Detecting Deadlock, Double-Free and Other Abuses in a Million Lines of Linux Kernel Source

Peter T. Breuer  Simon Pickin  Maria Larrondo Petrie

Universidad Carlos III de Madrid
Departamento de Ingenería Telemática
Avenida de la Universidad, 30
Leganes, Madrid E-28911 SPAIN

Abstract

The formal analysis described here detects two so far undetected real deadlock situations per thousand C source files or million lines of code in the open source Linux operating system kernel, and three undetected accesses to freed memory, at a few seconds per file. That is notable because the code has been continuously under scrutiny from thousands of developers’ pairs of eyes. In distinction to model-checking techniques, which also use symbolic logic, the analysis uses a “3-phase” compositional Hoare-style programming logic combined with abstract interpretation. The result is a customisable post-hoc semantic analysis of C code that is capable of several different analyses at once.

1 Introduction

The decade-long development of the open source Linux kernel has proved itself to be a process largely hostile towards formal methods workers. But . . .

“...the code is the commentary” has been one of the mantras of its development, meaning that the code should be so clear in its own right that it serves as the primary reference and specification. That puts out of bounds what is usually the first question of formal methods practitioners: “what should the code do” (in order to compare with what it does do)? And on the other hand, formal methods practitioners have generally not been able to find a way into the six million lines of largely undocumented C code (that changes daily) which comprises the Linux source.

How can formal methods benefit open source? This article describes the application of a formal programming logic to the C code of the Linux kernel as it is, and the resulting analysis detects real coding errors. This is post hoc analysis, out of necessity – open source code is generated anarchically via semi-evolutionary processes; it is not written from a specification, and a specification may likely never be written for it. Neither is the coding language designed for formal methods – it is plain C, mixed with assembler, because C is the language used by the kernel authors.

There are, however, implicit specifications that the source must obey – one is “the code shall not deadlock!” Therefore formal methods do have solid ground to work on in the kernel, despite the lack of an explicit and complete specification. Among the other unwritten specifications that must be obeyed are “array bounds shall not be overflowed”, and “do not access memory which has been freed”.

Violations of these imperatives leave the operating system vulnerable in a security sense: deadlock leaves the system open to a denial of service attack – in the case of some of the deadlocks found in the sound subsystem, possibly by causing a system sound to be played for the operator (“you have mail”); array overflow leaves the system open to corruption and possible subversion
via a classic attack, while access to freed memory allows either an information leak or corruption leading to a denial of service vulnerability.

The formal analysis described here copes with the mix of C and assembler in the Linux kernel source, and the analyser is written wholly in C (thus making it easy to compile and distribute in an open source environment) and is itself licensed under an open source license. That makes the tool accessible to Linux kernel source authors, a precondition for adoption. Moreover, the analyser is configurable, which means that it is re-programmed and extended largely via reconfiguration.

Any static analysis is difficult to apply to C code, because of C’s pointer arithmetic, aliasing, side-effecting arithmetic, etc., but some notable efforts towards that end have been made. David Wagner and collaborators in particular have been active in the area (see for example [8], where Linux user space and kernel space memory pointers are given different types, so that their use can be distinguished, and [9], where C strings are abstracted to a minimal and maximal length pair and operations on them abstracted to produce linear constraints on these numbers). That research group often uses model-checking to look for violations in possible program traces of an assertion such as “chdir before any other file operations”. In contrast, the approach in this article assigns a (customisable) approximation semantics to C programs, via a (customisable) program logic for C. A more lightweight technique still is that exemplified by Jeffrey Foster’s work with CQual1, which extends the type system of C in a customisable manner. In particular, CQual has been used to detect double-spinlock or once, which means treating a short program path beyond a conditional branch that it cannot predict with precision (due to opaque calls, or other approximations), yet in practice that path may never be taken due to the way the arithmetic really works out.

Invoking the symbolisation [5] of the program. Examples include ignoring state for the time taken to generate it, or restricting to a particular set of program variables (“slicing”), or looking only at the condition of certain logical assertions rather than the state itself. In the analysis here, abstract interpretation forms a fundamental part, causing a simplification in the symbolic logic description of state that is propagated by the analysis; for example, “don’t know” is a valid literal to the analysis, thus a program variable which may take any of the values 1, 2, or 3 may be described as having the value “don’t know” in the abstraction, leading to a state s described by one atomic proposition, not a disjunct of three. And although the logic of compound statements like for, while, etc. manipulates the logic of the component statements with generality in the standard configuration, the logic of atomic statements like assignment is usually configured by the user to provide some simplification; for example an assignment to program variable x may be configured to delete references to the old value of x in the state, but not to assign a (particular) new value, thus giving an abstraction in which only the fact of assignment and reference is visible, not the value assigned or read. Two approximations are calculated: the approximated “fuzzy” state s and a predicate p, such that for the real state x:

\[ x \in s \cap p \]

The approximate state s is a guide; if it says x = 0 exactly, then if(x) is processed down the else branch.

