Checking for Deadlock, Double-Free and Other Abuses 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. The analysis described in this article detects about two real and uncorrected deadlock situations per thousand C source files or million lines of code in the Linux kernel source, and three accesses to freed memory, at a few seconds per file. In distinction to model-checking techniques, the analysis applies a configurable "3-phase" Hoare-style logic to an abstract interpretation of C code to obtain its results.

1 Introduction

The pairing of formal methods practitioners and the Linux kernel has sometimes seemed more than unlikely. On the one hand kernel contributors have preferred to write elegant code rather than elegant specifications; "the code is the commentary" has been one of the mantras of the Linux kernel development, meaning that the code should be so clear in its own right that it serves as its own specification. That puts what is usually the first question of formal methods practitioners, "what should the code do" out of bounds. And on the other hand, formal methods practitioners have not been able to find a way into the six million lines of ever-changing C code that comprises the Linux kernel source.

This article describes the application of *post-hoc* formal logical analysis to the Linux kernel. The technology detects real coding errors that have been missed by thousands of eyes over the years. The analyser itself is written in C (thus making it easy to compile and distribute in an open source environment) and is itself licensed under an open source license. The analysis is configurable, which means that it is possible to re-program and extend it without rewriting its source.

By way of orientation, note that static analysis is in general difficult to apply to C code, because of 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 [7], where Linux user space and kernel space memory pointers are given different types, so that their use can be distinguished, and [8], 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 "`chroot` is always followed by `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 CQual [5, 6], which extends the type system of C in a customisable manner. In particular, CQual has been used to detect "spinlock under spinlock", a variant of the analysis described here. The technology described in this article was first described in prototype in [3], being an application of the generic "three-phase" program logic first described in [1] and developed in [2]. The tool itself now works at industrial scales, treating millions of lines of code in a few hours when run on a very modest PC.

In the analysis here, *abstract interpretation* [4] 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 abstract literal, 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.

To be exact, the analysis constructs two abstract approximations, a state s and a predicate p describing the real state x such that

$$x \in s \cap p \tag{1}$$

The approximated state s assigns a range of integer values to each variable.

Predicates are restricted to the class of disjunctions of conjuncts of simple ordering relations $\mathtt{x} \leq \mathtt{k}$, and there is a simple decision procedure for implication.

In [3] we focussed 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") that occupies the CPU completely until the spinlock is released on another CPU. If the thread that has locked the spinlock is scheduled out while the lock is held, then the only thread that likely has code to release the spinlock is not running. If by chance that thread is rescheduled in to the CPU before any other thread tries for the spinlock then all is well. But if another thread tries for the spinlock first, it will spin uselessly, keeping out of that CPU the thread that would have released the spinlock. If yet another thread tries for the spinlock, then on a 2-CPU SMP system, the machine is dead, with both CPUs spinning waiting for a lock that will never be released. Such vulnerabilities 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. So, calling a function that may sleep while holding the lock on a spinlock is a serious matter. Detecting it is one application of the abstract logic that may be applied by the analyser.

2 Example 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

files checked:	1055	
alarms raised:	18	(5/1055 files)
false positives:	16/18	
real errors:	2/18	(2/1055 files)
time taken:	~24h	
LOC:	~700K	(unexpanded)

```
1 instances of sleep under spinlock
        in sound/isa/sb/sb16_csp.c
1 instances of sleep under spinlock
        in sound/oss/sequencer.c
6 instances of sleep under spinlock
        in net/bluetooth/rfcomm/tty.c
7 instances of sleep under spinlock
        in net/irda/irlmp.c
3 instances of sleep under spinlock
        in net/irda/irttp.c
```

Fig. 1. Testing for sleep under spinlock in the 2.6.3 Linux kernel

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

Fig. 2. Sleep under spinlock instances in kernel 2.6.3

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 out of that selection showed up as suspicious under the analysis, as listed in Fig. 1.

Although the flagged constructs are indeed calls of the kernel memory allocation function `kmalloc` (which may sleep) under spinlock, the arguments to the call sometimes render it harmless, i.e. cause it not to sleep after all. The `kmalloc` function will not sleep with `GFP_ATOMIC` as second argument, and such is the case in several instances, but not in the two instances shown in Fig. 2.

3 Analytic Program Logic

The C code analyser is based on a compositional program logic called NRBG (for "normal", "return", "break", "goto", reflecting its four principal components). The four components, N, R, B, G, represent different kinds of control flows: a "normal" flow 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 control flow "falls off the end" of one fragment and into another. I.e., if p is the condition that holds before program $a;b$ runs, and q is the condition that holds after, then

$$p\ N(a;b)\ q \quad = \quad p\ N(a)\ r\ \wedge\ r\ N(b)\ q \tag{2}$$

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

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:

$$p\ R(a;b)\ q \quad = \quad p\ R(a)\ q\ \vee\ r\ R(b)\ q \tag{3}$$

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) equal to that of return:

$$p\ B(a;b)\ q \quad = \quad p\ B(a)\ q\ \vee\ r\ B(b)\ q \tag{4}$$

where again r is the condition attained after normal termination of a.

