static inline char * iwe_stream_add_event(
    char *   stream,        /* Stream of events */
    char *   ends,          /* End of stream */
    struct iw_event *iwe,   /* Payload */
    int      event_len )    /* Size of payload */
{
    /* Check if it's possible */
    if (likely((stream + event_len) < ends)) {
        iwe->len = event_len;
        memcpy(stream, (char *) iwe, event_len);
        stream += event_len;
    }
    return stream;
}
```

Figure 7: A strict aliasing violation, in `include/net/iw_handler.h` of the Linux kernel, which uses GCC's `-fno-strict-aliasing` to prevent possible reordering.

2.6 Type-Punned Pointer Dereference

C gives programmers the freedom to cast pointers of one type to another. Pointer casts are often abused to reinterpret a given object with a different type, a trick known as *type-punning*. By doing so, the programmer expects that two pointers of different types point to the same memory location (i.e., aliasing).

However, the C standard has strict rules for aliasing. In particular, with only a few exceptions, two pointers of different types do *not* alias [19, 6.5]. Violating strict aliasing leads to undefined behavior.

Figure 7 shows an example from the Linux kernel. The function first updates `iwe->len`, and then copies the content of `iwe`, which contains the updated `iwe->len`, to a buffer `stream` using `memcpy`. Note that the Linux kernel provides its own optimized `memcpy` implementation. In this case, when `event_len` is a constant 8 on 32-bit systems, the code expands as follows.

```
iwe->len = 8;
*(int *)stream = *(int *)((char *)iwe);
*((int *)stream + 1) = *((int *)((char *)iwe) + 1);
```

The expanded code first writes 8 to `iwe->len`, which is of type `uint16_t`, and then reads `iwe`, which points to the same memory location of `iwe->len`, using a different type `int`. According to the strict aliasing rule, GCC concludes that the read and the write do not happen at the same memory location, because they use different pointer types, and reorders the two operations. The generated code thus copies a stale `iwe->len` value [27]. The

```
struct timeval tv;
unsigned long junk;       /* XXX left uninitialized
                             on purpose */
gettimeofday(&tv, NULL);
srandom((getpid() << 16)
        ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

Figure 8: An uninitialized variable misuse for random number generation, in `lib/libc/stdlib/rand.c` of the FreeBSD libc, where the seed computation will be optimized away.

Linux kernel uses `-fno-strict-aliasing` to disable optimizations based on strict aliasing.

2.7 Uninitialized Read

A local variable in C is *not* initialized to zero by default. A misconception is that such an uninitialized variable lives on the stack, holding a "random" value. This is not true. A compiler may assign the variable to a register (e.g., if its address is never taken), where its value is from the last instruction that modified the register, rather than from the stack. Moreover, on Itanium if the register happens to hold a special not-a-thing value, reading the register traps except for a few instructions [16, 3.4.3].

Reading an uninitialized variable is undefined behavior in C [19, 6.3.2.1]. A compiler can assign any value not only to the variable, but also to expressions derived from the variable.

Figure 8 shows such a misuse in the `srandomdev` function of FreeBSD's libc, which also appears in DragonFly BSD and Mac OS X. The corresponding commit message says that the programmer's intention of introducing `junk` was to "use stack junk value," which is left uninitialized intentionally, as a source of entropy for random number generation. Along with current time from `gettimeofday` and the process identification from `getpid`, the code computes a seed value for `srandom`.

Unfortunately, the use of `junk` does not introduce more randomness from the stack. GCC assigns `junk` to a register. Clang further eliminates computation derived from `junk` completely, and generates code that does *not* use either `gettimeofday` or `getpid`.

3 Disabling Offending Optimizations

Experienced C programmers know well that code with undefined behavior can result in surprising results, and many compilers support flags to selectively disable certain optimizations that exploit undefined behavior. One reason for these optimizations, however, is to achieve good performance. This section briefly describes some of these flags, their portability across compilers, and the impact of optimizations that exploit undefined behavior on performance.

3.1 Flags

One way to avoid unwanted optimizations is to lower the optimization level, and see if the bugs like the ones in the previous section disappear. Unfortunately, this workaround is incomplete; for example, GCC still enables some optimizations, such as removing redundant null pointer checks, even at `-O0`.

Both GCC and Clang provide a set of fine-grained workaround options to explicitly disable certain optimizations, with which security checks that involve undefined behavior are not optimized away. Figure 9 summarizes these options and how they are adopted by four open-source projects to disable optimizations that caused bugs. The Linux kernel uses all these workarounds to disable optimizations, the FreeBSD kernel and PostgreSQL keep some of the optimizations, and the Apache HTTP server chooses to enable all these optimizations and fix its code instead. Currently neither GCC nor Clang has options to turn off optimizations that involve division by zero, oversized shift, and uninitialized read.

