**One Million (LOC) and Counting**

Static Analysis for Errors and Vulnerabilities in the Linux Kernel Source Code

Peter T. Breuer and Simon Pickin

Universidad Carlos III de Madrid, Leganes, Madrid, 28911 SPAIN

ptb@inv.it.uc3m.es, spickin@it.uc3m.es

**Abstract.** This article describes an analysis tool aimed at the C code of the Linux kernel, having been first described as a prototype (in this forum) in 2004. Its continuing maturation means that it is now capable of treating millions of lines of code in a few hours on very modest platforms. It detects about two uncorrected deadlock situations per thousand C source files or million lines of source code in the Linux kernel, and three accesses to freed memory. In distinction to model-checking techniques, the tool uses a configurable “3-phase” programming logic to perform its analysis. It carries out several different analyses simultaneously.

1 Introduction

Two years ago, our group had developed a prototype static analysis tool for the Linux kernel and described it in this forum ([1]). At that time, it was a matter of some pride that the prototype could efficiently deal with some thirty thousand lines or so of source code at a time, that being about the size that a small kernel driver source code of some five hundred lines or so of C code would expand to once referenced header files had been included and all macros expanded.

Taking the development onwards to deal with first hundreds of thousands and then millions of lines of (unexpanded) source code has not been merely a question of linear improvement. The tool had to be (a) coupled with a logic compiler in order to allow the programming logic to be reconfigured for different analyses and (b) the way the tool applied the logic to a parsed program syntax tree was made configurable via a user-defined set of trigger/action rules, again compiled into the tool on demand. The coverage had to be extended again and again to deal with the many unexpected C code constructions that the GNU C compiler allows and the Linux kernel makes use of, as they were discovered, the order of complexity of the algorithms involved had to be reduced greatly to deal with more than toy cases, an efficient parse had to be created for expressions which in places reach to 5000 lexical tokens, and logical predicates needed to be normalised on the fly in order to avoid the buildup of repetitious and redundant contributions that increase the complexity of the analysis task.

We take the opportunity in this article to state with specificity the analytic logic applied to every C code construct, as refined over the past two years. The
analysis copes with the mix of C and assembler in the Linux kernel source and the tool is written wholly in C, making it easy to compile and distribute in an open source environment, and it is licensed under an open source license.

Abstract interpretation [2] plays a fundamental rôle in the analysis, causing a simplification in the description of state that is propagated through a program code; for example, there is a literal (written “\texttt{NAN}”) meaning “don’t know” in the abstract domain, and thus a program variable which may take any of the values 1, 2, or 3 may be described as having the abstract value “don’t know”, leading to a state described by one atomic proposition, not a disjunct of three.

By way of orientation, note that static analysis is in general difficult to apply to C code because of C’s pointer arithmetic and aliasing, but some notable efforts to that end have been made. David Wagner and collaborators in particular have been active in the area (see for example [5], where Linux user space and kernel space memory pointers are given different types, so that their use can be distinguished, and [6], 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 “\texttt{chroot} is always followed by \texttt{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. It is \textit{not} model-checking, but a more lightweight approach, part-way between model-checking and Jeffrey Foster’s work with CQual [3, 4], which extends the type system of C in a customisable manner. In particular, CQual has been used to detect “spinlock-under-spinlock”, a sub-case of the analysis here.

The remainder of this article is structured as follows: an example run of the analysis will be presented in Section 2, the theory of the analytic logic used will be described in Section 3, and the detail of the treatment of C will be given in Section 4, along with the definitions that customise the analysis. Then configurations of the analyser for a small variety of problems and the results are discussed in Section 5.

