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.

*This is revision #2 of the paper, which corrects some mistakes found in the original version.

```
off_t j, pg_start = /* from user space */;
size_t i, page_count = ...;
int num_entries = ...;
if ((pg_start + page_count > num_entries) ||
    (pg_start + page_count < pg_start))
    return -EINVAL;
...
for (i = 0, j = pg_start; i < page_count; i++, j++)
    /* write to some address with offset j */;
```

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 > num_entries`, since `pg_start + page_count` wraps around. 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 PREfix+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., PREfix+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 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 PREfast-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

code; any violation in these checks will cause a trap at run time. Examples include GCC's `-ftrapv`, RICH [4], Archerr [11], IOC [15], blip [19], IntPatch [38], and PaX's overflow GCC plugin [29]. An alternative approach is to instrument binary executable files, such as IntFinder [10] and UQBTng [36]. Using such tools to find integer errors, however, requires carefully chosen inputs to trigger them. Because integer errors typically involve corner cases, the dynamic approaches tend to have low coverage. Because KINT is a static checker, it does not have this limitation.

Library and language support. To avoid integer errors, developers can adopt an integer library with error checks, such as CERT's IntegerLib [32, INT03-C] and SafeInt [22]. For example, developers change their code by calling a library function `addsl`(x, y) to add two signed long integers x and y. The library then performs sanity checks at run time, and invokes a preset error handler if an integer error occurs.

These integer libraries are trusted and supposed to be implemented correctly. Unfortunately, integer errors have recently been discovered in both IntegerLib and SafeInt [15].

Ada provides language support to define a ranged subtype (e.g., integers from 0 to 9). The runtime will raise an exception on any attempt to store an out-of-bounds value to variables of that subtype, and developers are responsible for handling the exception. There is a similar proposal that adds ranged integers to the C language [17]. Our NaN integer family is inspired by these designs.

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 (∞-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, sctp'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,

Subsystem	Module	Error	Arch		Impact	Attack vector	# of prev. checks
drivers:drm	crtc	$\times_u$	32	64	OOB write	user space	–
	nouveau	cmp	32	64	logic error	–	1
	vmwgfx	$\times_u$	32		OOB read	user space	–
		$\times_u$	32	64	OOB write	user space	2
	i915	$\times_u$ (2)	32		OOB write	user space	1 (2)
	savage	$\times_u$ (2)	32		OOB read	user space	–
drivers:input	cma3000_d0x	cmp	32	64	logic error	–	1
drivers:media	lgdt330x	cmp	32	64	logic error	–	1
	uvc	$\times_u$	32		OOB read	user space	–
	wl128x	cmp (36)	32	64	logic error	–	1 (36)
	v4l2-ctrls	$\times_u$ (2)	32		OOB write	user space	1 (2)
	zoran	$\times_u$ $+_s$	32	64	OOB write	user space	1
drivers:mtd	pmc551	cmp	32	64	logic error	–	1
drivers:scsi	iscsi_tcp	$\times_u$	32	64	OOB write	network	–
drivers:usb	usbtest	$\times_u$	32	64	OOB write	user space	–
		$\times_u$	32		logic error	user space	1
drivers:platform	panasonic-laptop	$\times_u$	32	64	OOB write	user space	1
drivers:staging	comedi	$\times_u$	32		OOB write	user space	1
	olpc_dcon	cmp	32	64	logic error	–	1
	vt6655 / vt6656	$\times_u$ $+_u$ (4)	32		OOB write	user space	–
drivers:xen	gntdev †	$\times_u$ (5)	32		OOB write	user space	1
	xenbus	$+_u$		64	–	–	–
block	rbd	$\times_u$ $+_u$	32		OOB write	disk	–
fs	ext4 †	cmp	32	64	logic error	–	1
		<<	32	64	DoS	disk	1 (CVE-2009-4307)
		$\times_u$	32		OOB write	disk	–
	nilfs2	$\times_u$ (2)	32		OOB read	user space	1
		$\times_u$	32	64	logic error	disk	–
	xfs	$\times_u$	32	64	OOB write	disk	1
	ceph	index (2)	32	64	OOB read	network	1 (2)
		$\times_u$	32		OOB write	network	1
		$+_u$	32	64	DoS	network	1
	jffs2	$+_u$	32	64	OOB write	disk	–
kernel	auditsc	cmp	32	64	logic error	–	–
	relayfs †	$\times_u$ (2)	32	64	OOB write	user space	–
mm	vmscan †	cmp	32	64	logic error	–	1
net	ax25	$\times_u$ (8)	32	64	timer	user space	–
		$\times_u$ (4)		64	timer	user space	1 (4)
	can	cmp	32	64	logic error	–	–
	ceph	$\times_u$ $+_u$ (2)	32		OOB write	network	1
		$+_u$	32	64	OOB write	network	–
		$\times_u, +_u$	32		OOB write	network	–
	irda	$\times_s$	32	64	timer	user space	–
	netfilter	$-_u$ (2)	32	64	wrong output	–	1 (2)
	netrom	$\times_u$ (4)	32	64	timer	user space	–
	rps	$\times_u$ $+_u$	32		OOB write	user space	1
	sctp	$\times_u$	32		timer	user space	3
		$+_u$	32	64	–	–	1 (CVE-2008-3526)
	unix	$+_s$	32	64	logic error	user space	–
sound	usb	$\times_u$	32		OOB write	usb	–
		$\times_u$	32		OOB read	usb	–

