Improving Integer Security for Systems with KINT

Xi Wang Haogang Chen Zhihao Jia† Nickolai Zeldovich M. Frans Kaashoek†‡
MIT CSAIL Tsinghua IIIS

Abstract

Integer errors have emerged as an important threat to systems security, because they allow exploits such as buffer overflow and privilege escalation. This paper presents KINT, a tool that uses scalable static analysis to detect integer errors in C programs. KINT generates constraints from source code and user annotations, and feeds them into a constraint solver for deciding whether an integer error can occur. KINT introduces a number of techniques to reduce the number of false error reports. KINT identified more than 100 integer errors in the Linux kernel, the lighttpd web server, and OpenSSH, which were confirmed and fixed by the developers. Based on the experience with KINT, the paper further proposes a new integer family with NaN semantics to help developers avoid integer errors in C programs.

1 Introduction

It is well known that integer errors, including arithmetic overflow, division-by-zero, oversized shift, lossy truncation, and sign misinterpretation, can be exploited by adversaries. Recently integer errors have emerged as one of the main threats to systems security. One reason is that it is difficult for programmers to reason about integer semantics [15]. A 2007 study of the Common Vulnerabilities and Exposures (CVE) [1] suggests that they are already “number 2 for OS vendor advisories” [12], second only to buffer overflows. A recent survey [9] reviews the Linux kernel vulnerabilities in CVE from 2010 to early 2011, and confirms the finding that integer errors account for more than one third of the vulnerabilities that can be misused to corrupt the kernel and gain root privilege.

Although integer errors are a known source of problems, there are no detailed studies of integer errors in large systems. This paper’s first contribution is a detailed study of integer errors in the Linux kernel, using a new static analysis tool called KINT that we will present in the rest of this paper. We conclude that integer errors are prevalent in all of the subsystems in Linux. We also found 105 new errors for which our patches have been accepted by the Linux kernel community. Finally, we found that two integer errors previously reported in the CVE database were fixed incorrectly, highlighting the difficulty of reasoning about integer semantics.

Figure 1: Patched code for the CVE-2011-1745 vulnerability in the Linux AGP driver. The original code did not have the overflow check \( pg\_start + page\_count < pg\_start \). In that case, an adversary could provide a large \( pg\_start \) value from user space to bypass the check \( pg\_start + page\_count < pg\_start \). This leads to out-of-bounds memory writes in later code.

In applying the tool to the Linux kernel, we found that the state-of-the-art in static analysis tools for finding integer errors have trouble achieving high coverage and avoiding false error reports when applied to large software systems. For example, both \texttt{PRE}fix+Z3 [27] and SmartFuzz [25] use symbolic execution to explore possible paths, but large systems have an exponentially large number of potential paths to explore, making it infeasible to achieve high coverage. Moreover, previous tools (e.g., \texttt{PRE}fix+Z3) generate many error reports that do not correspond to actual integer errors [27].

This paper introduces a scalable static analysis for finding integer errors, along with a number of automated and programmer-driven techniques to reduce the number of generated reports, implemented in a tool called KINT. Similar to previous analysis tools, KINT generates a constraint to represent the condition under which an integer error may occur, and uses an off-the-shelf solver to see if it is possible to satisfy the constraint and thus trigger the integer error. Unlike previous tools based on symbolic execution, KINT statically generates a constraint capturing the path condition leading to an integer error, as in Saturn [37], which allows KINT to scale to large systems while maintaining high coverage.

A problem for symbolic execution tools and KINT is the large number of error reports that can be generated for a complex system. To illustrate why it is necessary to reduce the number of error reports, consider the code snippet shown in Figure 1. This example illustrates a correct and widely used pattern for guarding against addition overflow by performing the addition and checking whether the result overflowed. Such checks are prevalent in systems code, including most parts of the Linux kernel.
However, a tool that signaled an error for every integer operation that goes out of bounds would incorrectly flag the overflow check itself as an error, because the check’s addition can overflow. In addition to common overflow check idioms, there are a number of other sources for false error reports, such as complex invariants that hold across the entire program which are difficult for an automated tool to infer, and external invariants that programmers assume, such as a number of CPUs not overflowing 2^{32}.

This paper provides several contributions to help developers effectively find and deal with integer errors, as follows. First, we provide a pragmatic definition of integer errors that avoids reporting common idioms for overflow checking. Second, we introduce a whole-program analysis for KINT that can capture certain invariants in a way that scales to large programs and reduces the number of false errors. Third, because our automated analysis still produces a large number of error reports for Linux (125,172), we introduce range annotations that allow programmers to inform KINT of more complex invariants that are difficult to infer automatically, and thus help reduce false error reports from KINT. Fourth, we introduce a family of overflow-checked integers for C that help programmers write correct code. Finally, we contribute a less error-prone API for memory allocation in the Linux kernel that avoids a common source of integer errors, inspired by Andrew Morton.

Although we focus on the Linux kernel, we believe KINT’s ideas are quite general. We also applied KINT to the lighttpd web server and OpenSSH, and found bugs in those systems too.

The rest of this paper is organized as follows. §2 differentiates KINT from previous work on integer error detection. §3 presents a case study of integer errors in the Linux kernel. §4 outlines several approaches to dealing with integer errors. §5 presents KINT’s design for generating constraints, including KINT’s integer semantics. §6 evaluates KINT using the Linux kernel and known CVE cases. §7 proposes the NaN integer family. §8 summarizes our conclusions.

2 Related work

There are a number of approaches taken by prior work to address integer errors, and the rest of this section outlines the relation between this paper and previous work by considering each of these approaches in turn.

Static analysis. Static analysis tools are appealing to find integer errors, because they do not require the availability of test inputs that tickle an integer error, which often involve subtle corner cases. One general problem with static analysis is reports of errors that cannot be triggered in practice, termed false positives.

One class of static analysis tools is symbolic model checking, which systematically explores code paths for integer errors by treating input as symbolic values and pruning infeasible paths via constraint solving. Examples include PREFIX+Z3 [27], KLEE [7], LLBMC [24], SmartFuzz [25], and IntScope [34]. While these tools are effective for exploring all code paths through a small program, they suffer from path explosion when applied to the entire Linux kernel.

KINT is carefully designed to avoid path explosion on large systems, by performing costly constraint solving at the level of individual functions, and by statically generating a single path constraint for each integer operation. This approach is inspired by Saturn [37].

PREFIX+Z3 [27], a tool from Microsoft Research, combines the PREFIX symbolic execution engine [6] with the Z3 constraint solver to find integer errors in large systems. PREFIX+Z3 proposed checking precise out-of-bounds conditions for integer operations using a solver. PREFIX, however, explores a limited number of paths in practice [6], and the authors of PREFIX+Z3 confirmed to us that their tool similarly stopped exploring code paths after a fixed threshold, possibly missing errors. The authors used some techniques to reduce the number of false positives, such as ignoring reports involving explicit casts and conversions between unsigned and signed. Despite these techniques, when applying their tool to 10 million lines of production code, the authors found that the tool generated a large number of false error reports, such as the overflow check described in the introduction.