### 2 First Example: Sleep Under Spinlock

In [1] we focused on checking for a particular problem in SMP systems – “sleep under spinlock”. A function that can sleep (i.e., that can be scheduled out of the CPU) ought never to be called from a thread that holds a “spinlock”, the SMP locking mechanism of choice in the Linux kernel. Trying to take a locked spinlock on one CPU provokes a busy wait (“spin”) in that thread that occupies the CPU completely until the same spinlock is released on another CPU. If the thread that has locked the spinlock is scheduled out of its CPU while the lock is held, then the only thread that likely has code to release the spinlock again is not running. If by chance it is rescheduled into the CPU before any other thread tries to take the spinlock, then all is well. But if another thread tries for the spinlock first, then it will spin, occupying the CPU and keeping out the
thread that would have released the spinlock. If yet another thread tries for the spinlock, then on a 2-CPU system, the machine is dead, with both CPUs spinning waiting for a lock that will never be released. Such opportunities are denial of service vulnerabilities that any user can exploit to take down a system. 2-CPU machines are also common – any Pentium 4 of 3.2GHz or above has a dual “hyper-threading” core. Detecting sleep under spinlock is one application of the abstract logic applied by the analyser.

That analysis may now be applied at almost industrial scales – Table 1 shows the results of checking for spinlock abuse in 1055 of the 6294 C source files in the Linux 2.6.3 kernel. This particular run took about 24 hours running in a single thread on a 550MHz (dual) SMP PC with 128MB ram (nowadays the analysis runs about two to three times as fast as that). 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 for the gcc compiler 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, etc.). Five files showed up as suspicious under the analysis, as listed in Fig. 1.

Although the flagged instances are indeed calls of the kernel memory allocation function kmalloc (which may sleep) under spinlock, the arguments to the call sometimes make it safe. The function will not sleep with GFP_ATOMIC there, and that is the case in several instances, but not in the two shown in Fig. 2.

The first of these two is in code that writes microcode from user-space to the sound processor chip on the sound card; in the 2.6.11 Linux kernel source, that section of code has been replaced entirely. The second, however, is still present in 2.6.11 and 2.6.12 Linux kernels. Alan Cox owned up to it at 2.6.12.5 (Linux kernel mailing list, in thread sleep under spinlock, sequencer.c, 2.6.12.5, dated 19 Aug 2005): *Yep thats (sic) 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 vulnerability might be exercised by evoking system sounds on an SMP machine (e.g. by triggering many “you have mail” notifications at once).*

<table>
<thead>
<tr>
<th>files checked:</th>
<th>1055</th>
</tr>
</thead>
<tbody>
<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>

**Fig. 1.** Testing for sleep under spinlock in the 2.6.3 Linux kernel.
### Analytic Program Logic

The analysis works by sweeping an initial description of state (for example, “the count of spinlocks taken so far is zero”), forward through the code, constructing descriptions of all the possible states reachable at every point.

The “descriptions” are *predicates* from a very restricted set, consisting of disjunctions and conjunctions of simple propositions of the form \( x \leq a \), \( x = b \), and so on, where \( a \) and \( b \) may only be constant values (including the abstract literal meaning “don’t know”, written “NAN”). These predicates correspond to shapes that are the unions of cubes in \( n \)-dimensional integer space and we can check for inclusion of one such “cuboid” \( p \) in another \( q \) via a linear-programming algorithm, thus determining mechanically (and efficiently) whether the implication \( p \rightarrow q \) holds or not. The variables that appear in the predicates are not program but *condition* variables, introduced purely for the purpose of the analysis and manipulated by the logic configured for the individual statements of C.

At the same time as the predicate descriptions are propagated, an approximate state (different in all likelihood from the predicate constructed, but not incompatible with it) is calculated at each point and swept through the program, in order to give some guidance. It is not necessary to construct this approximation but it is helpful in detecting dead code and forced branches. It is not uncommon to write “if 0”, for example, and the abstraction would calculate the value 0 for the “0” and guide the analysis to drop the block.