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 † 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 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`.

```
u32 yes = /* from network */;
if (yes > ULONG_MAX / sizeof(struct crush_rule_step))
    goto bad;
... = kmalloc(sizeof(*r) +
              yes * sizeof(struct crush_rule_step),
              GFP_NOFS);
```

Figure 5: A malformed check in the form $x > \text{uintmax}_n/b$ from the Ceph file system.

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

Figure 6: A malformed check in the form $x > \text{uintmax}_n/b - a$ 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 $a + \texttt{count} \times b$; a correct bounds check is $\texttt{count} > (\texttt{ULONG\_MAX} - a)/b$.

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 * 60 * 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 * 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_xo_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:

$$x >_u (\text{uintmax}_n - a)/_u b.$$

where a and b are constants, x is an n-bit unsigned integers, 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 `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.

```
u32 len = ...;
if (INT_MAX - len < sizeof(struct sctp_auth_bytes))
    return NULL;
... = kmalloc(sizeof(struct sctp_auth_bytes)
              + len, gfp);
```

Figure 8: An incorrect fix for CVE-2008-3526 from the SCTP network protocol implementation.

```
sbi->s_log_groups_per_flex = /* from disk */;
groups_per_flex = 1 << sbi->s_log_groups_per_flex;
if (groups_per_flex == 0)
    return 1;
flex_group_count = ... / groups_per_flex;
```

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

Tautological comparisons are often indicative of signedness errors. Surprisingly, a simple tautological expression that compares an unsigned integer x with 0 (i.e., $x <_u 0$) affected several subsystems. The wl128x 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 `0x7fffffff`, 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.

3.3.4 Undefined behavior

Figure 9 shows a fix for CVE-2009-4307. A developer discovered a division-by-zero bug, which an adversary could trigger by mounting a corrupted file system with a large `s_log_groups_per_flex`. To reject such illegal input, the developer added a zero check against `groups_per_flex`, the result of a shift operation [2].

However, this check turns out to be incorrect. Shifting an n-bit integer by n or more bits is undefined behavior in C, and the actual result varies across architectures and compilers. For example, when `s_log_groups_per_flex` is set to 36, which is an illegal value, 1 << 36 is essentially 1 << 4 = 16 on x86, since x86's shifting instruction truncates the amount to 5 bits. This will bypass the check, leaving the two values `s_log_groups_per_flex` (36) and `groups_per_flex` (16) inconsistent. Some C compilers even optimize away the check because they conclude that left-shifting the value 1 never produces zero [35], which effectively eliminates the fix.