Initially each loop is treated as traversed only zero or once, which means treating a while statement similarly to an if statement. The reduction is immediately of interest because the program states explored form a subset of the larger set of reachable states, so a program that shows up as problematic in the reduced analysis would also be problematic in a fuller analysis. The only proviso is that the condition that triggers the alarm be monotonic (increasing) in the set of states examined (an example of a non-monotonic condition would be “x=2 can never happen” – that may become false as more possible states are examined; a monotonic condition is “x=2 is a possibility”, which can only become more likely as more states are taken into consideration). But since positive p are all that can be phrased in the restricted class of predicates allowed:

1. if a condition is flagged by the analyser, it is a possible program condition;
2. if a condition is not flagged by the analyser, that does not mean that it is impossible.

(1) holds because conditions deal with abstractions and that means that flagged conditions may never arise during a real program execution - for example, the analysis may say that it is possible that program variable x is written to because it lies in a program path beyond a conditional branch that it cannot predict with precision (due to opaque calls, or other approximations), yet in practice that path may never be taken due to the way the arithmetic really works out.

It might be thought that (2) is overly cautious, because if a condition is not flagged, then it is not a reachable condition as far as the analysis can see. But the analysis does not fully take into account certain loops created by backward pointing calculated gotos, and it would be difficult to do so. In a structured program, however, all reachable paths are seen by the analyser, and a stronger conclusion applies:

1http://www.cs.umd.edu/~jfoster/cqual/. See [6, 7].
3. in a structured program, if a condition is not flagged by the analyser, that does mean that it is impossible.

This conclusion rests on part of the analysis which ensures that although loops are scanned only once, any changes seen to be made by the loop are evaluated in the broadest way, so that, for example, if the loop sets the external program variable \( x \) to 1 first time through, it may nevertheless be returned to the environment with the value “don’t know”, on the basis that repeat iterations may change it further. The same goes for the computed predicate \( p \), meaning that arbitrary loop repeats are properly taken into account.

It may sound like only a “subset of a superset” of problems are reported, but the outcome is merely some over-reporting and that is not a significant negative – kernel programming is a task of such difficulty as cannot rightly be appreciated from the outside and a few false reports are trivial compared to the effort in searching for weeks for a sporadically reported error, evoking system crashes in case of a successful test. Some of the errors reported by the analysis have been present for years in the Linux kernel code.

The program logic, as remarked, is configured rather than programmed into the analysis – it is embedded via an internal logic compiler. The way the logic interacts with other program properties such as the call graph is also configured, via a programmable rewrite engine that generates the relevant relationships (“calls”, “sleeps”, etc.) from the abstract syntax tree, and determines how the logic is applied.

These two elements of the analyser represent innovative but necessary steps introduced since the early prototype was first described in [4]. Since both the logic of C and the manner of its application to a C program are specified by the user, flexibility and – especially – reliability are obtained. Reliability accrues for the same reason as a program written in Fortran is more reliable than programmed into the analysis – it is embedded into the kernel, and keeping out the thread that would have released the spinlock is only one application of the abstract logic set out in the appendix, for completeness. Examples and statistics from real runs in the Linux kernel source are given in Section 2. Some other configurations of the analysis for other problems are discussed in Section 4.

2. Trial Run

About 1000 (1055) of the 6294 C source files in the Linux 2.6.3 kernel were checked for spinlock problems in a 24-hour period by the analyser running on a 550MHz (dual) SMP PC with 128MB RAM. About forty more files failed to parse at that time for various reasons (in one case, because of a real code error, in others because of the presence of gnu C extensions that the analyser could not cope with at that time, such as \_\_attribute\_ declarations in unexpected positions, case statement patterns matching a range instead of just a single number, array initialisations using “[\( 1,3,4 \) = \( x \) ]” notation, enumeration and typedef declarations inside code blocks, and so on). Five files showed up as suspicious under the analysis:

- total 1 instances of sleep under spinlock in sound/iss/ab/sb16_csp.c
- total 1 instances of sleep under spinlock in sound/oss/sequencer.c
- total 6 instances of sleep under spinlock in net/bluetooth/rfcomm/tty.c
- total 7 instances of sleep under spinlock in net/irda/irlmp.c
- total 3 instances of sleep under spinlock in net/irda/irttp.c

The logic used to describe C is detailed in Section 3, along with the customisations for the analysis. The theory behind the logic is set out in the appendix, for completeness. Examples and statistics from real runs in the Linux kernel source are given in Section 2. Some other configurations of the analysis for other problems are discussed in Section 4.
Table 1. Testing for sleep under spinlock in the 2.6.3 Linux kernel.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>files checked</td>
<td>1055</td>
</tr>
<tr>
<td>alarms raised</td>
<td>18 (5/1055 files)</td>
</tr>
<tr>
<td>false positives</td>
<td>16/18</td>
</tr>
<tr>
<td>real errors</td>
<td>2/18 (2/1055 files)</td>
</tr>
<tr>
<td>time taken</td>
<td>~24h</td>
</tr>
<tr>
<td>LOC</td>
<td>700K (unexpanded)</td>
</tr>
</tbody>
</table>

