Using Redundancies to Find Errors

Yichen Xie and Dawson Engler

Abstract—Programmers generally attempt to perform useful work. If they performed an action, it was because they believed it served some purpose. Redundant operations violate this belief. However, in the past, redundant operations have been typically regarded as minor cosmetic problems rather than serious errors. This paper demonstrates that, in fact, many redundancies are as serious as traditional hard errors (such as race conditions or null pointer dereferences). We experimentally test this idea by writing and applying five redundancy checkers to a number of large open source projects, finding many errors. We then show that, even when redundancies are harmless, they strongly correlate with the presence of traditional hard errors. Finally, we show how flagging redundant operations gives a way to detect mistakes and omissions in specifications. For example, a locking specification that binds shared variables to their protecting locks can use redundancies to detect missing bindings by flagging critical sections that include no shared state.

Index Terms—Extensible compilation, error detection, program redundancy, software quality.

1 Introduction

Programming languages have long used the fact that many high-level conceptual errors map to low-level type errors. This paper demonstrates the same mapping in a different direction: Many high-level conceptual errors also map to low-level redundant operations. With the exception of a few stylized cases, programmers are generally attempting to perform useful work. If they performed an action, it was because they believed it served some purpose. Spurious operations violate this belief and are likely errors. However, in the past, redundant operations have been typically regarded as merely cosmetic problems rather than serious mistakes. Evidence for this perception is that, to the best of our knowledge, the many recent error checking projects focus solely on hard errors such as null pointer dereferences or failed lock releases, rather than redundancy checking [2], [4], [5], [9], [13], [23], [25].

We experimentally demonstrate that, in fact, many redundancies signal mistakes as serious as traditional hard errors. For example, impossible Boolean conditions can signal mistaken expressions, critical sections without shared states can signal the use of the wrong variable, and variables written but not read can signal an unintentionally lost result. Even when harmless, these redundancies signal conceptual confusion, which we would also expect to correlate with hard errors, such as deadlocks, null pointer dereferences, etc.

In this paper, we use redundancies to find errors in three ways: 1) by writing checkers that automatically flag redundancies, 2) using these errors to predict nonredundant errors (such as null pointer dereferences), and 3) using redundancies to find incomplete program specifications. We discuss each below.

We wrote five checkers that flagged potentially dangerous redundancies:

1. idempotent operations,
2. assignments that were never read,
3. dead code,
4. conditional branches that were never taken, and
5. redundant NULL-checks.

The errors found would largely be missed by traditional type systems and checkers. For example, as Section 2 shows, assignments of variables to themselves can signal mistakes, yet such assignments will type check in any common programming language we know of.

Of course, some legitimate actions cause redundancies. Defensive programming may introduce "unnecessary" operations for robustness; debugging code, such as assertions, can check for "impossible" conditions; and abstraction boundaries may force duplicate calculations. Thus, to effectively find errors, our checkers must separate such redundancies from those induced by high-level confusion.

The technology behind the checkers is not new. Optimizing compilers use redundancy detection and elimination algorithms extensively to improve code performance. One contribution of our work is the realization that these analyses have been silently finding errors since their invention.

We wrote our redundancy checkers in the *MC* extensible compiler system [15], which makes it easy to build system-specific static analyses. Our analyses do not depend on an extensible compiler, but it does make it easier to prototype and perform focused suppression of false positive classes.

We evaluated how effective flagging redundant operations is at finding dangerous errors by applying the above five checkers to three open source software projects: Linux, OpenBSD, and PostgreSQL. These are large, widely used source code bases (we check 3.3 million lines of them) that serve as a known experimental base. Because they have been written by many people, they are representative of many different coding styles and abilities.

We expect that redundancies, even when harmless, strongly correlate with hard errors. Our relatively uncontroversial hypothesis is that confused or incompetent

• *The authors are with the Computer Systems Laboratory, 353 Serra Mall, Stanford University, Stanford, CA 94305.*
E-mail: yxie@cs.stanford.edu, engler@csl.stanford.edu.

Manuscript received 4 Mar. 2003; revised 9 June 2003; accepted 21 June 2003.
Recommended for acceptance by W. Griswold.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 118735.

programmers tend to make mistakes. We experimentally test this hypothesis by taking a large database of hard Linux errors that we found in prior work [8] and measure how well redundancies predict these errors. In our experiments, files that contain redundancies are roughly 45 to 100 percent more likely to have traditional hard errors compared to those drawn by chance. This difference holds across the different types of redundancies.