Another kernel developer later revised the zero check to `groups_per_flex < 2`, which still suffers from the same problem. This issue was re-assigned CVE-2012-2100 after we reported it. A correct fix is to check `s_log_groups_per_flex` before the shift, so as to avoid undefined behavior.

In general, it is unsafe to check the result of an integer operation that may involve undefined behavior, such as shifts, divisions, and signed integer operations [35]. One should instead check the operands *before* the operation.

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 & 0x7f00) >> 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. 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 `calloc` 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 $\texttt{n} \times_u \texttt{size}$, 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.

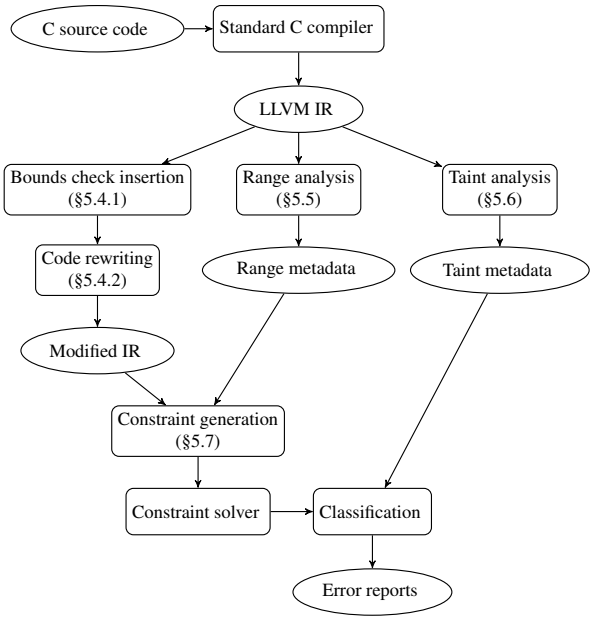

Figure 10: KINT's workflow. Ellipses represent data, and rectangles represent phases of KINT's workflow.

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

Integer operation	In-bounds requirement	Out-of-bounds consequence
$x +_s y$, $x -_s y$, $x \times_s y$	x_∞ op $y_\infty \in [-2^{n-1}, 2^{n-1}-1]$	undefined behavior [21, §6.5/5]
$x +_u y$, $x -_u y$, $x \times_u y$	x_∞ op $y_\infty \in [0, 2^n-1]$	modulo 2^n [21, §6.2.5/9]
$x /_s y$	$y \neq 0 \wedge (x \neq -2^{n-1} \vee y \neq -1)$	undefined behavior [21, §6.5.5]
$x /_u y$	$y \neq 0$	undefined behavior [21, §6.5.5]
$x << y$, $x >> y$	$y \in [0, n-1]$	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 ∞-bit mathematical integers.

(*function-name*, *parameter-index*) in a separate input to KINT. For example, (`memcpy`, 3) means that the third parameter of `memcpy` represents a data size, and KINT will check whether it is always non-negative.

Although KINT's automated analyses reduce the number of error reports significantly, applying KINT to the Linux kernel still produces a large number of false positives. In order to further reduce the number of error reports, the programmer can add range annotations on variables, function arguments, and function return values, which capture invariants that the programmer may know about. Range annotations in the Linux kernel help capture invariants that the programmer knows hold true: for example, that the `tail` pointer in an `sk_buff` is never greater than the `end` pointer. As another example, many `sysctl` parameters in the Linux kernel have lower and upper bounds encoded in the initialization code of their `sysctl` table entries. §6.3 evaluates such annotations, but we did not apply them for the case study in §3.

Finally, to help decide which of the reports are likely to be exploitable, and thus help focus on important errors, the programmer can annotate certain untrusted sources and sensitive sinks. For the Linux kernel, we annotated 20 untrusted sources. For example, (`copy_from_user`, 1) means the first parameter of `copy_from_user`, a pointer, is untrusted; KINT will mark all integers read from the pointer as untrusted. We also annotated 40 sensitive sinks, such as (`kmalloc`, 1); KINT will highlight errors the result of which is used as the first parameter of `kmalloc` (i.e., the allocation size).

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 (∞-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.