However, although many of the flagged constructs are indeed calls of the kernel memory allocation function `kmalloc` under spinlock, the arguments to the call render it harmless, i.e. cause it not to sleep after all. The `kmalloc` function will sleep with some arguments (with `GFP_KERNEL` as second argument, for example), and will not sleep with others (`GFP_ATOMIC` as second argument, for example). But the following is from the function `snd_sb_csp_load()` in `sb16_csp.c`:

```c
spin_lock_irqsave(&p->chip->reg_lock, flags);
unsigned char *kbuf, *_kbuf;
_kbuf = kbuf = kmalloc (size, GFP_KERNEL);
spin_unlock_irqrestore(&lock, flags);
```

so there is a real coding error here, as this `kmalloc` call can sleep, and is under spinlock. The code appears to be intended to write microcode from user-space to the sound processor chip on the sound card. In the 2.6.11 kernel, that section of code has been replaced.

The error flagged in `sound/oss/sequencer.c` is likewise real. The following code occurs in `midi_outc.c`:

```c
spin_lock_irqsave(&lock, flags);
while (n & !midi_devs[dev]->outputc(dev, data)) {
    interruptible_sleep_on_timeout(&seq_sleeper, ...);
    n--;
} spin_unlock_irqrestore(&lock, flags);
```

and the call to `interruptible_sleep_on_timeout` (which can sleep) is plainly made under spinlock. This erroneous code is still present in the 2.6.11 and 2.6.12 Linux kernel source. Alan Cox owned up to it (2.6.12.5) on the Linux kernel mailing list, thread `sleep under spinlock, sequencer.c, 2.6.12.5 dated Fri, 19 Aug 2005 19:07:09 +0100: “Yep thats a blind substitution of lock_kernel in an old tree it seems. Probably my fault. Should drop it before the sleep and take it straight after “."

The `tty.c` alarm is illusory. In `irlmp.c`, the code does use `kmalloc` under spinlock, but `kmalloc` takes an argument `GFP_ATOMIC` there that means it will not sleep. The `irtty.c` instance is similar.

Thus there are 2 real errors and 16 false alarms flagged in 1000 C source files in the Linux kernel (see Table 1). That implies (by extrapolation) that there are about 12 real deadlocks of the kind searched for to be discovered in the whole of the 2.6.3 Linux kernel source (6294 C source files). The rough indication is that the half-life of such errors without analysis is otherwise about eight kernel versions (~1 year).

3. Configuring the Analysis

The analysis requires (or allows) the logic to be specified, and thus to be modified for specific purposes. The motive was originally to make sure that the logic was implemented in a bug-free way – writing the logic directly in C made for too low-level an implementation for what is a very high-level set of concepts. A compiler into C for the program logic is incorporated into the analysis tool.

That logic compiler understands specifications

```c
txt precontext, precondition ::
    name(arguments) =
    postconditions with ctx postcontext;
```

where the `precondition` is an input, the entry condition for a code fragment, and `postconditions` is an output, a triple consisting of conditions applying to three standard exit paths from the program called `name`. These standard exit paths are respectively denominated the `normal, return` and `break` paths, or `N, R, B` paths, for short, and the output triple in the logic specification is called the NRB postcondition.

The `precontext` is a prevailing context within which the logic is evaluated. The `postcontext` is the output context, as modified. Contexts consist of a set of labelled conditions, associated with possible extra exit paths from the program, beyond the standard three, as evoked by `gos`. For example, the empty statement logic is:

```c
txt e, p::empty() = (p, F, F) with ctx e;
```

signifying that the empty statement preserves the entry condition `p` on normal exit `(p)`, and cannot exit via return `(F)` or break `(F)`. The context `(e)` is unaltered.

The specification for the `while` loop logic is given below. The normal exit from a `while` loop is with the entry condition `p` in case the loop test fails, or via normal exit with condition `n` from the body of the loop if the body is traversed, or via a break with condition `b` from the body of the loop, thus `p ∨ n ∨ b` is the normal
exit condition. The only way of taking a return exit is via return from the body of the loop. Counter-intuitively, there is no way of exiting via break, since a break from the body of the loop causes a normal exit from the loop, so the break exit condition is F.

\[ \text{ctx e, p;} : \text{while(body)} = (p / \neg p) \text{ with ctx f} \]

The post-context f lists the ways of exiting from the loop via goto and is generated by the body of the loop (from the context e given on entry).