The “approximate state” (currently) consists of the assignment of a range of integer values to program (not condition) variables (Fig. 3). This approximation is intended to capture all the possible values that the variable may take at that point. For example, if \( x \) is assigned the range \([-1,1]\) just prior to the statement \( x = x + 1 \); then it is assigned the range \([0,2]\) after it.

To take account of the effect of loops in the guiding approximation, changes made by the loop body to variables outside the loop are evaluated broadly. So, for example, if the loop enters with the external program variable \( x \) set to \([-1,1]\), but the body transforms it to \([0,2]\), it may be assigned the value \( \text{NAN} \) after the loop, on the basis that repeat iterations change it unpredictably. The aim is to generate a *loop invariant* abstract state. At present we do not try harder to

<table>
<thead>
<tr>
<th>File &amp; function</th>
<th>Code fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sb/sb16_csp.c:</td>
<td>619 spin_lock_irqsave(&amp;p-&gt;chip-&gt;reg_lock, flags);</td>
</tr>
<tr>
<td>snd_sb_csp_load</td>
<td>... ...</td>
</tr>
<tr>
<td></td>
<td>632 unsigned char *kbuf, *kbuf;</td>
</tr>
<tr>
<td></td>
<td>633 _kbuf = kbuf = kmalloc (size, GFP_KERNEL);</td>
</tr>
<tr>
<td>oss/sequencer.c:</td>
<td>1219 spin_lock_irqsave(&amp;lock,flags);</td>
</tr>
<tr>
<td>midi_outc</td>
<td>1220 while (n &amp;&amp; !midi_devs[dev]-&gt;outputc(dev, data)) {</td>
</tr>
<tr>
<td></td>
<td>1221 interruptible_sleep_on_timeout(&amp;seq_sleeper,HZ/25);</td>
</tr>
<tr>
<td></td>
<td>1222 n--;</td>
</tr>
<tr>
<td></td>
<td>1223 }</td>
</tr>
<tr>
<td></td>
<td>1224 spin_unlock_irqrestore(&amp;lock,flags);</td>
</tr>
</tbody>
</table>

Fig. 2. Sleep under spinlock instances in kernel 2.6.3.
find an accurate invariant state, though in principle we could try again with \([-1, 1] \cup [0, 2] = [-1, 2]\) as the putative invariant, and again for any desired number of repetitions. We currently move directly to \(\text{NAN}\) as the assigned value for any variant program variable, and stop the procedure as soon as we have an invariant approximation state. Assigning \(\text{NAN}\) for every variable always gives an invariant, so the procedure stops in at most \#variables steps.

At each point in the program code, the predicate (not the state approximation) description of the reachable states is evaluated to see if it may permit a violation of an objective that has been set. If it does, the line is flagged. Thus if we get “number of spinlocks taken may be in the range \([0, 2]\)” at a point where a function \(f\) that may sleep is called, the flag is set; it is set because the analysis says that sleepy function \(f\) may be called under spinlock there and the objective is that sleepy functions not be called under spinlock. In particular, if the uninformative statement “true” (\(T\)) were all that we had as the predicate description of state at that point, the alarm flag would be set because the state could be anything at all, and thus the objective may not be met there.

The predicate description of the reachable states at each point is propagated through the code by a compositional program logic called NRBG [1] (for “normal”, “return”, “break”, “goto”, reflecting its four principal components). The four components, N, R, B, G, represent different kinds of control flow: a “normal” flow, N, and several “exceptional” flows.

Program fragments are thought of as having three phases of execution: initial, during, and final. The initial phase is represented by a condition \(p\) that holds as the program fragment is entered. The only access to the internals of the during phase is via an exceptional exit (R, B, G; return, break, goto) from the fragment. The final phase is represented by a condition \(q\) that holds as the program fragment terminates normally (N).

The N part of the logic represents the way code “falls off the end” of one fragment and into another. That is, if \(p\) is the precondition that holds before program \(a; b\) 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 \(r\), then enter fragment \(b\), exiting it normally.