Verification tools such as ArC (now eCv) [13] and Frame-C’s Jessie plugin [26] can catch integer errors, but they accept only a restrictive subset of C (e.g., no function pointers) and cannot apply to systems like the Linux kernel.

Static analysis tools that do not keep track of sanity checks cannot precisely pinpoint integer errors. For example, a simple taint analysis that warns about untrusted integers used in sensitive sinks (e.g., allocation) [8, 16] would report false errors on correctly fixed code, such as the code shown in Figure 1.

The range checker from Stanford’s metacompiler [3] eliminates cases where a user-controlled value is checked against some bounds, and reports unchecked integer uses. A similar heuristic is used in a PREFIX-based tool from Microsoft [30]. This approach will miss integer errors due to incorrect bounds checking since it does not perform reasoning on the actual values of the bounds. KINT avoids these issues by carefully generating constraints that include path conditions.

Runtime detection. An advantage of runtime detection for integer errors is fewer false positives. Runtime integer error detection tools insert checks when generating
A naïve programmer may expect the result of an \( n \)-bit arithmetic operation to be equal to that of the corresponding mathematical (\( \infty \)-bit) operation—in other words, the result should fall within the bounds of the \( n \)-bit integer. Integer errors, therefore, are bugs that arise when the programmer does not properly handle the cases when \( n \)-bit arithmetic diverges from the mathematically expected result. However, not every integer overflow is an integer error, as described in §1 and shown in Figure 1.

In this section, we present a case study of integer errors in the Linux kernel, which will help motivate the rest of this paper. Figure 2 summarizes the integer errors we discovered in the Linux kernel as part of this case study. Each line represents a patch that fixes one or more integer errors; the number is shown in the “Error” column if it is more than one. An operation may have a subscript \( s \) or \( u \) to indicate whether it operates on signed or unsigned integers, respectively. As we will describe in the rest of this section, this case study shows that integer errors are a significant problem, and that finding and fixing integer errors is subtle and difficult.

### Case studies

**Case studies.** Integer overflows are a well-known problem, and a number of security vulnerabilities have been discovered due to integer errors. Many of the tools above find or mitigate integer errors, and have noted the complexity involved in reasoning about integer errors [15]. In particular, PREfix+Z3 was applied to over 10 millions lines of production code, and the authors of that tool found 31 errors, but provided few details. We are not aware of any detailed study of integer errors and their consequences for a complete OS kernel, which we provide in the next section.

### 3 Case study

A naïve programmer may expect the result of an \( n \)-bit arithmetic operation to be equal to that of the corresponding mathematical (\( \infty \)-bit) operation—in other words, the result should fall within the bounds of the \( n \)-bit integer. Integer errors, therefore, are bugs that arise when the programmer does not properly handle the cases when \( n \)-bit arithmetic diverges from the mathematically expected result. However, not every integer overflow is an integer error, as described in §1 and shown in Figure 1.

In this section, we present a case study of integer errors in the Linux kernel, which will help motivate the rest of this paper. Figure 2 summarizes the integer errors we discovered in the Linux kernel as part of this case study. Each line represents a patch that fixes one or more integer errors; the number is shown in the “Error” column if it is more than one. An operation may have a subscript \( s \) or \( u \) to indicate whether it operates on signed or unsigned integers, respectively. As we will describe in the rest of this section, this case study shows that integer errors are a significant problem, and that finding and fixing integer errors is subtle and difficult.

### 3.1 Methodology

To find the integer errors shown in Figure 2, we applied KINT (which we describe later in this paper) to Linux, analyzed the results, and submitted reports and/or patches to the kernel developers. KINT generated 125,172 error reports for the Linux kernel. To determine whether a report was legitimate required careful analysis of the surrounding code to understand whether it can be exploited or not. We could not perform this detailed analysis for each of the reports, but we tried a number of approaches for finding real errors among these reports, as described in §5.6, including several ad-hoc ranking techniques. Thus, our case study is incomplete: there may be many more integer errors in the Linux kernel. However, we report only integer errors that were acknowledged and fixed by kernel developers.

### 3.2 Distribution

As can be seen in Figure 2, the integer errors found in this case study span a wide range of kernel subsystems, including the core kernel, device drivers, file systems, and network protocols. 78 out of the 114 errors affect both 32-bit and 64-bit architectures; 31 errors are specific to 32-bit architecture, and the other 5 are specific to 64-bit architecture.

### 3.3 Incorrect fixes for integer errors

As part of our case study, we discovered that preventing integer errors is surprisingly tricky. Using the log of changes in the kernel repository, the “# of prev. checks” column in Figure 2 reports the number of previous sanity checks that were incorrect or insufficient. The fact that this column is non-zero for many errors shows that although developers realized the need to validate those values, it was still non-trivial to write correct checks. One of the cases, scps’s autoclose timer, was fixed three times before we submitted a correct patch. We will now describe several interesting such cases.

#### 3.3.1 Incorrect bounds

Figure 3 shows an example using a magic number \( 2^{30} \) as the upper bound for `count`, a value from user space. Unfortunately, \( 2^{30} \) is insufficient to limit the value of `count` on a 32-bit system: `sizeof(struct rps_dev_flow)` is 8,
<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Module</th>
<th>Error</th>
<th>Arch</th>
<th>Impact</th>
<th>Attack vector</th>
<th># of prev. checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>drivers:drm</td>
<td>crtc</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>nouveau</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>logic error</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>vmwgfx</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>OOB read</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>i915</td>
<td>\times u (2)</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td>drivers:input</td>
<td>cma3000_d0x</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>logic error</td>
<td>–</td>
</tr>
<tr>
<td>drivers:media</td>
<td>lgdt330x</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>logic error</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>uvc</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>OOB read</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>v4l2-ctrls</td>
<td>\times u (2)</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>zorans</td>
<td>\times u (2)</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td>drivers:platform</td>
<td>panasonic-laptop</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td>drivers:staging</td>
<td>comedi</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>olpc_dcon</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>logic error</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>vt6655 / vt6656</td>
<td>\times u +_u (4)</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td>drivers:xen</td>
<td>gntdev \dagger</td>
<td>\times u (5)</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>xenbus</td>
<td>\times u +_u</td>
<td>32</td>
<td>64</td>
<td>N/A</td>
<td>not exploitable</td>
</tr>
<tr>
<td>block</td>
<td>rbd</td>
<td>\times u +_u</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>disk</td>
</tr>
<tr>
<td></td>
<td>ext4 \dagger</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>logic error</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>ceph</td>
<td>\times u (2)</td>
<td>32</td>
<td>64</td>
<td>OOB read</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>xfs</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>logic error</td>
<td>disk</td>
</tr>
<tr>
<td></td>
<td>ceph</td>
<td>index (2)</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>network</td>
</tr>
<tr>
<td></td>
<td>jffs2</td>
<td>\times u (2)</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td>kernel</td>
<td>auditisc</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>logic error</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>relayfs \dagger</td>
<td>\times u (2)</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td>mm</td>
<td>vmscan \dagger</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>logic error</td>
<td>–</td>
</tr>
<tr>
<td>net</td>
<td>ax25</td>
<td>\times u (8)</td>
<td>32</td>
<td>64</td>
<td>timer</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>\times u (4)</td>
<td>32</td>
<td>64</td>
<td>timer</td>
<td>user space</td>
<td>1 (4)</td>
</tr>
<tr>
<td></td>
<td>\times u +_u</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>network</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>\times u +_u</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>network</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>\times u +_u</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>network</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>irda</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>timer</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>netfilter</td>
<td>\times u (2)</td>
<td>32</td>
<td>64</td>
<td>wrong output</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>netrom</td>
<td>\times u (4)</td>
<td>32</td>
<td>64</td>
<td>timer</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>rps</td>
<td>\times u +_u</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>scp</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>timer</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td>\times u +_u</td>
<td>32</td>
<td>64</td>
<td>N/A</td>
<td>user space</td>
<td>–</td>
</tr>
<tr>
<td>sound</td>
<td>usb</td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>OOB write</td>
<td>usb</td>
</tr>
<tr>
<td></td>
<td>\times u</td>
<td>32</td>
<td>64</td>
<td>OOB read</td>
<td>usb</td>
<td>–</td>
</tr>
</tbody>
</table>