Addition, subtraction, and multiplication. The mathematical result of an n-bit signed additive or multiplicative operation should fall in $[-2^{n-1}, 2^{n-1}-1]$, and that of an unsigned operation should fall in $[0, 2^n-1]$. For example, $2^{31} \times_u 16$ is not in bounds, because the expected mathematical product 2^{35} is out of the bounds of 32-bit unsigned integers.

Division. The divisor should be non-zero. Particularly, the signed division $-2^{n-1}/_s -1$ is not in bounds, because the expected mathematical quotient 2^{n-1} is out of the bounds of n-bit signed integers (at most $2^{n-1}-1$).

Shift. For n-bit integers, the shifting amount should be non-negative and at most $n-1$. Unlike multiplication, KINT assumes that programmers are aware of the fact that a shift operation is lossy since it shifts some bits out. Therefore, KINT considers that $x << 1$ is always in bounds, but $x \times_u 2$ is not.

Conversion. KINT does not flag conversions as integer errors (even if a conversion truncates a value into a narrower type), but does precisely model the effect of the conversion, so that an integer error may be flagged if the resulting value violates some invariant (e.g., a negative array index).

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 $x/_u y$, 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`$(x <_s 0)$.
- *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`$(x <_s 0)$. 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 $x \times_u y$ is as follows:

$$(x \times_u y)/_u y \neq x.$$

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.

Original expression	Simplified expression
$x + y <_u x$	uadd-overflow(x,y)
$(x \times y)/_u y \neq x$	umul-overflow(x,y)
$x >_u \text{uintmax}_n - y$	uadd-overflow(x,y)
$x >_u \text{uintmax}_n /_u y$	umul-overflow(x,y)
$x >_u N/_u y$	$x_{2n} \times_u y_{2n} > N$

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.

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:

```
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 $i = (\texttt{last} + 1)/_u \texttt{BITS\_PER\_PAGE}$.

Thus, the value of `off`, the subtraction of the two pointers, can be reduced to $(\texttt{pid\_ns} + 4 + i \times 8) - (\texttt{pid\_ns} + 4) = i \times 8$, 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.