The R part of the logic represents the way code flows exceptionally out of the parts of a routine through a “return” path. 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 the case of sequence) exactly equal to that of 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 $b$ (see Fig. 4(L)).

Where break and return logic do differ is in the treatment of loops. First of all, one may only return from a forever 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 (F, “false”) of leaving a forever while loop via a break exit, because a break in the body of the loop causes a normal exit from the loop itself, not a break exit:

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

The normal exit from a forever loop is by break from its body (see Fig. 4(R)):

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

To represent the loop as cycling possibly more than once, rather than the “almost once” of the above, one would extend (4), for example, to:

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

where $r$ is the intermediate condition that is attained after normal termination of $a$. However, in practice it suffices to check that $r \rightarrow p$ holds, because then the equation reduces to the form (4) given originally. In case $r \rightarrow p$ does not hold
immediately, \( p \) is relaxed until it does. What is meant by this is that a \( p' \geq p \) is found with the property \( p' \ N(a) \ p' \). There always is such a \( p' \) since \( T \) ("true") will do. We explain further below.

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 \( N, R, B \) have for each precondition \( p \) a strongest postcondition \( p \ SP_N(a) \), \( p \ SP_R(a) \), \( p \ SP_B(a) \), compatible with the program fragment \( a \) in question. For example, in the case of the logic component \( N \):

\[
p N(a) \ q \leftrightarrow p \ SP_N(a) \leq q
\]  

Each logic component \( X \) can be written as a function rather than a relation by identifying it with a postcondition generator no stronger than \( SP_X \). For example:

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

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

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

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

\[
p N(a; b) = (p N(a)) \ N(b) \\
p R(a; b) = p R(a) \lor (p N(a)) \ R(b) \\
p B(a; b) = p B(a) \lor (p N(a)) \ B(b)
\]

Returning briefly to how we relax a predicate \( p \) to \( p' \geq p \) with \( p' \ N(a) \ p' \), we first look at \( p' = p \lor pSP_N(a) \). If this implies \( p \), we are done, since \( p \) itself is an invariant. We next check if \( p' \geq p \) is an invariant by seeing if \( p' \lor p'SP_N(a) \) implies \( p' \). If it does, we are done. If not, there is a dimension (a variable \( x \) appearing in \( p' \)) in which the lack of fit of the one cuboid in the other is manifest, because \( x = k \) for some particular \( k \) is permitted by \( p' \) but not by \( p' \lor p'SP_N(a) \). We erase atomic propositions referring to \( x \) from \( p' \), thus obtaining a \( p'' \geq p' \) (\( p' \) is a positive dis/conjunctive form in the atomic ordering propositions, so erasing part of it makes it less restricting) and then check to see if \( p'' \lor p''SP_N(a) \) is contained in \( p'' \). If it is, we are done. If not we remove references to one more variable and repeat. The procedure terminates in at most \#variables steps. At worst it gives \( T \).

The G component of the logic is responsible for the proper treatment of \texttt{goto} statements. To allow this, the logic – each of the components \( N, R, B \) and \( G \) – works within an additional context, \( e \). A context \( e \) is a set of labelled conditions, each of which are generated at a \texttt{goto} \( x \) and are discharged/will take effect at a corresponding labelled statement \( x: \ldots \). The G component manages this
context, first storing the current pre-condition \( p \) as the pair \((x, p)\) (written \( x:p \)) in the context \( e \) at the point where the \texttt{goto} \( x \) is encountered:

\[
p \ G_e(x) = \{x:p\} \uplus e
\]  

(14)

The \( \{x:p\} \) in the equation is the singleton set \( \{(x, p)\} \), where \( x \) is some label (e.g. the “\texttt{foo}” in “\texttt{foo: a = 1;}”) and \( p \) is a logical condition like “\( \rho \leq 1 \)”.