Figure 2: Integer errors discovered by our case study in the Linux kernel. Each line is a patch that tries to fix one or more bugs (the number is in the “Error” column if more than one). For each patch, we list the corresponding subsystem, the error operation with the number of bugs, the affected architectures (32-bit and/or 64-bit), the security impact, a description of the attack vector and affected values, and the number of previous sanity checks from the history of the Linux kernel repository that attempt to address the same problem incorrectly or insufficiently. Numbers in parentheses indicate multiple occurrences represented by a single row in the table. Nine bugs marked with \dagger were concurrently found and patched by others.
unsigned long count = /* from user space */;
if (count > 1<<30)
    return -EINVAL;
table = vmalloc(sizeof(struct rps_dev_flow_table) +
count * sizeof(struct rps_dev_flow));
...
for (i = 0; i < count; i++)
table->flow[i] = ...

Figure 3: Incorrect bounds in the receive flow steering (RPS) implementation. The magic number 1 << 30 (i.e., 2^{30}) cannot prevent integer overflow in the computation of the argument to vmalloc.

int opt = /* from user space */;
if (opt < 0 || opt > ULONG_MAX / (60 * HZ))
    return -EINVAL;
... = opt * 60 * HZ;

Figure 4: A type mismatch between the variable opt, of type int and the bounds ULONG / (60 * HZ), of type unsigned long. This mismatch voids the checks intended to prevent integer overflow in the computation opt * 60 * HZ.

ui2 yes = /* from network */;
if (yes > ULONG_MAX / sizeof(struct crush_rule_step))
    goto bad;
... = kmalloc(sizeof(struct crush_rule_step) +
yes * sizeof(struct crush_rule_step),
GFP_NOFS);

Figure 5: A malformed check in the form x > uintmax_n/b from the Ceph file system.

if (num > ULONG_MAX / sizeof(u64) - sizeof(*snapc))
    goto fail;
... = kmalloc(sizeof(*snapc) + num * sizeof(u64),
GFP_NOFS);

Figure 6: A malformed check in the form x > uintmax_n/b - a from the Ceph file system.

and multiplying it with a count holding the value 2^{30} overflows 32 bits. In that case, the allocation size for vmalloc wraps around to a small number, leading to buffer overflows later in the loop.

Using magic numbers for sanity checks is not only error-prone, but also makes code hard to maintain: developers need to check and update all such magic numbers if they want to add a new field to struct rps_dev_flow, which increases its size. A better practice is to use explicit arithmetic bounds. In this case, the allocation size is in the form of a + count \times b; a correct bounds check is count > (ULONG_MAX - a)/b.

In addition, one needs to ensure that the type of the bounds check matches that of the variable to be checked, otherwise a mismatch may void the check. Figure 4 shows one such example. Since opt is read from user space, the code checks if the computation of opt \times 60 \times HZ overflows, but the check is incorrect. On a 64-bit system, opt of type int is a 32-bit integer, while ULONG_MAX of type unsigned long is a 64-bit integer, with value 2^{64} - 1. Therefore, the upper bound ULONG_MAX / (60 \times HZ) fails to prevent a 32-bit multiplication overflow, voiding the check. A correct fix is to change the type of opt to unsigned long, to match ULONG_MAX’s type.

struct dcon_platform_data { ... u8 (*read_status)(void);
}; /* ->read_status() implementation */
static u8 dcon_read_status_to_1_5(void)
{
    if (!dcon_was_irq())
        return -1;
    ...
}
static struct dcon_platform_data *pdata = ...;
irqreturn_t dcon_interrupt(...)
{
    int status = pdata->read_status();
    if (status == -1)
        return IRQ_NONE;
    ...
}

Figure 7: An integer error in the OLPC secondary display controller driver of the Linux kernel. Since ->read_status() returns an unsigned 8-bit integer, the value of status is in the range of [0, 255], due to zero extension. Comparing status with \(-1\) will always be false, which breaks the error handling.

3.3.2 Malformed checks

As discussed in §3.3.1, the correct bounds check to avoid overflow in the expression \(a + x \times b\) is:

\[ x > a / b \]

where \(a\) and \(b\) are constants, \(x\) is an \(n\)-bit unsigned integer, and uintmax_n denotes the maximum unsigned \(n\)-bit integer \(2^n - 1\).

One common mistake is to check for \(x > \text{uintmax}_n / b\), which an adversary can bypass with a large \(x\). As shown in Figure 5, yes is read from network, and a crafted value can bypass the broken check and overflow the addition in the kmalloc allocation size, leading to further buffer overflows. Another common broken form is \(x > \text{uintmax}_n / b - a\), as shown in Figure 6.

Both forms also appeared when a developer tried to fix a similar integer error in the Linux perf tools; the developer wrote three broken checks before coming up with a correct version [31]. We will use this fix from perf as an example to demonstrate how to simplify bounds checking using NaN integers in §7.

3.3.3 Sign misinterpretation

C provides both signed and unsigned integer types, which are subject to different type conversion rules. Inconsistent choice of signedness often breaks sanity checks. For example, in Figure 7, the intent of the comparison \(\text{status} == -1\) was to check whether read_status returns \(-1\) on error. However, since the function returns an unsigned 8-bit integer, which is zero-extended to int according to C’s conversion rules, status is non-negative. Consequently, the comparison always evaluates to false (i.e., a tautological comparison), which disables the error handling. Using signed int for error handling fixes the bug.
Therefore, CVE-2008-3526 is not exploitable, and the effectively eliminates the fix.

Some C compilers even optimize away the check because they conclude that groups_per_flex (16) inconsistent. Some C compilers leaving the two values s_log_groups_per_flex potentially 1.

The result of a shift operation \[2\].

The developer added a zero check against s_log_groups_per_flex triggered by mounting a corrupted file system with a large covered a division-by-zero bug, which an adversary could.