```
/* 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));
```

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 $x \in [1, 10]$.

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 `optlen` before it reaches the call to `copy_from_user`. For clarification purposes, `optlen` is renumbered every time it

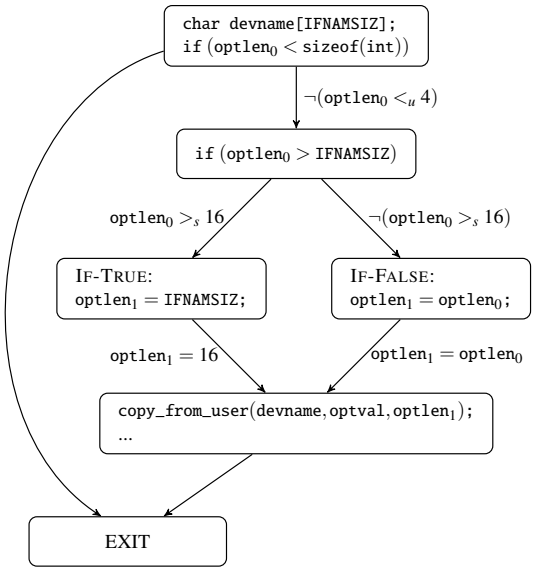

Figure 14: The control flow of the code snippet in Figure 12.

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:

$$((\texttt{optlen}_1 = 16) \wedge \text{PathConstraint}(\text{IF-TRUE})) \\ \vee((\texttt{optlen}_1 = \texttt{optlen}_0) \wedge \text{PathConstraint}(\text{IF-FALSE})).$$

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

$$((\texttt{optlen}_1 = 16) \wedge (\texttt{optlen}_0 >_s 16) \wedge \neg(\texttt{optlen}_0 <_u 4)) \\ \vee((\texttt{optlen}_1 = \texttt{optlen}_0) \wedge \neg(\texttt{optlen}_0 >_s 16) \\ \wedge \neg(\texttt{optlen}_0 <_u 4)).$$

After computing the path constraint, KINT feeds the logical AND of the path constraint and the error constraint (i.e., $\texttt{optlen}_1 <_s 0$) 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, $\texttt{optlen}_0 = -1$.

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].

function PATHCONSTRAINT(*blk*)
  if *blk* is entry **then**
    return true
  $g \leftarrow$ **false**
  for all *pred* $\in$ *blk*'s predecessors **do**
    $e \leftarrow (pred, blk)$
    if *e* is not a back edge **then**
      $br \leftarrow e$'s branching condition
      $as \leftarrow \bigwedge_i (x_i = y_i)$ for all assignments along e
      $g \leftarrow g \vee (\text{PATHCONSTRAINT}(pred) \wedge br \wedge as)$
  return g

Figure 15: Algorithm for path constraint generation.

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:

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

KINT generates an error constraint $i <_s 0$ 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 $n <_s 0$ outside the loop, by substituting the loop variable i with its exit value n in the constraint $i <_s 0$.

Finally, the Boolector 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)

	Caught in original?	Cleared in patch?
CVE-2011-4097	✓	page semantics
CVE-2010-3873	✓	CVE-2010-4164
CVE-2010-3865	accumulation	✓
CVE-2009-4307	✓	bad fix (§3.3.4)
CVE-2008-3526	✓	bad fix (§3.3.3)
All 32 others (⋆)	✓	✓

(⋆) CVE-2011-4077, CVE-2011-3191, CVE-2011-2497, CVE-2011-2022, CVE-2011-1770, CVE-2011-1759, CVE-2011-1746, CVE-2011-1745, CVE-2011-1593, CVE-2011-1494, CVE-2011-1477, CVE-2011-1013, CVE-2011-0521, CVE-2010-4649, CVE-2010-4529, CVE-2010-4175, CVE-2010-4165, CVE-2010-4164, CVE-2010-4162, CVE-2010-4157, CVE-2010-3442, CVE-2010-3437, CVE-2010-3310, CVE-2010-3067, CVE-2010-2959, CVE-2010-2538, CVE-2010-2478, CVE-2009-3638, CVE-2009-3280, CVE-2009-2909, CVE-2009-1385, CVE-2009-1265.

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

- How long does it take KINT to analyze a large system such as the Linux kernel? (§6.4)
- How important are KINT's techniques to reducing the number of error reports? (§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 KINT to the latest Linux kernel from November 2011 (v3.1) to April 2012 (v3.4-rc4), and submitted patches according to KINT's reports. As discussed in §3, Linux kernel developers confirmed and fixed 105 integer errors. We also applied KINT to two popular user-space applications, lighttpd and OpenSSH; the developers fixed respectively 1 and 5 integer errors reported by KINT. The results show that KINT is effective in finding new integer errors, and the developers are willing to fix them.

6.2 Completeness

To evaluate KINT'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 KINT). As shown in Figure 16, KINT is able to catch 36 out of the 37 integer errors.

KINT misses one case, CVE-2010-3865, an addition overflow that happens in an accumulation loop. KINT cannot catch the bug since it unrolls the loop only once.

6.3 False errors

To understand what causes KINT to generate false error reports, we performed three experiments, as follows.

CVE experiment. We first tested KINT on the patched code of the CVE cases in §6.2, expecting that ideally KINT would not report any error. The results are also shown in Figure 16. KINT reports no bugs in 33 of the 37 cases, and reports errors in 4 cases. One case, the patched code of CVE-2010-3873, contains additional integer errors that are covered by CVE-2010-4164, which KINT correctly identified; two cases contain incorrect fixes as we have shown in §3.3.3 and §3.3.4. One case is a false error in CVE-2011-4097, as detailed below.

```
long points; /* int points; */
points = get_mm_rss(p->mm) + p->mm->nr_ptes;
points += get_mm_counter(p->mm, MM_SWAPENTS);
points *= 1000;
points /= totalpages;
```

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. KINT'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, KINT 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 KINT 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 KINT 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 KINT'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 KINT 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:

```
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:

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

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.