In the simplest case, the operator \( \uplus \) is set theoretic disjunction. But if an element \( x:q \) is already present in the context \( e \), signifying that there has already been one \texttt{goto} \( x \) statement encountered, then there are now two possible ways to reach the targeted label, so the union of the two conditions \( p \) and \( q \) is taken and \( x:q \) is replaced by \( x: (p \cup q) \) in \( e \).

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

\[
p \ N_e(a; b) = (p \ N_e(a)) \ N_{pG_e(a)}(b)
\]  

(15)

\[
p \ R_e(a; b) = p \ R_e(a) \ \lor \ (p \ N_e(a)) \ R_{pG_e(a)}(b)
\]  

(16)

\[
p \ B_e(a; b) = p \ B_e(a) \ \lor \ (p \ N_e(a)) \ B_{pG_e(a)}(b)
\]  

(17)

The \( N, R, B \) semantics of a \texttt{goto} statement are vacuous, signifying one cannot exit from a \texttt{goto} in a normal way, nor on a break path, nor on a return path.

\[
p \ N_e(\texttt{goto} \ x) = p \ R_e(\texttt{goto} \ x) = p \ B_e(\texttt{goto} \ x) = F
\]  

(18)

The only effect of a \texttt{goto} is to load the context for the logic with an extra exit condition. The extra condition will be discharged into the normal component of the logic only when the label corresponding to the \texttt{goto} is found (\( e_x \) is the condition labelled with \( x \) in environment \( e \), if any):

\[
p \ N_e(\{x:p\} \cup e(x)) = p \lor q \quad p \ R_e(x) = F
\]  

(19)

\[
p \ B_e(x) = F \quad p \ G_e(x) = e - \{x:e_x\}
\]  

(20)

This mechanism allows the program analysis to pretend that 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 label \texttt{foo} has already been encountered, then we have to check at \texttt{goto foo} that the current program condition is an invariant for the loop back to \texttt{foo};, or raise an alarm.

False positives are possible, but false negatives (in the sense of detectable cases that are somehow missed) are not, provided only that the code does not contain backward-going \texttt{gotos}, which we do not currently treat with full genericity (in the future that may change). This is not a claim for omniscience in the technology, just an observation that the predicate description that is calculated at each point is intentionally broad enough to encompass (1) every value that may be obtained in (2) every state that may be reached there.

That is subject to several provisos; the code must not do something odd like call an opaque subroutine that modifies its own code or data, because that is a possibility not modelled in the logic. And the analysis cannot know if \texttt{int *x=123456789; (*x)++;} modifies a memory location that is significant
other than as data; perhaps it is the stack return address. The logic detailed in the next section ignores possible accesses other than by name to the data in variables, and indeed, as configured, takes no note of what value is stored.

4 The Analyser

The static analyser allows the program logic of C to be configured in detail by the user. 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 specifications of the program logic was written and incorporated into the analysis tool. The logic compiler understands specifications of the format

\[ \text{ctx pre-context, precondition :: name( arguments) = postconditions with ctx post-context; } \]

where the \textit{precondition} is an input argument, the entry condition for a code fragment, and \textit{postconditions} is an output, a tuple consisting of the N, R, B exit conditions according to the logic. The \textit{pre-context} is the prevailing \textit{goto} context. The \textit{post-context} is the output \textit{goto} context, consisting of a set of labelled conditions. For example, the specification of the empty statement logic is:

\[ \text{ctx e, p::empty()} = (p, F, F) \text{ with ctx e; } \]

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 analysis propagates a specified initial condition forward through the program, developing postconditions after each program statement that are checked for conformity with a specified objective. The full set of logic specifications is given in Table 1. To relate it to the logic presentation in Section 3, keep in mind:

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

means

\[ p \ N_e(k) = n \quad p \ R_e(k) = r \quad p \ B_e(k) = b \quad p \ G_e(k) = e' \]

written out in the mathematical notation of Section 3.