Figure 9 shows a fix for CVE-2009-4307. A developer dis-

3.3.4 Undefined behavior

Tautological comparisons are often indicative of signed-

nean errors. Surprisingly, a simple tautological expression that compares an unsigned integer \(x\) with 0 (i.e., \(x <_u 0\)) affected several subsystems. The w1128x driver alone contained 36 such bugs, effectively disabling most of its error handling paths.

Figure 8 shows another example, the fix for CVE-2008-3526, where a sanity check tries to reject a large len and avoid overflowing the allocation size. However, the check does not work. Consider len = 0xffffffff. Since INT_MAX is 0xffffffff, the result of the left-hand side of the check is then 0x80000000. Note that len is unsigned, the left-hand side result is also treated as unsigned (i.e., \(2^{31}\)), which bypasses the check. A correct check is len > INT_MAX − sizeof(struct sctp_auth_bytes).

After discussion with the kernel developers, we came to the conclusion that len could not become that large. Therefore, CVE-2008-3526 is not exploitable, and the fix is unnecessary. Our patch was nonetheless applied to clarify the code.

Figure 9: An incorrect fix to CVE-2009-4307 in the ext4 file system [2], because oversized shifting is undefined behavior in C.

3.4 Impact

Integer errors that allow out-of-bounds writes (i.e., buffer overflow) can break the integrity of the kernel and potentially enable privilege escalation attacks. They can be exploited via network, local access, or malformed file systems on disk. Figure 3 shows a typical example of an integer error that allows out-of-bounds writes. We found a large number of such errors in ioctl, an infamous error-prone interface. There are also two interesting vulnerabilities in the sound subsystem; an adversary can exploit them by plugging in a malicious USB audio device that responds with bogus sampling rates, leading to a kernel hang, DoS, or buffer overflow.

Integer errors cause timing bugs in several network protocol implementations. For example, when a user-space application provides a large timeout argument, the internal timer can wrap around to a smaller timeout value.

Most logic related integer errors are due to tautological comparisons. These bugs would effectively disable error handling, or make the kernel behave in unanticipated ways. One example from the CAN network protocol implementation is as follows:

```
if (((errc & 0x7f) >> 8) > 127) ...  
```

The intent of the code is to test whether the error counter errc has reached certain level. However, this comparison will never be true because the left-hand side of the test, which extracts 7 bits from errc, is at most \(2^7 - 1 = 127\). The fix is to check the right bit according to the specification, using errc & 0x80.

4 Problems and approaches

As the previous section illustrated, integer errors are common, can lead to serious problems, and are difficult to fix even for experts. Thus, it is important both to find integer errors and to help developers verify their patches or write correct code in the first place.

One approach to prevent integer errors is to avoid the fixed-width arithmetic that leads to integer operations deviating from the mathematically expected semantics. Many languages, such as Python and Haskell, take this approach. However, this is not always feasible because there is a performance penalty for using infinite-precision arithmetic.
arithmetic. Moreover, the runtime and libraries of these languages are often implemented in C, and can have integer errors as well (e.g., CVE-2011-0188 in Ruby’s integer implementation). As a result, this paper focuses on helping developers find or avoid integer errors in the presence of fixed-width arithmetic, such as in C code.

Another approach to dealing with integer errors is to find them using static analysis. The key challenges in making this approach work well lie in scaling the analysis to large systems while achieving good coverage and minimizing the number of false error reports. Minimizing false errors is particularly important for verifying correctness of patches. We describe the design of our scalable static analysis tool for finding integer errors, and techniques for reducing the number of false positives, in §5.

Based on the case study, we find that many integer errors occur when computing the number of bytes to allocate for a variable-sized data structure, such as an array of fixed-sized elements. Better APIs that perform overflow-checked multiplication for the caller, similar to the `malloc` function, can help avoid this class of integer errors. To help developers avoid this common problem, we contributed `kmalloc_array(n, size)` for array allocation to the Linux kernel, which checks overflow for `n × u.size`, as suggested by Andrew Morton. This function has been incorporated in the Linux kernel since v3.4-rc1.

Finally, as illustrated by the case study, programmers can make mistakes in writing overflow checks for integer operations. One approach taken by prior work is to raise an exception every time the value of an integer expression goes out of bounds, such as in Ada or when using GCC’s `-ftrapv` flag. However, this can generate too many false positives for overflows that do not matter. §7 describes our proposal for a C language extension that helps developers deal with integer overflows in complex expressions, without forcing all expressions to avoid integer overflows.

5 Design

This section describes the design of KINT, and introduces a number of techniques that help KINT reduce the number of error reports for large systems.

5.1 Overview

Figure 10 summarizes the design of KINT. The first step in KINT’s analysis is to compile the C source code to the LLVM intermediate representation (IR), using a standard C compiler (e.g., Clang). KINT then performs three different analyses on this IR, as follows.

The first analysis, which we will call function-level analysis, instruments the IR with checks that capture the conditions under which an integer error may occur, for each individual function. KINT infers integer errors in two ways: first, KINT looks for certain expressions whose value in C is different from its mathematically expected value, and second, KINT looks for values that can violate certain invariants—for example, array indexes that can be negative, control flow conditions that are tautologically true or false, or programmer-supplied invariants.

The second analysis, called range analysis, attempts to infer range constraints on values shared between functions (e.g., arguments, return values, and shared data structures). This analysis helps KINT infer global invariants and thus reduce false error reports.

The third analysis, which we will call taint analysis, performs taint tracking to determine which values can be influenced by an untrusted source, and which values may be used in a sensitive context, such as memory allocation; some of these sources and sinks are built in, and others are provided by the programmer. This analysis helps the programmer focus on the errors that are most likely to be exploitable.

Based on the output of function-level and range analyses, KINT generates constraints under which an integer error may occur, and feeds them to a solver to determine whether that integer error can be triggered, and if so, what inputs trigger it. Finally, KINT outputs all cases that trigger integer errors, as reported by the solver, along with annotations from the taint analysis to indicate the potential seriousness of the error.

5.2 Applying KINT to Linux

To help KINT detect integer errors, the programmer can define invariants whose violation indicates an integer error. For the Linux kernel, we annotate 23 functions like `memcpy` with the invariant that the size parameter must be non-negative. Annotations are in the form

![Figure 10: KINT’s workflow. Ellipses represent data, and rectangles represent phases of KINT’s workflow.](image)
Any out-of-bounds operation violates the expectation and words, the result should fall in the
standard integer representation on modern architectures. In other words, the result should fall in the
$n$-bit integer bounds. Any out-of-bounds operation violates the expectation and suggests an error. Figure 11 lists the requirements of producing an in-bounds result for each integer operation.

### Table: In-bounds Requirements of Integer Operations

<table>
<thead>
<tr>
<th>Integer operation</th>
<th>In-bounds requirement</th>
<th>Out-of-bounds consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x +_y y$</td>
<td>$x + y &lt; 2^n$</td>
<td>undefined behavior [21, §6.5/5]</td>
</tr>
<tr>
<td>$x + a$</td>
<td>$x + a &lt; 2^n$</td>
<td>undefined behavior [21, §6.5/5]</td>
</tr>
</tbody>
</table>
| $x / y$ | $y 
eq 0$ | undefined behavior [21, §6.5/5] |
| $x \ll y$ | $y > 0$ | undefined behavior [21, §6.5/7] |

Figure 11: In-bounds requirements of integer operations. Both $x$ and $y$ are $n$-bit integers; $x, y$ denote their $n$-bit mathematical integers.