A small deficiency in the logic given is that it does not allow for return or break out of the loop test condition. Although that is impossible in ANSI C, GNU gcc permits, as an extension, that any statement be usable as an expression (the last statement in a sequence of statements must evaluate to a value, which is returned as the value of the expression). Breaks and returns may therefore occur in the test condition, in theory. But in practice such a construction is never used, as it would break principles of good style in C.

It should be emphasised that side-effects in expressions are correctly treated by the analysis. Although the analyser evaluates expressions to a value (in the simple domain shown in Table 3), it generates a post-condition as it does so, and that post-condition is used appropriately. For example, in the while loop logic, the expression is evaluated first, to determine if it is a constant, and if it is a constant the loop is treated as a no-op or an infinite loop, as appropriate. If it is neither, the general loop logic given above is used. The post-condition generated from the expression is used as the input pre-condition p to the loop logic.

### Table 3. The domain for term evaluations contains only simple constants, arithmetic combinations, plus variables (the Linux kernel contains no floats). Logical statements are positive propositional forms. Arrows show refinement.

<table>
<thead>
<tr>
<th>const:</th>
<th>range:</th>
<th>prpn:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-1, 0, 1, \ldots)</td>
<td>[c₁, c₂], [x : p]</td>
<td>p₁ ∨ p₂</td>
</tr>
<tr>
<td>(\neg a)\</td>
<td>c₁ ∈ const</td>
<td>p₁</td>
</tr>
<tr>
<td>value:</td>
<td>atom:</td>
<td>p₁ ∨ p₂</td>
</tr>
<tr>
<td>x, y, z, \ldots</td>
<td>x &lt; c, x &gt; c</td>
<td>p, q, \ldots</td>
</tr>
<tr>
<td>c, v + v, v - c</td>
<td>x = c, x ≤ c</td>
<td>a, p[x ± c/x], p[x/y]</td>
</tr>
<tr>
<td>upper(r), lower(r)</td>
<td>x ≥ c, T, F</td>
<td>a ∈ atom, p₁ ∈ prpn</td>
</tr>
<tr>
<td>r ∈ range, c ∈ const</td>
<td>c ∈ const</td>
<td>c ∈ const</td>
</tr>
</tbody>
</table>

A very restricted class of logical statements is available, facilitating normalisation “on the fly”. This reduces the complexity of the logical statements generated; \(p \lor p\) will be normalised to \(p\) or better, for example. A reduced normal statement has no implications between any of its subexpressions. The class of atomic statements contains \(x = k\), \(x < k\), etc. Conjunctions and disjunctions are allowed, but negations are not. Inequality can be expressed as the disjunction of two ordering constraints. Terms \(x + k\) are allowed, as is \("upper[x : p]"\), which expresses the maximal value of variable \(x\) that satisfies statement \(p\) in which \(x\) appears free. The \("upper\) simply selects the second of a pair, since \("[x : p]") signifies the range (the pair) from the lowest value that \(x\) can take under \(p\) to the highest. This construct introduces abstraction, since it removes detail of the set of values satisfying \(p\).

The full logic specification for C is given in Table 2. The treatment of \(spin\_unlock\) calls, \(write\_unlock\) calls, \(read\_unlock\) calls, etc. in Linux kernel code is managed by the \(unlock\) entry in the table. These calls decrement the spinlock counter \(n\). The argument \(label\) 1 to the call is an identifier for the spinlock address that appears as an argument to the call in the syntax tree. Similarly the \(lock\) entry in the table represents the logic of the \(spin\_lock\), \(write\_lock\), \(read\_lock\), etc. calls. These calls all increment \(n\).

### Table 4. Defining initial conditions, and an objective function calculated at every syntax tree node that counts the number \(n\) of spinlocks held.

```
::initial() = (n ≤ 0);
p::objective() = upper[n:p];
```

An objective function is specified for the analysis (see Table 4). Its value is computed at each node of the syntax tree and gives the estimated upper limit of the spinlock counter \(n\) at that node. Calls to functions which can \(sleep\) when the objective is positive are eventually reported in the analysis output, thus indicating a call to a sleep function under spinlock.

The \(initial\) specification \((n ≤ 0)\) in Table 4 defines the program state at the runtime initialisation, where the spinlock counter \(n\) is zero in value. This condition is forward-propagated through the program by the analysis, using the logic defined in Table 2.