The treatment of \textit{spin_unlock} calls, \textit{write_unlock} calls, \textit{read_unlock} calls, etc. in Linux kernel code is managed by the \textit{unlock} entry in the table. These all decrement the spinlock counter \( n \). The argument \texttt{label 1} to the call is an identifier for the spinlock address that appears as an argument to the call. Similarly the \textit{lock} entry in the table represents the logic of the \textit{spin_lock}, \textit{write_lock}, \textit{read_lock}, etc. calls. These calls all increment the spinlock counter \( n \).

Note that function calls act like spinlock no-ops. That is, other functions are assumed to be balanced with respect to their effect on spinlocks. That is a good heuristic, because the only function that is explicitly unbalanced in that respect in the Linux kernel is the call \textit{spin_trylock}, which takes the spinlock if it is free and returns 0, or else cannot take it and returns 1. And if any (other) function were unbalanced it would be noticed during the analysis of that function.
Table 1. The single precondition/triple postcondition program logic of C.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Logic Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx e, p::for(stmt)</td>
<td>(n ∨ b, r, F) with ctx f</td>
</tr>
<tr>
<td>ctx e, p::empty()</td>
<td>(p, F, F) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::unlock(label l)</td>
<td>(p[n+1/n], F, F) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::lock(label l)</td>
<td>(p[n-1/n], F, F) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::assembler()</td>
<td>(p, F, F) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::function()</td>
<td>(p, F, F) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::sleep(label l)</td>
<td>(p, F, F) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::sequence(s₁, s₂)</td>
<td>(n₂, r₁ ∨ r₂, b₁ ∨ b₂) with ctx g;</td>
</tr>
<tr>
<td>ctx e, p::switch(stmt)</td>
<td>(n₁ ∨ b₁, r₁ ∨ r₂, b₁ ∨ b₂) with ctx f₁</td>
</tr>
<tr>
<td>ctx e, p::if(s₁, s₂)</td>
<td>(n₁ ∨ b₁, r₁ ∨ r₂, b₁ ∨ b₂) with ctx f₁</td>
</tr>
<tr>
<td>ctx e, p::while(stmt)</td>
<td>(n₁ ∨ b₁, r₁ ∨ r₂, b₁ ∨ b₂) with ctx f₁</td>
</tr>
<tr>
<td>ctx e, p::do(stmt)</td>
<td>(n₁ ∨ b₁, r₁ ∨ r₂, b₁ ∨ b₂) with ctx f₁</td>
</tr>
<tr>
<td>ctx e, p::goto(label l)</td>
<td>(F, F, F) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::continue()</td>
<td>(F, F, p) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::break()</td>
<td>(F, F, p) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::return()</td>
<td>(F, F, p) with ctx e;</td>
</tr>
<tr>
<td>ctx e, p::labeled(label l)</td>
<td>(p ∨ e.l, F, F) with ctx e</td>
</tr>
</tbody>
</table>

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;
- labelled - labelled statements.

An “objective” function for the analysis is specified by an objective specification in the same format as the logic specifications (see Table 1). The term upper[n:p] gives the estimated upper limit of the counter n subject to the constraints in the precondition p. The limit is +∞ if p is “true” (T). The predicate must contain information that bounds n away from positive values if the objective is not to generate a positive value, and less information in the predicate will cause a more positive value to be calculated as the spinlock count upper bound.

The objective is computed at each node of the syntax tree. Positive values of the objective function are reported to the user (with the trigger/action rules that are currently in force and which will be described in the following part of this section). In particular, calls to functions which can sleep at a node where the objective function is positive are reported (this indicates where a call to a sleepy function might occur under spinlock).
Table 2. Trigger/action rules which propagate information through the syntax tree.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$\text{SLEEP!} \ &amp; \ \text{OBJECTIVE_SET} \ &amp; \ \text{OBJECTIVE} \geq 0$</td>
<td>$\text{aliases} \models \text{SLEEP}, \ \text{callers} \models \text{SLEEP}$</td>
</tr>
</tbody>
</table>
| 2.   | $\text{REF!} \ & \ \text{SLEEP}$ | $\text{callers} \models \text{SLEEP}, \ 
eg \text{REF}$ |
| 3.   | $(\text{SLEEP} \ & \ \text{OBJECTIVE\_SET} \ & \ \text{OBJECTIVE} \geq 0)!$ | $\rightarrow \text{output()}$ |