5.3 Integer semantics

KINT assumes two’s complement [20, §4.2.1], a de facto standard integer representation on modern architectures. An $n$-bit signed integer is in the bounds $-2^{n-1}$ to $2^{n-1} - 1$, with the most significant bit indicating the sign, while an $n$-bit unsigned integer is in the bounds $0$ to $2^n - 1$.

KINT assumes that programmers expect the result of an $n$-bit arithmetic operation to be equal to that of the corresponding mathematical ($\infty$-bit) operation. In other words, the result should fall in the $n$-bit integer bounds. Any out-of-bounds operation violates the expectation and suggests an error. Figure 11 lists the requirements of producing an in-bounds result for each integer operation.

5.4 Function-level analysis

The focus of function-level analysis is to detect candidate integer errors at the level of individual functions. The analysis applies to each function in isolation in order to scale to large code sizes.

5.4.1 Bounds check insertion

KINT treats any integer operation that violates the in-bounds requirements shown in Figure 11 as a potential integer error. To avoid false errors, such as when programmers explicitly check for overflow using an overflowing expression, KINT reports an error only if an out-of-bounds value is observable [14] outside of the function. A value is observable if it is passed as an argument to another function, used in a memory load or store (e.g., as an address or the value being stored), returned by the function, or can lead to undefined behavior (e.g., dividing by zero).

At the IR level, KINT flags potential integer errors by inserting a call to a special function called kint bug on which takes a single boolean argument that can be true.
#define IFNAMSIZ 16

static int ax25_setsockopt(...,
    char __user *optval, int optlen)
{
    char devname[IFNAMSIZ];
    /* consider optlen = 0xffffffff */
    /* optlen is treated as unsigned: 2^32 - 1 */
    if (optlen < sizeof(int))
        return -EINVAL;
    /* optlen is treated as signed: -1 */
    if (optlen > IFNAMSIZ)
        optlen = IFNAMSIZ;
    copy_from_user(devname, optval, optlen);
    ...
}

Figure 12: An integer error in the AX.25 network protocol implementation of the Linux kernel (CVE-2009-2909). A negative optlen will bypass both sanity checks due to sign misinterpretation and reach the copy_from_user call, which interprets optlen as a large positive integer. Depending on the architecture-specific implementation, the consequence may be a silent failure, a kernel crash, or a stack overflow.

if an integer error can occur (i.e., the negation of the in-bounds requirements show in Figure 11). KINT will later invoke the solver to determine if this argument can ever be true, in which case an error report will be generated. For example, for division \( x/y \), the in-bounds requirement of which is \( y \neq 0 \), KINT inserts kint_bug_on\((y==0)\).

KINT also generates calls to kint_bug_on for invariants hard-coded in KINT or specified by the programmer:

- **Array index.** For an array index \( x \), KINT generates a call to kint_bug_on\((x < s)\).

- **Data size.** A common programmer-supplied invariant is that data size arguments to functions like memcpy be non-negative. For calls to such functions with data size argument \( x \), KINT generates a call to kint_bug_on\((x < s)\). Figure 12 shows an example of such an error.

Tautological control flow conditions, such as in Figure 7, cannot be expressed using calls to the special kint_bug_on function. KINT separately generates constraints to check for these kinds of integer errors.

### 5.4.2 Code rewriting

In order to reduce false errors and to improve performance, KINT performs a series of code transformations on the generated LLVM IR.

#### Simplifying common idioms

Explicit overflow checks can lead to complex constraints that are difficult for constraint solvers to reason about. For example, given two \( n \)-bit unsigned integers \( x \) and \( y \), a popular overflow checking idiom for \( x \times y \) is as follows:

\[
(x \times y)/u y \neq x.
\]

KINT replaces such idioms in the LLVM IR with equivalent expressions, as shown in Figure 13, by using LLVM intrinsic functions that check for overflow. This helps KINT produce simpler constraints to improve solver performance.

### Simplifying pointer arithmetic

KINT represents each pointer or memory address as a symbolic expression [33], and tries to simplify it if possible. KINT considers a pointer expression that it fails to simplify as an unconstrained integer, which can be any value within its range. Consider the following code snippet:

```c
struct pid_namespace {
    int kref;
    struct pidmap pidmap[PIDMAP_ENTRIES];
    ...;
};
struct pid_namespace *pid_ns = ...;
unsigned int last = ...;
struct pidmap *map = 
    &pid_ns->pidmap[(last + 1)/BITS_PER_PAGE];
int off = map -> pid_ns -> pidmap;
```

Assume that the offset into the structure field pidmap[] is 4 bytes, and the size of its element is 8 bytes. The symbolic expression for map and pid_ns->pidmap would be pid_ns + 4 + i \( \times 8 \) and pid_ns + 4 respectively, where the array index \( i = (\text{last} + 1) / \text{BITS_PER_PAGE} \).

Thus, the value of off, the subtraction of the two pointers, can be reduced to \((\text{pid_ns} + 4 + i \times 8) - (\text{pid_ns} + 4) = i \times 8\), which is independent from the value of pointer pid_ns. Without this rewriting, KINT would have considered off to be the result of a subtraction between two unconstrained integers, and would have flagged an error.

#### Merging memory loads

KINT employs a simple memory model: a value returned from a load instruction is unconstrained (unless the value has a range annotation). KINT further merges load instructions to reduce false errors. Consider the example below.

```c
/* arg is a function parameter */
if (arg->count < 1 || arg->count > 128)
    return -EINVAL;
int *klist = kmalloc(arg->count * sizeof(int), ...);
if (!klist)
    return -ENOMEM;
ret = copy_from_user(klist, user_ptr,
    arg->count * sizeof(int));
```

<table>
<thead>
<tr>
<th>Original expression</th>
<th>Simplified expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x + y &lt; u x )</td>
<td>uadd-overflow((x, y))</td>
</tr>
<tr>
<td>( x - y &lt; u 0 )</td>
<td>umul-overflow((x, y))</td>
</tr>
<tr>
<td>((x \times y)/u y \neq x)</td>
<td>(x &lt; u y)</td>
</tr>
<tr>
<td>((x \times y)/u y \neq x)</td>
<td>(x \times y/u y \neq x)</td>
</tr>
<tr>
<td>(x &gt; u \text{uintmax}_n - y)</td>
<td>umul-overflow((x, y))</td>
</tr>
<tr>
<td>(x &gt; u \text{uintmax}_n /u y)</td>
<td>umul-overflow((x, y))</td>
</tr>
<tr>
<td>(x &gt; u N/u y)</td>
<td>(x_{2n} \times u y_{2n} &gt; N)</td>
</tr>
</tbody>
</table>

Figure 13: Bounds checking idioms that KINT recognizes and simplifies. Here \( x, y \) are \( n \)-bit unsigned integers, and \( x_{2n}, y_{2n} \) denote their \( 2n \)-bit zero-extended values, respectively. Both uadd-overflow and umul-overflow are LLVM intrinsic functions for overflow detection.
The code correctly limits arg->count to prevent a multiplication overflow in arg->count * sizeof(int). To avoid reporting false errors, KINT must know that the value loaded from arg->count that appears in the copy_from_user call is the same as in the earlier if check.