The logic is complemented by a system which acts whenever one of several flags change on a node. Whenever a change is made, the two rules in Table 5 are applied (among others). The first rule applies when a function is newly marked sleepy (\(SLEEP!\)). Then, if the \(objective\) function (here the maximal value of
Table 2. Defining the single precondition/triple NRB postcondition program logic of C to the logic compiler.

| ctx e, p::for(stmt) | = (n vb, r, F) with ctx f where ctx e, p::stmt = (n,r,b) with ctx f; |
| ctx e, p::empty()   | = (p, F, F) with ctx e; |
| ctx e, p::unlock(label l) | = (p[n+1/n], F, F) with ctx e; |
| ctx e, p::lock(label l)  | = (p[n-1/n], F, F) with ctx e; |
| ctx e, p::assembler() | = (p, F, F) with ctx e; |
| ctx e, p::function()  | = (p, F, F) with ctx e; |
| ctx e, p::sleep(label l) | = (p, F, F) with ctx e; |
| ctx e, p::sequence(s1, s2) | = (n2, r1 ∨ r2, b1 ∨ b2) with ctx g where ctx f, n1::s2 = (n2,r2,b2) with ctx g and ctx e, p::s1 = (n1,r1,b1) with ctx f; |
| ctx e, p::switch(stmt) | = (n vb, r, F) with ctx f where ctx e, p::stmt = (n,r,b) with ctx f; |
| ctx e, p::if(s1, s2) | = (n1 ∨ n2, r1 ∨ r2, b1 ∨ b2) with ctx f1 ∨ f2 where ctx e, p::s1 = (n1,r1,b1) with ctx f1 and ctx e, p::s2 = (n2,r2,b2) with ctx f2; |
| ctx e, p::while(stmt) | = (p vb, r, F) with ctx f where ctx e, p::stmt = (n,r,b) with ctx f; |
| ctx e, p::do(stmt)   | = (n vb, r, F) with ctx f where ctx e, p::stmt = (n,r,b) with ctx f; |
| ctx e, p::goto(label l) | = (F, F, F) with ctx e; |
| ctx e, p::continue() | = (F, F, p) with ctx e; |
| ctx e, p::break()    | = (F, F, p) with ctx e; |
| ctx e, p::return()   | = (F, F, F) with ctx e; |
| ctx e, p::labeled(label l) | = (pve.1, F, F) with ctx e ∩ l; |

Legend:
- assembler – gcc inline assembly code;
- sleep – calls to C functions which can sleep;
- function – calls to other C functions;
- sequence – two statements in sequence;
- switch – C case statement;
- while – C while loop;
- do – C do while loop;
- if – C conditional statement;
- labeled – labelled statements.

Table 5. Trigger/action rules which propagate flags through the syntax tree and eventually sound the alarm when sleep under spinlock is possible.

SLEEP! & OBJECTIVE_SET & OBJECTIVE ≥ 0 → aliases |= SLEEP, callers |= SLEEP
REF! & SLEEP → callers |= SLEEP, ¬REF

the spinlock count x) has already been calculated on that node (OBJECTIVE_SET), all the known aliases of the same semantic entity are also marked sleepy, as are all the known callers of this node (which will be the current function, plus all callers of aliases).

The second rule in Table 5 is triggered when a known sleepy function is referenced (REF!). Then all the callers (including the new referrer) are marked as sleepy if they were not so-marked before. The REF flag is removed as soon as it is added so every new reference triggers this rule. The action of the two rules is to efficiently create the transitive sleepy call graph.

A list is created of calls that may sleep made under possible positive spinlock. The list is compiled by the rule given in Table 6. The output() function call does the collation. Entries are added to the list when a call is (a) sleepy, and (b) the spinlock count at that node is already known and (c) is nonnegative (positive counts will be starred in the output list, but all calls will be listed). The entries are added whenever any one of these conditions becomes true when the other conditions already are true. A sample output from an analyser run is shown in Table 7.

The file being analysed for Table 7 was 592 lines long (489 nonempty, 336 lines of code, 13 functions, Mc-
Table 6. Trigger/action rule which creates the list of sleepy calls under positive spinlock.

(SLEEP & OBJECTIVE_SET & OBJECTIVE ≥ 0)! → output()

Table 7. Sleep under spinlock analysis output.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Line No</th>
<th>Symbol</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>smb_lock_server</td>
<td>66</td>
<td>down</td>
<td>0</td>
</tr>
<tr>
<td>lock_parent</td>
<td>1620</td>
<td>down</td>
<td>0</td>
</tr>
<tr>
<td>double_down</td>
<td>1646</td>
<td>down</td>
<td>0</td>
</tr>
<tr>
<td>triple_down</td>
<td>1684</td>
<td>down</td>
<td>0</td>
</tr>
<tr>
<td>double_lock</td>
<td>1712</td>
<td>double_down</td>
<td>0</td>
</tr>
<tr>
<td>lock_super</td>
<td>40</td>
<td>down</td>
<td>0</td>
</tr>
<tr>
<td>sbull_ioctl</td>
<td>184</td>
<td>interruptible_sl..</td>
<td>0</td>
</tr>
<tr>
<td>sbull_request</td>
<td>420</td>
<td>interruptible_sl..</td>
<td>1</td>
</tr>
<tr>
<td>sbull_init</td>
<td>469</td>
<td>kmalloc</td>
<td>0</td>
</tr>
</tbody>
</table>