The initial specification ($n \leq 0$) shown in Table 3 describes the initial program state at runtime. It says here that the spinlock counter $n$ is less than or equal to zero (actually, zero, but the inequality is just as good and simpler).

The analysis also assumes that the tested value in conditionals, case statements and loops contains no significant program code (break, continue, etc.) – GNU C allows it but it does not appear in practice in the Linux kernel source.

The logic propagation through the syntax tree is complemented by a trigger/action system which acts whenever a property changes on a node. As the analysis tool is currently configured, the rules in Table 2 are applied. Their principal aim is to construct the list of sleepy functions, checking for calls by name of already known sleepy functions and thus constructing the transitive closure of the list under the call (by name) graph.

Rule (1) applies whenever a function is newly marked as sleepy ($\text{SLEEP!}$). Then if the objective function (here the maximal value of the spinlock count $n$) has already been calculated on that node ($\text{OBJECTIVE\_SET}$) and is not negative ($\text{OBJECTIVE} \geq 0$, indicating that the spinlock count is 0 or higher) then all the known aliases (other syntactic nodes which refer to the same semantic entity) are also marked sleepy, as are all the known callers (by name) of this node (which will be the current surrounding function, plus all callers of aliases of this node).

The reason why sleepiness is not propagated under negative spinlock is quite subtle. Consider function $f$ called from function $g$ called from function $h$. If the spinlock count is negative at the call of $f$ in $g$, then $g$ is intended to be called under spinlock (releasing an already released spinlock is a design error). If $f$ is sleepy then $g$ would ordinarily be marked sleepy too and that would be marked as an error when $g$ is called under spinlock in $h$. But that is wrong when $f$ is under negative spinlock in $g$, because then $f$ is not under spinlock when $g$ is called under spinlock in $h$ and it is not a problem in $h$ that $f$ chooses to sleep inside $g$. So, under these conditions, $g$ should not be marked as sleepy.

Rule (2) in Table 2 is triggered when a known sleepy function is referenced ($\text{REF!}$). 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 effect of rules (1) and (2) together is to efficiently create the transitive sleepy call (by name) graph.

A list of all calls to functions that may sleep under a positive spinlock count is created via rule (3) in Table 2. Entries are added 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).
Table 3. Defining initial conditions, and an objective function to be calculated at every node of the syntax tree.

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

The analyser is called with the same arguments as the gcc compiler would have used. That enables the kernel to be compiled once, the calls to gcc recorded, and then the analyser to be run using the same arguments as were used for gcc.

The parser handles both the code of the 2.4 series Linux kernel and the 2.6 series. The lexer is user-configurable and needs seeding with the names of those functions which are known a priori to sleep, and the names of the spinlock lock and unlock calls. Less than twenty seed functions have been used.

5 More Targets

Spinlock-under-spinlock 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 that graph requires attaching the code

```
setflags(SPINLOCK)
```

into the logic of the spin lock function calls in Table 1, just as for the sleep function calls. The trigger/action rules in Table 2 are then duplicated, substituting SPINLOCK for SLEEP in the existing rules, so that the rules propagate the SPINLOCK flag as well as the SLEEP flag from callee to caller. Then a single trigger/action rule is added which outputs an alert when a function marked with SPINLOCK (i.e. a function which calls a function which ... takes a spinlock) is called under spinlock:

```
(SPINLOCK & SPIN_SET & SPIN > 0)! -> output()
```

Why is taking a spinlock twice dangerous? Taking the same spinlock twice is deadly, as Linux kernel spinlocks are not reentrant. The result will be to send the CPU into a busy forever loop. Taking two different spinlocks one under the other in the same thread is not dangerous, unless another thread takes the same two spinlocks, one under the other, in the reverse order. There is a short window where both threads can take one spinlock and then busy-wait for the other thread to release the spinlock they have not yet taken, thus spinning both CPUs simultaneously and blocking further process. In general, there is a deadlock window like this if there exists any spinlock cycle such that A is taken under B, B is taken under C, etc. Detecting double-takes flags the potential danger.