Finally, we discuss how traditional checking approaches based on annotations or specifications can use redundancy checks as a safety net to find missing annotations or incomplete specifications. Such specification mistakes commonly map to redundant operations. For example, assume we have a specification that binds shared variables to locks. A missed binding will likely lead to redundancies: a critical section with no shared state and locks that protect no variables. We can flag such omissions because we know that every lock should protect some shared variable and that every critical section should contain some shared state.

This paper makes four contributions:

1. The idea that redundant operations, like type errors, commonly flag serious correctness errors.
2. Experimentally validating this idea by writing and applying five redundancy checkers to real code. The errors found often surprised us.
3. Demonstrating that redundancies, even when harmless, strongly correlate with the presence of traditional hard errors.
4. Showing how redundancies provide a way to detect dangerous specification omissions.

The main caveat with our approach is that the errors we count might not be errors since we were examining code we did not write. To counter this, we only diagnosed ones that we were reasonably sure about. We have had close to three years of experience with Linux bugs, so we have reasonable confidence that our false positive rate of bugs that we diagnose, while nonzero, is probably less than 5 percent.

In addition, some of the errors we diagnose are not traditional "hard errors"—they, by themselves, would probably not cause system crashes or security breaches. Rather, they are nonsensical redundancies that, in our opinion, result in unnecessary complexity and confusion. So, the diagnosis of these errors involve personal judgments that may not be shared by all readers. Although they are not as serious as hard errors, we think they should nevertheless be fixed in order to improve program clarity and readability.

Sections 2 through 6 present the five checkers. Section 7 measures how well these redundant errors correlate with and predict traditional hard errors. Section 8 discusses how to check for completeness using redundancies. Section 9 discusses related work. Finally, Section 10 concludes.

2 Idempotent Operations

The checker in this section flags idempotent operations where a variable is:

1. assigned to itself (x = x),
2. divided by itself (x / x),
3. bitwise or'd with itself (x | x), or
4. bitwise and'd with itself (x & x).

TABLE 1
Bugs Found by the Idempotent Checker in Linux 2.4.5-ac8, OpenBSD 3.2, and PostgreSQL 7.2

System	Bugs	Minor	False
Linux 2.4.5-ac8	7	6	3
OpenBSD 3.2	2	6	8
PostgreSQL	0	0	0

The checker is the simplest in the paper (it requires about 10 lines of code in our system). Even so, it found several interesting cases where redundancies signal high-level errors (see Table 1). Four of these were apparent typos in variable assignments. The clearest example was the following Linux code, where the programmer makes a mistake while copying structure sa to da:

```
/*_linux2.4.1/net/appletalk/aarp.c:aarp_rcv_*/
else { /* We need to make a copy of the entry.*/
    da.s_node = sa.s_node;
    da.s_net= da.s_net;
```

This is a good example of how redundancies catch cases that type systems miss. This code—an assignment of an integer variable to itself—will type check in all common languages we know of, yet clearly contains an error. Two of the other errors in Linux were caused by integer overflow (or'ing an 8-bit variable by a constant that only had bits set in the upper 16 bits) which was optimized away by the gcc front end. The final one in Linux was caused by an apparently missing conversion routine. The code seemed to have been tested only on a machine where the conversion was unnecessary, which prevented the tester from noticing the missing routine.

The two errors we found in OpenBSD are violations of the ANSI C standard. They both lie in the same function in the same source file. We show one of them below:

```
1 /* openbsd3.2/sys/kern/subr_userconf.c:userconf_add */
2 for (i= 0; i< pv_size; i++) {
3    if (pv[i] != -1 && pv[i] >= val)
4          pv[i] = pv[i]++; /* error */
5 }
```

The error occurs at line 4 and is detected with the help of the code canonicalization algorithm in the *xgcc* front end that translates this statement into:

```
pv[i] = pv[i]; /* redundant */
pv[i]++;
```

The ANSI C standard (Section 6.3) stipulates that "between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression." It is mere coincidence that gcc chooses to implement the side-effects of pv[i]++ after that of the assignment itself. In fact, the Compaq C compiler[1] evaluates pv[i]++ first and stores the *old* value back to pv[i], causing the piece of code to fail in a nonobvious way. We tested four

1. Compaq C V6.3-129 (dtk) on Compaq Tru64 UNIX V5.0A (Rev. 1094).

C compilers by different vendors on different architectures.[2] None of them issued a warning on this illegal statement.