3.2 Portability

A standard-conforming C compiler is *not* obligated to provide the flags described in the previous subsection. For example, one cannot turn off optimizations based on signed integer overflow when using IBM's XL and Intel's C compilers (even with `-O0`). Even for the same option, each compiler may implement it in a different way. For example, `-fno-strict-overflow` in GCC does not fully enforce two's complement on signed integers as `-fwrapv` does, usually allowing more optimizations [26], while in Clang it is merely a synonym for `-fwrapv`. Furthermore, the same workaround may appear as different options in two compilers.

3.3 Performance

To understand how disabling these optimizations may impact performance, we ran SPECint 2006 with GCC and Clang, respectively, and measured the slowdown when compiling the programs with all the three `-fno-*` options shown in Figure 9. The experiments were conducted on a 64-bit Ubuntu Linux machine with an Intel Core i7-980 3.3 GHz CPU and 24 GB of memory. We noticed slowdown for 2 out of the 12 programs, as detailed next.

456.hmmer slows down 7.2% with GCC and 9.0% with Clang. The first reason is that the code uses an `int` array index, which is 32 bits on x86-64, as shown below.

Undefined behavior	GCC workaround	Linux kernel	FreeBSD kernel	PostgreSQL	Apache
division by zero	N/A				
oversized shift	N/A				
signed integer overflow	`-fno-strict-overflow` (or `-fwrapv`)	✓		✓	
out-of-bounds pointer	`-fno-strict-overflow` (or `-fwrapv`)	✓		✓	
null pointer dereference	`-fno-delete-null-pointer-checks`	✓			
type-punned pointer dereference	`-fno-strict-aliasing`	✓	✓	✓	
uninitialized read	N/A				

Figure 9: GCC workarounds for undefined behavior adopted by several popular open-source projects.

```
int k;
int *ic, *is;
...
for (k = 1; k <= M; k++) {
    ...
    ic[k] += is[k];
    ...
}
```

As allowed by the C standard, the compiler assumes that the signed addition `k++` cannot overflow, and rewrites the loop using a 64-bit loop variable. Without the optimization, however, the compiler has to keep `k` as 32 bits and generate extra instructions to sign-extend the index `k` to 64 bits for array access. This is also observed by LLVM developers [14].

Surprisingly, by running OProfile we found that the most time-consuming instruction was not the sign extension but loading the array base address `is[]` from the stack in each iteration. We suspect that the reason is that the generated code consumes one more register for loop variables (i.e., both 32 and 64 bits) due to sign extension, and thus spills `is[]` on the stack.

If we change the type of `k` to `size_t`, then we no longer observe any slowdown with the workaround options.

462.libquantum slows down 6.3% with GCC and 11.8% with Clang. The core loop is shown below.

```
quantum_reg *reg;
...
// reg->size:          int
// reg->node[i].state: unsigned long long
for (i = 0; i < reg->size; i++)
    reg->node[i].state = ...;
```

With strict aliasing, the compiler is able to conclude that updating `reg->node[i].state` does not change `reg->size`, since they have different types, and thus moves the load of `reg->size` out of the loop. Without the optimization, however, the compiler has to generate code that reloads `reg->size` in each iteration.

If we add a variable to hold `reg->size` before entering the loop, then we no longer observe any slowdown with the workaround options.

While we observed only moderate performance degradation on two SPECint programs with these workaround options, some previous reports suggest that using them would lead to a nearly 50% drop [6], and that re-enabling strict aliasing would bring a noticeable speed-up [24].

4 Research Opportunities

Compiler improvements. One approach to eliminate bugs due to undefined behavior is to require compilers to detect undefined behavior and emit good warnings. For example, in Figure 1, a good warning would read "removing a zero check `!msize` at line *x*, due to the assumption that `msize`, used as a divisor at line *y*, cannot be zero." However, current C compilers lack such support, and adding such support is difficult [21].

Flagging all unexpected behavior statically is undecidable [13]. Therefore, C compilers provide options to insert runtime checks on undefined behavior, such as GCC's `-ftrapv` (for signed integer overflow) and Clang's `-fcatch-undefined-behavior`. Similar tools include the IOC integer overflow checker [11]. They help programmers catch undefined behavior at run time. However, these checks cover only a subset of undefined behavior that occurs on particular execution paths with given input, and are thus incomplete.

Another way to catch bugs due to undefined behavior is to define "expected" semantics for the constructs that have undefined behavior, and subsequently check if the compiled code after optimizations has the same program semantics as the non-optimized one. Unfortunately, determining program equivalence is undecidable in general [25], but it might be possible to devise heuristics for this problem.