* found 1 instances of sleep under spinlock

Table 8. Command line arguments to the analyser (“c”) are the same as used for the C compiler - shown for an analysis run on msr.c in the 2.6.3 Linux kernel.

```
c -Wp,-MD,arch/i386/kernel/.msr.o.d -nostdinc -iwithprefix include -D__KERNEL__ -Iinclude -D__KERNEL__ -Iinclude -Wall -march=i686 -Wstrict-prototypes -Wno-trigraphs -fno-strict-aliasing -fno-common -pipe -mprefered-stack-boundary=2 -Iinclude/asm-i386/mach-default -pipe -fomit-frame-pointer -DGNUC \ -DKBUILD_BASENAME=msr -DKBUILD_MODNAME=msr -o arch/i386/kernel/msr.o \ -c arch/i386/kernel/msr.c
```

Cabe’s Cyclomatic Number average per function 6.917), but the analysis dragged in about 30000 lines of extra Linux kernel code via (202) included headers and their includes. After macro expansion, the file and its includes totalled 30044 lines of code (10246 nonempty).

In-line assembler is permitted in gnuc, and appears often in the kernel, so the parser has to recognise the assembler produced by certain key constructs such as spin lock calls. The parser is based on a PRECC [3] adaptation of the widely available Lex and YACC specification for (then) ANSI C first published on the Internet by Jutta Degener in 1995 which is in turn based on a specification published in 1985 by Jeff Lee following the April 30, 1985 ANSI C draft, and apparently posted to Usenet on net.sources in 1987 by Tom Stockfish.

The PRECC adaptation was constructed with reference to the YACC grammar in the source code for gnuc 2.95 dated 2001/10/06 but was not based on it as the ANSI grammar was clearer.

The parser handles the code of both the 2.4 and 2.6 series Linux kernels. The lexer needs seeding with the names of functions which are known a priori to sleep, and the names of the spinlock lock and unlock calls. The complete list of names is given in Table 9.

4. More Applications

Double-takes of spinlocks can be detected by first constructing the transitive graph of functions which call functions which take spinlocks, and sounding the alarm at a call of such a function under spinlock.

Making the graph requires attaching the code

2Degener’s grammar is available at http://www.lysator.liu.se/c/ANSI-C-grammar-y.html.
3The Stockfish grammar is available on ftp://ftp.uu.net as the file usenet/net.sources/ansi.c.grammar.Z, being referenced in the answer to question 18.5 (“Where can I get a BNF or YACC grammar for C?”) of the comp.lang.c FAQ.
4The lang.c FAQ is available at http://www.eskimo.com/scs/C-faq/top.html.
Table 9. User-configurable function types recognized by the analyzer.

```c
class SLEEP, SPINLOCK, SPINUNLOCK, KMALLOC

# lexer: declare functions known to be sleepy
symbol SLEEP schedule
symbol SLEEP interruptible_sleep_on[_timeout]
symbol SLEEP down[_interruptible]
# from uaccess.h
symbol SLEEP verify_area, __get_user?
symbol SLEEP __{put,get}_user_bad
symbol SLEEP __copy_{to,from}_user_ll
symbol SLEEP might_sleep, [...strcpy_from_user
symbol SLEEP strnlen_user, [...clear_user

# lexer: declare functions known to be spinlocks
symbol SPINLOCK spin_lock[_irq[save]]
symbol SPINLOCK _raw_spin_lock, spin_try_lock
symbol SPINLOCK read_lock[_irqsave]
symbol SPINLOCK write_lock[_irqsave]
# lexer: declare functions known to be spinunlocks
symbol SPINUNLOCK spin_unlock[_irq[restore]]
symbol SPINUNLOCK _raw_spin_unlock
symbol SPINUNLOCK read_unlock[_irqrestore]
symbol SPINUNLOCK write_unlock[_irqrestore]
# lexer: spot calls to kmalloc - these can sleep if
# __GFP_WAIT is in second arg
symbol KMALLOC [...kmalloc
```
Table 11. Corrected test for access to kfreed memory in the 2.6.13 Linux kernel.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>files checked</td>
<td>1646</td>
</tr>
<tr>
<td>alarms raised</td>
<td>8</td>
</tr>
<tr>
<td>false positives</td>
<td>8/8</td>
</tr>
<tr>
<td>real errors</td>
<td>0/8</td>
</tr>
<tr>
<td>time taken</td>
<td>~6h</td>
</tr>
<tr>
<td>LoC</td>
<td>982K</td>
</tr>
</tbody>
</table>

the bug and easy to correct it (a question of rewriting the line in the specification that set the logic of assignment) points to the reliability obtained from using a specification-based approach to the logic.