The minor errors were operations that seemed to follow a nonsensical but consistent coding pattern, such as adding 0 to a variable for typographical symmetry with other nonzero additions such as the following:

```
/* linux2.4.5-ac8/drivers/video/riva/riva_hw.c:
                                        nv4CalcArbitration */
nvclks += 1;
nvclks += 1;
nvclks += 1;
if (mp_enable)
     mclks +=4;
nvclks += 0; /* suspicious, is it a typo or should it
                  really be"+=1"? */
```

Curiously, each of the 11 false positives we found was annotated with a comment explaining why the redundant operation was being done. This gives evidence for our belief that programmers regard redundancy as somewhat unusual.

Macros are the main source of false positives. They represent logical operations that may not map to concrete actions. For example, networking code contains many calls of the form "x = ntohs(x)" used to reorder the bytes in variable x in a canonical "network order" so that a machine receiving the data can unpack it appropriately. However, on machines on which the data is already in network order, the macro will expand to nothing, resulting in code that will simply assign x to itself. To suppress these false positives, we modified the preprocessor to note which lines contain macros—we simply ignore warnings on these lines.

3 Redundant Assignments

The checker in this section flags cases where a value assigned to a variable is not subsequently used. The checker tracks the lifetime of variables using a simple intraprocedural analysis. At each assignment, it follows the variable forward on all paths. It emits an error message if the variable is not read on any path before either exiting scope or being assigned another value. As we show, in many cases, such lost values signal real errors, such as control flow following unexpected paths, results computed but not returned, etc.

The checker found thousands of redundant assignments in the three systems we checked. Since it was so effective, we minimized the chance of false positives by radically restricting the variables it would follow to nonglobal and nonvolatile ones that were not aliased in any way (i.e., local variables that never had their addresses taken).

Most of the checker code deals with differentiating the errors into three classes, which it ranks in the following order:

1. Variables assigned values that are not read, and which are never subsequently reassigned. Empirically, these errors tend to be the most serious since they flag unintentionally lost results.

2. Sun Workshop 6 update 2, GNU GCC 3.2.1, Compaq C V6.3-129, and MS Visual Studio .NET.

TABLE 2
Bugs Found by the Redundant Assignment Checker in Linux 2.4.5-ac8, OpenBSD 3.2, and PostgreSQL 7.2

System	Bugs	False	Uninspected
Linux 2.4.5-ac8	129	26	1840
OpenBSD 3.2	63	36	117
PostgreSQL 7.2	37	10	0

2. Variables assigned a nonconstant value that are not read, and which are subsequently reassigned. These are also commonly errors, but tend to be less severe. False positives in this class tend to come from assigning a return value from a function call to a dummy variable that is ignored, which is prevalent in PostgreSQL. Fortunately, such variables tend to share a consistent naming pattern (e.g., they are commonly prefixed with double underscores (_ _) in PostgreSQL) and, therefore, can be easily suppressed with grep. In presenting bug counts in Table 2, we do not count warnings that are so suppressed.
3. Variables assigned a constant and then reassigned other values without being read. These are frequently due to defensive programming, where the programmer always initializes a variable to some safe value (most commonly: NULL, 0, 0xffffffff, and -1) but does not read it before redefinition. We track the initial value and emit it when reporting the error so that messages with a common defensive value can be easily filtered out. Again, we do not count filtered messages in Table 2.

Suppressing false positives. As with many redundant checkers, macros and defensive programming cause most false positives. To minimize the impact of macros, the checker does not track variables killed or generated within them. Its main remaining vulnerability is values assigned and then passed to debugging macros that are turned off:

```
x = foo->bar;
DEBUG("bar = %d", x);
```

Typically, there are a small number of such debugging macros, which we manually turn back on by modifying the header files in which they are defined.

We use ranking to minimize the impact of defensive programming. Redundant operations that can be errors when done within the span of a few lines can be robust programming practice when separated by 20 lines. Thus, we rank reported errors based on 1) the line distance between the assignment and reassignment and 2) the number of conditional branches on the path. Close errors are most likely; farther ones arguably defensive programming.