For this purpose, KINT aggressively merges these loads of arg->count. It adopts an unsafe assumption that a pointer passed to a function argument or a global variable points to a memory location that is distinct from any other pointers [23]. By assuming that kmalloc cannot hold a pointer to arg, KINT concludes that the call to kmalloc does not modify arg->count, and merges the two loads.

Eliminating checks using compiler optimizations. As the last step in code rewriting, KINT invokes LLVM’s optimizer. For each call to kint_bug_on which KINT inserted for bounds checking, once the optimizer deduces that the argument always evaluates to false, KINT removes the call. Eliminating these calls using LLVM’s optimizer helps avoid subsequent invocations to the constraint solver.

5.5 Range analysis

One limitation of per-function analysis is that it cannot capture invariants that hold across functions. Generating constraints based on an entire large system such as the Linux kernel could lead to more accurate error reports, but constraint solvers cannot scale to such large constraints. To achieve more accurate error reports while still scaling to large systems such as the Linux kernel, KINT employs a specialized strategy for capturing certain kinds of cross-function invariants. In particular, KINT’s range analysis infers the possible ranges of values that span multiple functions (i.e., function parameters, return values, global variables, and structure fields). For example, if the value of a parameter x ranges from 1 to 10, KINT generates the range x ∈ [1, 10].

KINT keeps a range for each cross-function entity in a global range table. Initially, KINT sets the ranges of untrusted entities (i.e., the programmer-annotated sources described in §5.2) to full sets and the rest to empty. Then it updates ranges iteratively, until the ranges converge, or sets the ranges to full after a limited number of rounds.

The iteration works as follows. KINT scans through every function of the entire code base. When encountering accesses to a cross-function entity, such as loads from a structure field or a global variable, KINT retrieves the entity’s value range from the global range table. Within a function, KINT propagates value ranges using range arithmetic [18]. When a value reaches an external sink through argument passing, function returns, or stores to structure fields or global variables, the corresponding range table entry is updated by merging its previous range with the range of the incoming value.

To propagate ranges across functions, KINT requires a system-wide call graph. To do so, KINT builds the call graph iteratively. For each indirect call site (i.e., function pointers), KINT collects possible target functions from initialization code and stores to the function pointer.

KINT’s range analysis assumes strict-aliasing rules; that is, one memory location cannot be accessed as two different types (e.g., two different structs). Violations of this assumption can cause the range analysis to generate incorrect ranges.

After the range table converges or (more likely) a fixed number of iterations, the range analysis halts and outputs its range table, which will be used by constraint generation to generate more precise constraints for the solver.

5.6 Taint analysis

To help programmers focus on the highest-risk reports, KINT’s taint analysis classifies error reports by indicating whether each error involves data from an untrusted input (source), or is used in a sensitive context (sink). KINT propagates untrusted inputs across functions using an iterative algorithm similar to the range analysis which we discussed in the previous subsection.

KINT hardcodes one sensitive context: tautological comparisons. Other sensitive sinks are specified by the programmer, as described in §5.2.

5.7 Constraint generation

To detect integer errors, KINT generates error constraints based on the IR as modified and annotated by the previous three analyses. For integer errors represented by calls to kint_bug_on, KINT reports an error if the argument to kint_bug_on may be true. To detect integer errors that lead to tautological comparisons, KINT derives an error constraint from each comparison operation used for control flow: if the expression is always true or always false, KINT reports an error.

For every integer error, KINT must also verify that the error can be triggered in the program’s execution; otherwise, KINT would produce false error reports. To do this, KINT generates a path constraint for each integer operation, which encodes the constraints on the variables that arise from preceding operations in the function’s control flow, similar to Saturn [37]. These constraints arise from two sources: assignments to variables by preceding operations, and conditional branches along the execution path. Satisfying the path constraint with a set of variable assignments means that the integer operation is reachable from the beginning of the function with the given variable values. The path constraint filters out integer errors that cannot happen due to previous statements in a function, such as assignments or explicit overflow checks.

Consider loop-free programs first, using the code in Figure 12 as an example. The control flow of the code is shown in Figure 14. There are two sanity checks on opt1en before it reaches the call to copy_from_user. For clarification purposes, opt1en is renumbered every time it
is assigned a new value [28, §8.11]. Our goal is to evaluate the path constraint for the call to `copy_from_user`.

The basic algorithm works as follows. Since there is no loop, the path constraint of the call to `copy_from_user` is simply the logical OR of the constraints from each of its predecessors, namely `IF-TRUE` and `IF-FALSE`. For each of those two blocks, the constraint is a logical AND of three parts: the branching condition (for the transition from that block to `copy_from_user`), the assignment(s) in that block, and the path constraint of that block. Both `IF-TRUE` and `IF-FALSE` unconditionally jump to `copy_from_user`, so their branching conditions are simply true, which can be ignored. Now we have the following path constraint:

\[(\text{optlen}_1 = 16) \land \text{PathConstraint(} \text{IF-TRUE} \text{)}\]  
\[\lor (\text{optlen}_1 = \text{optlen}_0) \land \text{PathConstraint(} \text{IF-FALSE} \text{)}\].

By recursively applying the same algorithm to `IF-TRUE` and `IF-FALSE`, we obtain the fully expanded result:

\[((\text{optlen}_1 = 16) \land (\text{optlen}_0 > 16) \land \neg (\text{optlen}_0 < 4))\]  
\[\lor (\text{optlen}_1 = \text{optlen}_0) \land \neg (\text{optlen}_0 > 16)\]  
\[\lor (\neg (\text{optlen}_0 < 4)).\]

After computing the path constraint, KINT feeds the logical AND of the path constraint and the error constraint (i.e., `optlen_1 < 0`) into the solver to determine whether the integer operation can have an error. In this case, the solver will reply with an assignment that triggers the error: for example, `optlen_1 = -1`.

For programs that contain loops, the path constraint generation algorithm unrolls each loop once and ignores branching edges that jump back in the control flow [37].

This approach limits the growth of complexity of the path constraint, and thus sacrifices soundness for performance. The complete algorithm is shown in Figure 15.

To alleviate missing constraints due to loop unrolling, KINT moves constraints inside a loop to the outer scope if possible. Consider the following loop:

```plaintext```
for (i = 0; i < n; ++i)  
a[i] = ...;
```

KINT generates an error constraint `i < 0` since `i` is used as an array index. Simply unrolling the loop once (i.e., `i = 0`) may miss a possible integer error (e.g., if the code does not correctly limit `n`). KINT will generate a new constraint `n < 0` outside the loop, by substituting the loop variable `i` with its exit value `n` in the constraint `i < 0`.

Finally, the Booleterror constraint solver provides an API for constructing efficient overflow detection constraints [5, §3.5]. KINT invokes this API to generate constraints for additive and multiplicative operations, which reduces the solver’s running time.