Even with that bug, only 30 of the 1151 files examined were reported as suspicious (see Table 10). Just one of these \( \text{aic7xxx\_old.c} \) generated the majority, 209, of the alarms, another \( \text{aic7xxx\_proc.c} \) 80, another \( \text{cpqphp\_ctrl.c} \) 54, another 23, another 10, then 8, 7, 5, 4, 2, 2, 2, and the rest 1 alarm each, so the indicators pointed firmly at three or four chief culprits. Three (3) of the flagged files turned out on inspection to contain real errors. For example, \( \text{fm8010gp.c} \) contains the sequence:

```c
kfree(gp);
printk("unable to grab region 0x%x-0x%x\n",
gp->gameport.io, gp->gameport.io + 0x0f);
```

Curiously, \( \text{drivers/scsi/aic7xxx\_old.c} \) was flagged appropriately. The first alarm is raised for \( \text{temp\_p} \) in:

```c
while(current_p && temp_p) {
  if(((current_p->pci_bus==temp_p->pci_bus) && ... ){...
    kfree(temp_p);
    continue;
  ...)
... 
```

The remaining alarms on that file derived from repeated references to the same vulnerable pointer.

A repeat experiment on 1646 source files (982K LOC, unexpanded) of the Linux 2.6.12.3 kernel found that all the errors detected in the experiment on kernel 2.6.3 had been repaired, and no further real errors were detected (see Table 11). There were 8 false alarms given on 7 files (all due to a parser bug during that experiment which caused a \text{struct} field dereference to be treated like a variable). Perhaps it is not surprising that errors detected earlier had been corrected, since they had been reported to the maintainers, but there were indications from the maintenance history that in two of the three cases the error had been spotted and corrected independently.

5. Software

The source code of the software described in this article is available for download from

ftp://oboe.it.uc3m.es/pub/Programs/c-1.2.13.tgz

under the conditions of the Gnu Public Licence, ver. 2.

6. Acknowledgements

This work has been partly supported by funding from the EVERYWARE (MICyT No. TIC2003-08995-C02-01) project, to which we express our thanks.

7. Summary

A practical C source static analyser for the Linux kernel has been described, capable of dealing with the millions of lines of code in the kernel on a reasonable timescale, at a few seconds per file. The analysing logic is configured by the user to obtain different analyses (and several are performed at once).

The particular logical analysis described here has detected about two uncorrected deadlock situations per thousand files in the Linux 2.6 kernel, and about three per thousand files which access freed memory.

References

A Analytic Program Logic

The C code analyser implements a compositional (programmable) program logic called NRBG (for “normal”, “return”, “break”, “goto”, reflecting its four distinct components). The four parts, N, R, B, G, of the logic represent different kinds of control flow: a “normal”, “return”, “break”, “goto”, reflecting its four distinct components. See Figure 1 for a representation of the sequential part of the logic.

Figure 1. Normal and exceptional flow through two consecutive program fragments.

The N (“normal”) part of the logic represents the way code flows through “falling off the end” of one fragment and into another. That is, if \( p \) is the precondition that holds before program fragment \( a; b \) runs, and \( q \) is the postcondition that holds afterwards, then

\[
p N(a; b) q = p N(a) r \land r N(b) q
\]

To exit normally with \( q \), the program must flow normally through fragment \( a \), hitting an intermediate condition \( r \), and then enter fragment \( b \) and exit it normally.

On the other hand, the R part of the logic represents the way code flows out of the parts of a routine through a “return” path. Thus, if \( r \) is the intermediate condition that is attained after normal termination of \( a \), then:

\[
p R(a; b) q = p R(a) q \lor r R(b) q
\]

That is, one may either return from program fragment \( a \), or else terminate \( a \) normally, enter fragment \( b \) and return from \( b \).

The logic of break is (in this case) equal to return:

\[
p B(a; b) q = p B(a) q \lor r B(b) q
\]

where again \( r \) is the intermediate condition that is attained after normal termination of \( a \). One may either break out of \( a \), or wait for \( a \) to terminate normally, enter \( b \), and break out of \( b \).

Where break and return logic do differ is in the treatment of loops. First of all, one may only return from a forever \( \text{while} \) loop by returning from its body:

\[
p R(\text{while}(1)a) q = p R(a) q
\]

On the other hand, (counter-intuitively at first reading) there is no way of leaving a forever \( \text{while} \) loop via a break exit, because a break in the body of the loop causes a normal exit from the loop, not a break exit:

\[
p B(\text{while}(1)a) F
\]

The normal exit from a forever loop is by break from its body:

\[
p N(\text{while}(1)a) q = p B(a) q
\]

See Figure 2 for a representation of the logic.

All of these clauses follow the “once through” principle for loops. They treat loops as though they may cycle at most once. If it were required to represent the loop as cycling more than once (which it is not), one would write for the R component, for example:

\[
p R(\text{while}(1)a) q = p R(a) q \lor r R(\text{while}(1)a) q
\]

where \( r \) is the intermediate condition that is attained after normal termination of \( a \).