Bug-finding tools. Bug finding tools, such as Clang's built-in static analyzer and the KLEE symbolic execution engine [8], are useful for finding undefined behavior. However, these tools often implement different C semantics from the optimizer, and miss undefined behavior the optimizer exploits. For example, both Clang's static

analyzer and KLEE model signed integer overflow as wrapping, and thus are unable to infer that the check `offset + len < 0` in Figure 4 will vanish.

Improved standard. Another approach is to outlaw undefined behavior in the C standard, perhaps by having the compiler or runtime raise an error for any use of undefined behavior, similar to the direction taken by the KCC interpreter [12].

The main motivation to have undefined behavior is to grant compiler writers the freedom to generate efficient code for a wide range of instruction sets. It is unclear, however, how important that is on today's hardware. A research question is to re-assess whether the performance benefits outweigh the downsides of undefined behavior, and whether small program changes can achieve equivalent performance, as in Section 3.3.

5 Conclusion

This paper shows that understanding the consequences of undefined behavior is important for both system developers and the research community. Several case studies of undefined behavior in real-world systems demonstrate it can result in subtle bugs that have serious consequences (e.g., security vulnerabilities). Current approaches to catching and preventing bugs due to undefined behavior are insufficient, and pose interesting research challenges: for example, systematically identifying the discrepancy between programmers' understanding and compilers' realization of undefined constructs is a hard problem.

Acknowledgments

We thank John Regehr, Linchun Sun, and the anonymous reviewers for their feedback. This research was partially supported by the DARPA CRASH program (#N66001-10-2-4089).

References

[1] Common vulnerabilities and exposures (CVE). `http://cve.mitre.org/`.

[2] `assert(int+100 > int)` optimized away. Bug 30475, GCC, 2007. `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475`.

[3] C compilers may silently discard some wraparound checks. Vulnerability Note VU#162289, US-CERT, 2008. `http://www.kb.cert.org/vuls/id/162289`.

[4] ext4: fixpoint divide exception at `ext4_fill_super`. Bug 14287, Linux kernel, 2009. `https://bugzilla.kernel.org/show_bug.cgi?id=14287`.

[5] postgresql-9.0: FTBFS on sparc64, testsuite issues with int8. Bug 616180, Debian, 2011. `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=616180`.

[6] D. Berlin. Re: changing "configure" to default to "gcc -g -O2 -fwrapv ...". `http://lists.gnu.org/archive/html/autoconf-patches/2006-12/msg00149.html`, 2006.

[7] J. Bloch and N. Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley Professional, 2005.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th OSDI*, San Diego, CA, December 2008.

[9] J. Corbet. Fun with NULL pointers, part 1. `http://lwn.net/Articles/342330/`, 2009.

[10] R. Cox. Re: plan9port build failure on Linux (debian). `http://9fans.net/archive/2008/03/89`, 2008.

[11] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proc. of the 34th ICSE*, pages 760–770, Zurich, Switzerland, June 2012.

[12] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proc. of the 39th POPL*, pages 533–544, Philadelphia, PA, January 2012.

[13] C. Ellison and G. Roşu. Defining the undefinedness of C. Technical report, University of Illinois, April 2012.

[14] D. Gohman. The nsw story. `http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-November/045730.html`, 2011.

[15] *Power ISA*. IBM, 2010. `http://www.power.org/`.

[16] *Intel Itanium Architecture Software Developer's Manual, Volume 1: Application Architecture*. Intel, 2010.

[17] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel, 2012.

[18] *ISO/IEC TR 18037:2006, Programming languages – C – Extensions to support embedded processors*. ISO/IEC, 2006.

[19] *ISO/IEC 9899:2011, Programming languages – C*. ISO/IEC, 2011.

[20] B. Jack. Vector rewrite attack: Exploitable NULL pointer vulnerabilities on ARM and XScale architectures. In *CanSecWest 2007*, Vancouver, BC, Canada, April 2007.

[21] C. Lattner. What every C programmer should know about undefined behavior #3/3. `http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html`, 2011.

[22] C. Price. *MIPS IV Instruction Set*. MIPS Technologies, 1995.

[23] J. Regehr. A guide to undefined behavior in C and C++, part 3. `http://blog.regehr.org/archives/232`, 2010.

[24] B. Rosenkränzer. Compiler flags used to speed up Linaro Android 2011.10, and future optimizations. `http://www.linaro.org/linaro-blog/2011/10/25/compiler-flags-used-to-speed-up-linaro-android-2011-10-and-future-optimizations/`, 2011.

[25] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, second edition, 2005.

[26] I. L. Taylor. Signed overflow. `http://www.airs.com/blog/archives/120`, 2008.

[27] J. Tourrilhes. Invalid compilation without -fno-strict-aliasing. `http://lkml.org/lkml/2003/2/25/270`, February 2003.

[28] J. F. Woods. Re: Why is this legal? `http://groups.google.com/group/comp.std.c/msg/dfe1ef367547684b`, 1992.