5.8 Limitations

KINT will miss the following integer errors. KINT only understands code written in C; it cannot detect integer errors written in assembly language. KINT will miss conversion errors that are not caught by existing invariants (see §5.4.1). KINT merges loads in an unsafe way and thus may miss errors due to aliasing. KINT analyzes loops by unrolling them once, so it will miss integer errors caused by looping, for example, an addition overflow in an accumulation. Finally, if the solver times out, KINT may miss errors corresponding to the queried constraints.

6 Evaluation of KINT

The evaluation answers the following questions:

- Is KINT effective in discovering new integer errors in systems? (§6.1)
- How complete are KINT’s reports? (§6.2)
- What causes KINT to generate false error reports, and what annotations can a programmer provide to avoid these reports? (§6.3)
We periodically applied K\textsuperscript{willing} to fix them.

To understand what causes K\textsuperscript{willing} to report integer errors, we performed three experiments, as follows.

6.3 False errors

We conducted two bug review “marathons” to inspect reports related to allocation sizes in detail. The first inspection was in November 2011: one author applied an early version of K\textsuperscript{INT} to Linux kernel 3.1, spent 12 hours inspecting 97 bug reports and discovered the first batch of 6 exploitable bugs. The 97 reports were selected by manually matching function names that contained “ioctl,” so the additions never overflow. K\textsuperscript{INT}’s automated analyses are unaware of the rule and reports false errors for these additions, although a programmer can add an explicit annotation to specify this invariant.

Whole-kernel report analysis. For the whole Linux kernel, K\textsuperscript{INT} reported 125,172 warnings in total. After filtering for sensitive sinks, 999 are related to memory allocation sizes, 741 of which are derived from untrusted inputs.

We conducted two bug review “marathons” to inspect reports related to allocation sizes in detail. The first inspection was in November 2011: one author applied an early version of K\textsuperscript{INT} to Linux kernel 3.1, spent 12 hours inspecting 97 bug reports and discovered the first batch of 6 exploitable bugs. The 97 reports were selected by manually matching function names that contained “ioctl,” since range and taint analyses were not yet implemented.

The second inspection was in April 2012: another author applied K\textsuperscript{INT} to Linux kernel 3.4-rc1, spent 5 hours inspecting 741 bug reports, and found 11 exploitable bugs. All these bugs have been confirmed by Linux kernel developers, and the corresponding patches we submitted have been accepted into the Linux kernel. This shows that K\textsuperscript{INT}’s taint-based classification strategy is effective in helping users focus on high-risk warnings.

Single module analysis. To understand in detail the sources of false errors that K\textsuperscript{INT} reports, and how many

<table>
<thead>
<tr>
<th>Caught in original?</th>
<th>Cleared in patch?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2011-4097</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2010-3873</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2010-3865</td>
<td>accumulation ✓</td>
</tr>
<tr>
<td>CVE-2009-4307</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2008-3526</td>
<td>✓</td>
</tr>
<tr>
<td>All 32 others (+)</td>
<td>✓</td>
</tr>
</tbody>
</table>


Figure 16: The result of applying K\textsuperscript{INT} to integer errors in Linux kernel from the CVE database. For each case, we show whether K\textsuperscript{INT} catches the expected bugs in the original code, and whether K\textsuperscript{INT} determines that the bug is fixed in the patched code.

- How long does it take K\textsuperscript{INT} to analyze a large system such as the Linux kernel? ($\S$6.4)
- How important are K\textsuperscript{INT}’s techniques to reducing the number of error reports? ($\S$6.5)

All 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. The processor has 6 cores, and each core has 2 hardware threads.

6.1 New bugs

We periodically applied K\textsuperscript{INT} to the latest Linux kernel from November 2011 (v3.1) to April 2012 (v3.4-rc4), and submitted patches according to K\textsuperscript{INT}’s reports. As discussed in $\S$3, Linux kernel developers confirmed and fixed 105 integer errors. We also applied K\textsuperscript{INT} to two popular user-space applications, lighttpd and OpenSSH; the developers fixed respectively 1 and 5 integer errors reported by K\textsuperscript{INT}. The results show that K\textsuperscript{INT} is effective in finding new integer errors, and the developers are willing to fix them.

6.2 Completeness

To evaluate K\textsuperscript{INT}’s completeness, we collected 37 known integer errors in the Linux kernel from the CVE database [1] over the last three years (excluding those found by K\textsuperscript{INT}). As shown in Figure 16, K\textsuperscript{INT} is able to catch 36 out of the 37 integer errors.

K\textsuperscript{INT} misses one case, CVE-2010-3865, an addition overflow that happens in an accumulation loop. K\textsuperscript{INT} cannot catch the bug since it unrolls the loop only once.

6.3 False errors

To understand what causes K\textsuperscript{INT} to generate false error reports, we performed three experiments, as follows.

The code computes a score proportional to process $p$’s memory consumption. It sums up the numbers of different memory pages that $p$ takes, divides the result by the total number of pages to get a ratio, and scales it by 1000. When the whole system is running out of memory, the kernel kills the process with the highest score.

The patch changes the type of points from int to long because points could be large on 64-bit systems; multiplying it by 1000 could overflow and produce an incorrect score, causing an innocent process to be killed.

There is an implicit rule that the sum of these numbers of pages (e.g., from get_mm_rss) is at most totalpages, so the additions never overflow. K\textsuperscript{INT}’s automated analyses are unaware of the rule and reports false errors for these additions, although a programmer can add an explicit annotation to specify this invariant.

The second inspection was in April 2012: another author applied K\textsuperscript{INT} to Linux kernel 3.4-rc1, spent 5 hours inspecting 741 bug reports, and found 11 exploitable bugs. The 97 reports were selected by manually matching function names that contained “ioctl,” since range and taint analyses were not yet implemented.

The code computes a score proportional to process $p$’s memory consumption. It sums up the numbers of different memory pages that $p$ takes, divides the result by the total number of pages to get a ratio, and scales it by 1000. When the whole system is running out of memory, the kernel kills the process with the highest score.

The patch changes the type of points from int to long because points could be large on 64-bit systems; multiplying it by 1000 could overflow and produce an incorrect score, causing an innocent process to be killed.

There is an implicit rule that the sum of these numbers of pages (e.g., from get_mm_rss) is at most totalpages, so the additions never overflow. K\textsuperscript{INT}’s automated analyses are unaware of the rule and reports false errors for these additions, although a programmer can add an explicit annotation to specify this invariant.

Whole-kernel report analysis. For the whole Linux kernel, K\textsuperscript{INT} reported 125,172 warnings in total. After filtering for sensitive sinks, 999 are related to memory allocation sizes, 741 of which are derived from untrusted inputs.

We conducted two bug review “marathons” to inspect reports related to allocation sizes in detail. The first inspection was in November 2011: one author applied an early version of K\textsuperscript{INT} to Linux kernel 3.1, spent 12 hours inspecting 97 bug reports and discovered the first batch of 6 exploitable bugs. The 97 reports were selected by manually matching function names that contained “ioctl,” since range and taint analyses were not yet implemented.