Typically the precondition \( p \) is the claim that the spinlock count \( \rho \) is below or equal to \( n \), for some \( n: \rho \leq n \). In that case the logical components \( i = N, R, B \) have for each precondition \( p \) a strongest postcondition \( p \text{SP}\iota(a) \) compatible with the program fragment \( a \) in question. For example, in the logic component \( N \):

\[
p N(a) q = p \text{SP}_N(a) q
\]
So each logic component can be written as a function rather than a relation by identifying it with its strongest postcondition generator.

For all atomic statements except the following particular few, the action of N is as the identity function:

\[
\begin{align*}
(p \leq n) \quad & N \left( \begin{array}{c}
\text{spin\_lock}(\& x) \\
\text{spin\_unlock}(\& x)
\end{array} \right) = \\
& \begin{cases} 
 p \leq n + 1 \\
 p \leq n - 1
\end{cases}
\end{align*}
\]

The action on a more general precondition \( p \) is to substitute \( p \) by \( p \pm 1 \) in \( p \), giving \( p[\rho - 1/\rho] \) (for \( \text{spin\_lock} \)) and \( p[\rho + 1/\rho] \) (for \( \text{spin\_unlock} \)) respectively:

\[
p N \left( \begin{array}{c}
\text{spin\_lock}(\& x) \\
\text{spin\_unlock}(\& x)
\end{array} \right) = \\
\begin{cases} 
 p[\rho - 1/\rho] \\
 p[\rho + 1/\rho]
\end{cases}
\]

The functional action on sequences of statements is then described as follows:

\[
\begin{align*}
p N(a; b) &= p N(a) N(b) \\
p R(a; b) &= p R(a) \lor p N(a) R(b) \\
p B(a; b) &= p B(a) \lor p N(a) B(b)
\end{align*}
\]

The G component of the logic is responsible for the proper treatment of \texttt{goto} statements. It ensures that loops created by \texttt{goto} statements are analysed as though they were traversed at most once. To allow this, the logic – each of the components N, R, B and G – works within the additional context of a set \( e \) of labelled \texttt{goto} conditions, each of which will take effect when the corresponding labelled statement is encountered. The G component manages this so-called \texttt{goto context} and it alone, storing the current pre-condition in the context \( e \) as a \texttt{goto} is traversed:

\[
p G_e(\texttt{goto} \ x) = \{ x : p \} \cup e
\]

If the label \( x \) is already present in the context \( e \), signifying that there has already been a \texttt{goto} \( x \) statement, then there are two possible ways to eventually reach the targeted label, so the union of the two conditions is taken and stored in the new context under label \( x \).

Augmenting the logic of sequence to take account of the context gives the following statements:

\[
\begin{align*}
p N_e(a; b) &= p N_e(a) N_{pG_e(a)}(b) \\
p R_e(a; b) &= p R_e(a) \lor p N(e) R_{pG_e(a)}(b) \\
p B_e(a; b) &= p B_e(a) \lor p N(e) B_{pG_e(a)}(b)
\end{align*}
\]

The N, R, and B semantics of a \texttt{goto} statement, which give post-conditions, are vacuous. This signifies that one cannot exit from a \texttt{goto} statement in a normal way, or on a break path, or a return path:

\[
\begin{align*}
p N_e(\texttt{goto} \ x) &= F \\
p R_e(\texttt{goto} \ x) &= F \\
p B_e(\texttt{goto} \ x) &= F
\end{align*}
\]

The effect of a \texttt{goto} is to load the context for the logic with an extra exit condition, which will be discharged to the normal component of the logic only when the label corresponding to the \texttt{goto} is encountered:

\[
\begin{align*}
p N_{(x:e)}(x : a) &= p \lor q \\
p R_e(x : a) &= F \\
p B_e(x : a) &= F \\
p G_e(x : a) &= e - \{ x : e_x \}
\end{align*}
\]

This mechanism allows the program analysis to pretend that there is an alternate entry point at the label. There is a “short-cut” from the site of the \texttt{goto} to the label, and one can get there either via the short-cut or by traversing the rest of the program.

If the label precedes the \texttt{goto} in the program, then the condition accrued from the \texttt{goto} will never be discharged at the label (or elsewhere). The result is that forward-going \texttt{goto} branches are traversed (once), and backward-going ones are traversed never. Since a backward \texttt{goto} makes a loop, this mechanism results in them being treated as though only the forward-going part of the complete loop were traversed, which is to say that the loop is traversed at most once.

To translate the syntax used in the logic specifications for the analyser as shown in Table 4 back into the logic given in Section A, consider that

\[
\text{ctx } e, \ p ::= k \in (n, r, b) \text{ with ctx } e';
\]

means

\[
\begin{align*}
p N_e(k) &= n \\
p R_e(k) &= r \\
p B_e(k) &= b \\
p G_e(k) &= e'
\end{align*}
\]

when written out in the longer format.