Technique	Time (s)	Queries	Reports
Strawman (§6.5)	834	770,445	231,003
+ Observability (§5.4.1)	801	738,723	201,026
+ Code rewriting (§5.4.2)	584	408,880	168,883
+ Range analysis (§5.5)	1,124	420,742	125,172
+ Taint analysis (§5.6)	2,238	420,742	85,017

Figure 17: Effectiveness of techniques in KINT, enabling each of them one by one in order. The time is for constraint generation and solving only. The compilation time is 33 minutes for all techniques. Range and taint analyses themselves take additional 87 minutes, if enabled.

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 NaN 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.

```
size_t symsz     = /* input */;
size_t nr_events = /* input */;
size_t histsz, totalsz;
if (symsz > (SIZE_MAX - sizeof(struct hist))
            / sizeof(u64))
    return -1;
histsz = sizeof(struct hist) + symsz * sizeof(u64);
if (histsz > (SIZE_MAX - sizeof(void *))
             / nr_events)
    return -1;
totalsz = sizeof(void *) + nr_events * histsz;
void *p = malloc(totalsz);
if (p == NULL)
    return -1;
```

Figure 18: Preventing integer overflows using manual checks, which are verbose and error-prone [31].

```
nan size_t symsz     = /* input */;
nan size_t nr_events = /* input */;
nan size_t histsz, totalsz;
histsz  = sizeof(struct hist) + symsz * sizeof(u64);
totalsz = sizeof(void *) + nr_events * histsz;
void *p = malloc(totalsz);
if (p == NULL)
    return -1;
```

Figure 19: Preventing integer overflows using NaN integers (see Figure 18 for a comparison).

With NaN integers, the developers can simplify this code by declaring the appropriate variables using type `nan size_t` and removing the overflow checks, as in Figure 19. If any computation overflows, `totalsz` will be in the NaN state. To catch allocation sizes that are in the NaN state, we modify `malloc` as follows:

```
void *malloc(nan size_t size)
{
    if (isnan(size))
        return NULL;
    return libc_malloc((size_t) size);
}
```

The modified `malloc` takes a `nan size_t` as an argument and uses the built-in function `isnan`(x) to test if the argument is in the NaN state. If so, `malloc` returns `NULL`.

To help programmers insert checks for the NaN state, the type conversion rules for NaN integers are as follows:

- An integer of type T will be automatically promoted to `nan` T when used with an integer of type `nan` T.
- The resulting type of an arithmetic or comparison operation with operands of type `nan` T is also `nan` T.
- An integer of type `nan` T can be converted to T only with an explicit cast.

We implemented NaN integers by modifying the Clang compiler. The compiler inserts overflow checks for every arithmetic, conversion, and comparison operation of type `nan` T, and sets the result to NaN if any source operand is in the NaN state, or if the result went out of bounds; otherwise the operation follows standard C rules. Currently we support only unsigned NaN integers. We chose the maximum value $2^n - 1$ to represent the NaN state for n-bit unsigned integers; `isnan`(x) simply compares x with the maximum value. This choice requires programmers to not store $2^n - 1$ in a NaN integer.

	w/o `malloc`	w/ `malloc`
No check	3.00±0.01	79.03±0.01
Manual check	24.01±0.01	104.04±0.03
NaN integer check	4.05±0.17	82.03±0.05

Figure 20: Performance overhead of checking for overflow in $x \times_u y$ using a manual check (`x != 0 && y > SIZE_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.

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 $x \times_u y$, as well as a multiplication followed by a `malloc` call, in three scenarios: with no overflow check, with a manual overflow check using (`x != 0 && y > SIZE_MAX / x`), and with an overflow check using NaN integers. Figure 20 shows the results. With a single multiplication, overflow checking using NaN integers adds 1–3 cycles on average, and manual overflow checking adds 21–25 cycles. Given the negligible overhead, we believe that it is practical to replace manual overflow checks with NaN integers.

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 2011CBA00300 and NSFC 61033001.

References

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

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

[3] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy*, pages 143–159, Oakland, CA, May 2002.

[4] D. Brumley, T. Chiueh, and R. Johnson. RICH: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb–Mar 2007.