The second inspection was in April 2012: another author applied K\textsuperscript{INT} to Linux kernel 3.4-rc1, spent 5 hours inspecting 741 bug reports, and found 11 exploitable bugs. All these bugs have been confirmed by Linux kernel developers, and the corresponding patches we submitted have been accepted into the Linux kernel. This shows that K\textsuperscript{INT}’s taint-based classification strategy is effective in helping users focus on high-risk warnings.

Single module analysis. To understand in detail the sources of false errors that K\textsuperscript{INT} reports, and how many
annotations are required to eliminate all false errors (assuming developers were to regularly run KINT against their source code), we examined every error report for a single Linux kernel module, the Unix domain sockets implementation. We chose it because its code is mature and we expected all the reports to be false errors (although we ended up finding one real error).

Initially, KINT generated 43 reports for this module. We found that all but one of the reports were false errors. To eliminate the false reports, we added 23 annotations; about half of them apply to common Linux headers, and thus are reusable by other modules. We describe a few representative annotations next.

The ranges of five variables are determined by a computation. Consider the following example:

```c
static u32 ordernum = 1 __range(0, 0xFFFFF);
...
ordernum = (ordernum+1)&0xFFFFF;
```

Since the result is masked with 0xFFFFF, the value of ordernum is up to the mask value. We specified this range using the annotation `__range(min, max)` as shown. We used this same annotation to specify the ranges of two structure fields that have ranges defined by existing macros, to specify the lower bound for `struct sock`’s `sk_sndbuf`, and to specify the upper bound of `struct sk_buff`’s `len`. In one case of a reference counter (struct dentry’s `d_count`), we are not certain whether it is possible for an adversary to overflow its value. Using `__range` we specified a “workaround” range to suppress related warnings.

For ranges that cannot be represented by constant integers on structure fields or variables, we added assumptions using a special function `kint_assume`, similar to KLEE [7]. An example use is as follows:

```c
int skb_tailroom(const struct sk_buff *skb)
{
    kint_assume(skb->end >= skb->tail);
    return skb_is_nonlinear(skb);
}
```

Some of these annotations could be inferred by a better global analysis, such as an extension of our range analysis. However, many annotations involve complex reasoning about the total number of objects that may exist at once, or about relationships between many objects in the system. These invariants are likely to require programmer annotations even with a better tool.

### 6.4 Performance

To measure the running time of KINT, we ran KINT against the source code of Linux kernel 3.4-rc1, with all modules enabled. We set the timeout for each query to the constraint solver to 1 second. KINT analyzed 8,916 files within roughly 160 minutes: 33 minutes for compilation using Clang, 87 minutes for range and taint analyses, and 37 minutes for generating constraints and solving 420,742 queries, of which 3,944 (0.94%) queries timed out. The running time for other analyses was negligible. The results show that KINT can analyze a large system in a reasonable amount of time.

### 6.5 Technique effectiveness

To evaluate the effectiveness of KINT’s techniques, we measured the running time, the total number of queries, and the number of error reports for different configurations of KINT when analyzing the Linux kernel. We start with a strawman design which generates a constraint for each integer expression as shown in Figure 11, feeds this constraint (combined with the path constraint) to the solver, and reports any satisfiable constraints as errors. We then evaluate KINT’s techniques by adding them one at a time to this strawman: observability-based bounds checking (§5.4.1), code rewriting (§5.4.2), range analysis (§5.5), and taint analysis (§5.6), using the annotations described in §5.2 and discarding reports with no source or sink classifications.

Figure 17 shows the results, which suggest that all of KINT’s techniques are important for analyzing a large system such as the Linux kernel.

### 7 Naïve integer semantics

§3 shows that writing correct overflow checks is tricky and error-prone, yet KINT generates many error reports, which makes it difficult for programmers to examine every one of them to ensure that no integer overflows remain. To help programmers nonetheless write correct code, we propose a new integer family with NaN (not-a-number) semantics: once an integer goes out of bounds, its value enters and stays in a special NaN state.

We demonstrate the use of NaN integers using an example from the Linux `perf` tools, which contains the two verbose overflow checks, as shown in Figure 18. Before this correct version, the developers proposed three incorrect checks [31], as we discussed in §3.3.2.

With NaN integers, the developers can simplify this code by declaring the appropriate variables using type

<table>
<thead>
<tr>
<th>Technique</th>
<th>Time (s)</th>
<th>Queries</th>
<th>Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strawman (§6.5)</td>
<td>834</td>
<td>770,445</td>
<td>231,003</td>
</tr>
<tr>
<td>+ Observability (§5.4.1)</td>
<td>801</td>
<td>738,723</td>
<td>201,026</td>
</tr>
<tr>
<td>+ Code rewriting (§5.4.2)</td>
<td>584</td>
<td>408,880</td>
<td>168,883</td>
</tr>
<tr>
<td>+ Range analysis (§5.5)</td>
<td>1,124</td>
<td>420,742</td>
<td>125,172</td>
</tr>
<tr>
<td>+ Taint analysis (§5.6)</td>
<td>2,238</td>
<td>420,742</td>
<td>85,017</td>
</tr>
</tbody>
</table>
The modified Figure 18

The maximum value. This choice requires programmers not to store $2^n - 1$ in a NaN integer.

The runtime overhead of NaN integers is low, since the compiler generates efficient overflow detection instructions for these checks. On x86, for example, the compiler inserts one jno instruction after the multiplication, which jumps in case of no overflow.

We compare the cost of a single multiplication $x \times y$, as well as a multiplication followed by a malloc call, in three scenarios: with no overflow check, with a manual overflow check using $(x \neq 0 \&\& y > \text{SIZE}\_\text{MAX} / x)$ and using NaN integers, and with and without a malloc call using the result, in cycles per operation over $10^6$ back-to-back operations, averaged over 1,000 runs.

Figure 20: Performance overhead of checking for overflow in $x \times y$, using a manual check $(x \neq 0 \&\& y > \text{SIZE}\_\text{MAX} / x)$ and using NaN integers, with and without a malloc call using the result, in cycles per operation over $10^6$ back-to-back operations, averaged over 1,000 runs.

8 Conclusion

This paper describes the design and implementation of KINT, a tool that uses scalable static analysis to identify integer errors. It aided in fixing more than 100 integer errors in the Linux kernel, the lighttpd web server, and OpenSSH. KINT introduces several automated and programmer-driven techniques that help reduce the number of false error reports. The error reports highlight that a common integer error is unanticipated integer wraparound caused by values from untrusted inputs, and this paper also proposes NaN integers to mitigate this problem in the future. All KINT source code is publicly available at http://pdos.csail.mit.edu/kint/.

Acknowledgments

We thank Jon Howell, Robert Morris, Yannick Moy, Armando Solar-Lezama, the anonymous reviewers, and our shepherd, Tim Harris, for their feedback. Fan Long helped implement an earlier version of range analysis. The Linux kernel, lighttpd, and OpenSSH developers reviewed our bugs reports and patches. This research was supported by the DARPA CRASH program (#N66001-10-2-4089). Zhihao Jia was supported by the National Basic Research Program of China Grant 2011CB00300 and NSFC 61033001.
References