We have also been able to detect accesses to freed memory (including frees of freed memory). The technique consists of setting the logic of a kfree call on a variable containing a memory address to increment a counter variable a(1)
Fig. 5. Testing for access to kfreed memory in the 2.6.3 Linux kernel.

<table>
<thead>
<tr>
<th>File &amp; function</th>
<th>Code fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>fm801-gp.c:</td>
<td></td>
</tr>
<tr>
<td>fm801 gp_probe</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td><code>kfree(gp);</code></td>
</tr>
</tbody>
</table>
| 102              | `printk("unable to grab region 0x%x-0x%x\n",
           gp->gameport.io, gp->gameport.io + 0x0f);` |
| aic7xxx_old.c:  |               |
| aic7xxx_detect  |               |
| 9240            | `while(current_p && temp_p)` |
| 9241            | `{`         |
| 9242            | `if (((current_p->pci_bus==temp_p->pci_bus) && ...){` |
| 9243            | `...`       |
| 9248            | `kfree(temp_p);` |
| 9249            | `continue;`  |

Fig. 6. Access to kfreed memory in kernel 2.6.3.

unique to the (integer index label \(l\) generated by the analysis for the) variable. Assigning the variable again resets the counter to zero \(p!a(l)\) means proposition \(p\) relaxed to remove references to the counter \(a(l)\); \(a\) is treated like a vector where appropriate, so initial condition \(a \leq 0\) has \(a(l) \leq 0\) too:

\[
\text{ctx } e, p :: \text{kfree(label } l \text{)} = (p[a(l)-1/a(l)], F, F) \text{ with ctx } e; \\
\text{ctx } e, p :: \text{assignment(label } l \text{)} = (p!a(l)], F, F) \text{ with ctx } e;
\]

The alarm is sounded when the symbol with label \(l\) is accessed where the counter \(a(l)\) may take a positive value – variable with index \(l\) may point to freed memory.

A survey of 1151 C source files in the Linux 2.6.3 kernel reported 426 “alarms” but most of these were clusters with a single origin. Exactly 30 of the 1151 files were reported as suspicious in total (see Table 5). One of these (aic7xxx_old.c) generated 209 of the alarms, another (aic7xxx_proc.c) 80, another (cpqphp_ctrl.c) 54, another 23, another 10, then 8, 7, 5, 4, 2, 2, 2, and the rest 1 alarm each. Three (3) of the flagged files contained real errors of the type searched for. Two of the error regions are shown in Fig. 6. Curiously, drivers/scsi/aic7xxx_old.c is flagged correctly, as can be seen in the second code segment in the figure.

All the false alarms were due to a bug in the postcondition logic of assignment at the time of the experiment, which caused a new assignment to \(x\) closely following on the heels of a `kfree(x)` to be (erroneously) flagged.

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 errors were detected. There were 8 false alarms given on 7 files (all due to a parser bug at the time which led to a field dereference being treated like reference to a variable of the same name).
6 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 (GPL), version 2.

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 time-scale, at a few seconds per file. The analysing logic is configured 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 already freed memory.

8 Acknowledgements

This work has been partly supported by funding from the EVERYWARE (MCyT No. TIC2003-08995-C02-01) project, to which we express our thanks. We are also grateful to the shepherding member of the program committee for his helpful guidance in the final preparation.

References