[5] R. Brummayer. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays*. PhD thesis, Johannes Kepler University, Linz, Austria, Nov 2009.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, 2000.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, Dec 2008.

[8] E. N. Ceesay, J. Zhou, M. Gertz, K. Levitt, and M. Bishop. Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs. In *Proceedings of the 3rd GI/IEEE SIG SIDAR Conference on Detection of Intrusions & Malware, and Vulnerability Assessment*, pages 1–16, Berlin, Germany, Jul 2006.

[9] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, Jul 2011.

[10] P. Chen, H. Han, Y. Wang, X. Shen, X. Yin, B. Mao, and L. Xie. IntFinder: Automatically detecting integer bugs in x86 binary program. In *Proceedings of the 11th International Conference on Information and Communications Security*, pages 336–345, Beijing, China, Dec 2009.

[11] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. ARCHERR: Runtime environment driven program safety. In *Proceedings of the 9th European Symposium on Research in Computer Security*, pages 385–406, Sophia Antipolis, France, Sep 2004.

[12] S. Christey and R. A. Martin. *Vulnerability Type Distributions in CVE*, May 2007. http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf.

[13] D. Crocker. *Verifying absence of integer overflow*, Jun 2010. http://critical.eschertech.com/2010/06/07/verifying-absence-of-integer-overflow/.

[14] R. B. Dannenberg, W. Dormann, D. Keaton, T. Plum, R. C. Seacord, D. Svoboda, A. Volkovitsky, and T. Wilson. As-if infinitely ranged integer model. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, pages 91–100, San Jose, CA, Nov 2010.

[15] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering*, Zurich, Switzerland, Jun 2012.

[16] D. Evans, J. V. Guttag, J. J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96, New Orleans, LA, Dec 1994.

[17] J. Gennari, S. Hedrick, F. Long, J. Pincar, and R. C. Seacord. Ranged integers for the C programming language. Technical Note CMU/SEI-2007-TN-027, Carnegie Mellon University, 2007.

[18] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[19] O. Horovitz. Big loop integer protection. *Phrack*, 9(60), 2002.

[20] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. Intel, 2011.

[21] *ISO/IEC 9899:2011, Programming languages — C*. ISO/IEC, 2011.

[22] D. LeBlanc. Integer handling with the C++ SafeInt class. http://msdn.microsoft.com/en-us/library/ms972705, 2004.

[23] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 317–326, Helsinki, Finland, Sep 2003.

[24] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, and Experiments*, Philadelphia, PA, Jan 2012.

[25] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th USENIX Security Symposium*, pages 67–81, Montreal, Canada, Aug 2009.

[26] Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, Orsay, France, Jan 2009.

[27] Y. Moy, N. Bjørner, and D. Sielaff. Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis. Technical Report MSR-TR-2009-57, Microsoft Research, 2009.

[28] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[29] E. Revfy. Inside the size overflow plugin. http://forums.grsecurity.net/viewtopic.php?f=7&t=3043, Aug 2012.

[30] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. Technical Report MSR-TR-2006-44, Microsoft Research, 2006.

[31] C. Schafer. [PATCH v3] perf: prevent overflow in size calculation. https://lkml.org/lkml/2012/7/19/507, Jul 2012.

[32] R. C. Seacord. *The CERT C Secure Coding Standard*. Addison-Wesley Professional, 2008.

[33] R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the 10th International Conference on Compiler Construction*, pages 118–132, Genova, Italy, Apr 2001.

[34] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb 2007.

[35] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the 3rd ACM SIGOPS Asia-Pacific Workshop on Systems*, Seoul, South Korea, Jul 2012.

[36] R. Wojtczuk. UQBTng: A tool capable of automatically finding integer overflows in Win32 binaries. In *Proceedings of the 22nd Chaos Communication Congress*, Berlin, Germany, Dec 2005.

[37] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, Long Beach, CA, Jan 2005.

[38] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Proceedings of the 15th European Symposium on Research in Computer Security*, pages 71–86, Athens, Greece, Sep 2010.