Where break and return logic differ is in the treatment of loops. First of all, one may only return from a forever `while` loop by returning from its body:

$$p\ R(\texttt{while}(1)\,a)\ q \quad = \quad p\ R(a)\ q \tag{5}$$

On the other hand, (counter-intuitively at first reading) there is no way 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:

$$p\ B(\texttt{while}(1)\,a)\ F \tag{6}$$

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

$$p\ N(\texttt{while}(1)\,a)\ q \quad = \quad p\ B(a)\ q \tag{7}$$

To represent the loop as cycling possibly more than once, one would write for the R component, for example:

$$p\ R(\texttt{while}(1)\,a)\ q \quad = \quad p\ R(a)\ q\ \vee\ r\ R(\texttt{while}(1)a)\ q \tag{8}$$

where r is the intermediate condition that is attained after normal termination of a. However, in practice it suffices to check that $r \to p$ holds, because then (8) reduces to (5). If $r \to p$ does not hold, p is *relaxed* to $p' \geq p$ for which it does.

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

$$p\ N(a)\ q \quad \leftrightarrow \quad p\,\mathrm{SP}_N(a) \leq q \tag{9}$$

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:

$$(\rho \leq n)\ N \begin{pmatrix} \texttt{spin\_lock}(\&x) \\ \texttt{spin\_unlock}(\&x) \end{pmatrix} \quad = \quad \begin{pmatrix} \rho \leq n+1 \\ \rho \leq n-1 \end{pmatrix} \tag{10}$$

Or in the general case, the action on precondition p is to substitute ρ by $\rho \pm 1$ in p, giving $p[\rho - 1/\rho]$ (for `spin_lock`) and $p[\rho + 1/\rho]$ (for `spin_unlock`) respectively:

$$p\ N \begin{pmatrix} \texttt{spin\_lock}(\&x) \\ \texttt{spin\_unlock}(\&x) \end{pmatrix} \quad = \quad \begin{pmatrix} p[\rho - 1/\rho] \\ p[\rho + 1/\rho] \end{pmatrix} \tag{11}$$

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

$$p\ N(a;b) = (p\ N(a))\ N(b) \tag{12}$$

$$p\ R(a;b) = p\ R(a)\ \vee\ (p\ N(a))\ R(b) \tag{13}$$

$$p\ B(a;b) = p\ B(a)\ \vee\ (p\ N(a))\ B(b) \tag{14}$$

The G component of the logic is responsible for the proper treatment of `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 `goto` x and are discharged/will take effect at a corresponding labelled statement x`: ...`. 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 `goto` x is encountered:

$$p\ G_e(\texttt{goto}\ x) = \{x{:}p\} \uplus e \tag{15}$$

The $\{x{:}p\}$ in the equation is the singleton set $\{(x, p)\}$, where x is some label (e.g. the "`foo`" in "`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 `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 (12-14) to take account of context gives:

$$p\ N_e(a;b) = (p\ N_e(a))\ N_{pG_e(a)}(b) \tag{16}$$

$$p\ R_e(a;b) = p\ R_e(a)\ \vee\ (p\ N_e(a))\ R_{pG_e(a)}(b) \tag{17}$$

$$p\ B_e(a;b) = p\ B_e(a)\ \vee\ (p\ N_e(a))\ B_{pG_e(a)}(b) \tag{18}$$

The N, R, B semantics of a `goto` statement are vacuous, signifying one cannot exit from a `goto` in a normal way, nor on a break path, nor on a return path.

$$p\ N_e(\texttt{goto}\ x) = p\ R_e(\texttt{goto}\ x) = p\ B_e(\texttt{goto}\ x) = F \tag{19}$$

The only significant effect of a `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 `goto` is found (e_x is the condition labeled with x in environment e, if any):

$$\begin{array}{llll} p\ N_{\{x:q\}\cup e}(x{:}) & = p \vee q & \quad p\ R_e(x{:}) = F \\ p\ B_e(x{:}) & = F & \quad p\ G_e(x{:}) = e - \{x{:}e_x\} \end{array} \tag{20}$$

This mechanism allows the program analysis to pretend that there is a "short-cut" from the site of the `goto` to the label, and one can get there either via the short-cut or by traversing the rest of the program. If label `foo` has already been encountered, then we have to check at `goto foo` that the current program condition is an invariant for the loop back to `foo:`, or raise an alarm.

The equations given can be refined by introducing temporal logic (CTL). Consider the return logic of sequence, for example. If $\mathbf{EF}p$ is the statement that there is at least one trace leading to condition p at the current flow point, then:

$$\frac{pR(a)\mathbf{EF}r_1 \qquad pN(a)\mathbf{EF}q \qquad qR(b)\mathbf{EF}r_2}{pR(a;b)(\mathbf{EF}r_1 \wedge \mathbf{EF}r_2)} \tag{21}$$

The deduction that $\mathbf{EF}r_1 \wedge \mathbf{EF}r_2$ holds is stronger than $r_1 \vee r_2$, which is what would be deduced in the absence of CTL.

The above is a *may* semantics, because it expresses the possible existence of a trace. A *must* semantics can be phrased via the the operator $\mathbf{AF}p$, which expresses that all traces leading to the current point give rise to condition p here:

$$\frac{pR(a)\mathbf{AF}r_1 \qquad pN(a)\mathbf{AF}q \qquad qR(b)\mathbf{AF}r_2}{pR(a;b)(\mathbf{AF}(r_1 \vee r_2))} \tag{22}$$

In general, the deduction systems prove $pX\,\mathbf{AF}q_1$, pXq_2, $pX\mathbf{EF}q_3$ with $q_1 \leq q_2$ and $q_2 \leq q_3$, which brackets the analysis results between forced and possible.

4 Configuring the Analysis

The static analyser allows the program logic set out in the last section to be specified 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. Instead, 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

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

Table 1. Defining the single precondition/triple postcondition NRB logic of C

ctx e, p::for(stmt)	= (n$\vee$b, 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(s_1, s_2)	= (n_2, $r_1 \vee r_2$, $b_1 \vee b_2$) with ctx g
	where ctx f, n_1::s_2 = (n_2,r_2,b_2) with ctx g
	and ctx e, p::s_1 = (n_1,r_1,b_1) with ctx f;
ctx e, p::switch(stmt)	= (n$\vee$b, r, F) with ctx f
	where ctx e, p::stmt = (n,r,b) with ctx f
ctx e, p::if(s_1, s_2)	= ($n_1 \vee n_2$, $r_1 \vee r_2$, $b_1 \vee b_2$) with ctx $f_1 \vee f_2$
	where ctx e, p::s_1 = (n_1,r_1,b_1) with ctx f_1
	and ctx e, p::s_2 = (n_2,r_2,b_2) with ctx f_2;
ctx e, p::while(stmt)	= (n$\vee$b, r, F) with ctx f
	where ctx e, p::stmt = (n,r,b) with ctx f;
ctx e, p::do(stmt)	= (n$\vee$b, 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$\vee${l::p};
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, p, F) with ctx e;
ctx e, p::labeled(label l)	= (p$\vee$e.l, F, F) with ctx e\\l;

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

where the *precondition* is an input, the entry condition for a code fragment, and *postconditions* is an output, a tuple consisting of the N, R, B exit conditions according to the logic. The *precontext* is the prevailing `goto` context. The *postcontext* is the output `goto` context, consisting of a set of labelled conditions.

For example, the specification of the empty statement logic is:

```
ctx 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 full set of logic specifications is given in Table 1. To translate back into the logic presentation in Section 3, consider that

$$\texttt{ctx}\ e,\ p\ ::\ k = (n, r, b)\ \texttt{with ctx}\ e';$$

means

$$\begin{array}{ll} p\ N_e(k) = n & p\ R_e(k) = r \\ p\ B_e(k) = b & p\ G_e(k) = e' \end{array}$$

when written out in the longer format.

5 Software

The source code of the software described here 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, version 2.

6 Summary

A C source static analyser for the Linux kernel has been created, capable of dealing with the millions of lines of code in the kernel on a reasonable timescale, at a few seconds per file. It is based on a "three-phase" logic of imperative programming, as described in this article.

References

1. P.T. Breuer, N. Martínez Madrid, L. Sánchez, A. Marín, C. Delgado Kloos: A formal method for specification and refinement of real-time systems. In *Proc. 8'th EuroMicro Workshop on Real Time Systems*, pages 34–42. IEEE Press, July 1996. L'aquilla, Italy.
2. P.T. Breuer, C. Delgado Kloos, N. Martínez Madrid, A. López Marin, L. Sánchez: A Refinement Calculus for the Synthesis of Verified Digital or Analog Hardware Descriptions in VHDL. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19(4):586–616, July 1997
3. Peter T. Breuer, Marisol Garciá Valls: Static Deadlock Detection in the Linux Kernel, pages 52-64 In *Reliable Software Technologies - Ada-Europe 2004, 9th Ada-Europe International Conference on Reliable Software Technologies, Palma de Mallorca, Spain, June 14-18, 2004*, Eds. Albert Llamosí and Alfred Strohmeier, ISBN 3-540-22011-9, Springer LNCS 3063, 2004.
4. P. Cousot, R. Cousot: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. on the Principles of Programming Languages*, pages 238–252, 1977.
5. Jeffrey S. Foster, Manuel Fähndrich, Alexander Aiken: A Theory of Type Qualifiers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99).* Atlanta, Georgia. May 1999.
6. Jeffrey S. Foster, Tachio Terauchi, Alex Aiken: Flow-Sensitive Type Qualifiers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 1-12. Berlin, Germany. June 2002.
7. Rob Johnson, David Wagner: Finding User/Kernel Pointer Bugs With Type Inference. In *Proc. 13th USENIX Security Symposium, 2004* August 9-13, 2004, San Diego, CA, USA.
8. David Wagner, Jeffrey S. Foster, Eric A. Brewer, Alexander Aiken: A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. Network and Distributed System Security (NDSS) Symposium 2000*, February 2-4 2000, San Diego, CA, USA.