The errors. This checker found more errors than all the others we have written combined. There were two interesting error patterns that showed up as redundant assignments: 1) variables whose values were (unintentionally) discarded and 2) variables whose values were not used because of surprising control flow (e.g., an unexpected return).

```
/* linux2.4.5-ac8/net/decnet/af_decnet.c:dn_wait_run */
do {
    . . .
    if (signal_pending(current)) {
        err = −ERESTARTSYS; /* BUG: lost value */
        break;
    }
    . . .
} while(scp−>state != DN_RUN);
return 0;     /* should be "return err" here */
```

Fig. 1. Lost return value caught by flagging the redundant assignment to err.

The majority of the errors (126 of the 129 diagnosed ones) fall into the first category. Fig. 1 shows a representative example from Linux. Here, if the function signal_pending returns true (a signal is pending to the current process), an error code is set (err = -ERESTARTSYS) and the code breaks out of the enclosing loop. The value in err must be passed back to the caller so that it will retry the system call. However, the code always returns 0 to the caller, no matter what happens inside the loop. This will lead to an insidious error: The code usually works but, occasionally, it will abort but return a success code, causing the client to assume that the operation happened.

There were numerous similar errors on the caller side where the result of a function was assigned to a variable, but then ignored rather than being checked. The fact that logically the code contains errors is readily flagged by looking for variables assigned but not used.

The second class contains three diagnosed errors that comes from calculations aborted by unexpected control flow. Fig. 2 gives one example from Linux: Here, all paths through a loop end in a return, wrongly aborting the loop after a single iteration. This error is caught by the fact that an assignment used to walk down a linked list is never read because the loop iterator that would do so is dead code. Fig. 3 shows a variation on the theme of unexpected control flow. Here, an if statement has an extraneous statement terminator at its end, making the subsequent return to be

```
/* linux2.4.1/net/atm/lec.c:lec_addr_delete: */
for(entry=priv−>lec_arp_tables[i];
    entry != NULL;
    entry=next) {    /* BUG: never reached */
  next = entry−>next;
  if (. . .) {
      lec_arp_remove(priv−>lec_arp_tables, entry);
      kfree(entry);
  }
  lec_arp_unlock(priv);
  return 0;
}
```

Fig. 2. A single-iteration loop caught by flagging the redundant assignment next = entry→maps next. The assignment appears to be read in the loop iteration statement (entry = next), but it is dead code since the loop always exits after a single iteration. The logical result will be that, if the entry the loop is trying to delete is not the first one in the list, it will not be deleted.

```
/* linux2.4.5-ac8/fs/ntfs/unistr.c:ntfs_collate_names */
for (cnt = 0; cnt < min(name1_len, name2_len); ++cnt) {
    c1 = le16_to_cpu(*name1++);
    c2 = le16_to_cpu(*name2++);
    if (ic) {
        if (c1 < upcase_len)
            c1 = le16_to_cpu(upcase[c1]);
        if (c2 < upcase_len)
            c2 = le16_to_cpu(upcase[c2]);
    }
    /* [META] stray terminator! */
    if (c1 < 64 && legal_ansi_char_array[c1] & 8);
        return err_val;
    if (c1 < c2)
        return −1;
    . . .
```

Fig. 3. Catastrophic return caught by the redundant assignment to c2. The second to last conditional is accidentally terminated because of a stray statement terminator (";") at the end of the line, causing the routine to always return err_val.

always taken. In these cases, a coding mistake caused "dangling assignments" that were not used. This fact allows us to flag such bogus structures, even when we do not know how control flows in the code. It is the presence of these errors that has led us to write the dead-code checker in the next section.

Reassigning values is typically harmless, but it does signal fairly confused programmers. For example:

```
/* linux2.4.5-ac8/drivers/net/wan/sdla_x25.c:
                        alloc_and_init_skb_buf */
struct sk_buff *new_skb = *skb;
new_skb = dev_alloc_skb(len+ X25_HRDHDR_SZ);
```

where new_skb is assigned the value *skb, but then immediately reassigned another allocated value. A different case shows a potential confusion about how C's iteration statement works:

```
/* linux2.4.5-ac8/drivers/scsi/scsi.c:
                   scsi_bottom_half_handler */
SCnext = SCpnt->bh_next;
for (; SCpnt; SCpnt = SCnext) {
SCnext = SCpnt->bh_next;
```

Note that the variable SCnext is assigned and then immediately reassigned in the loop. The logic behind this decision remains unclear.

The most devious error. A few of the values reassigned before being used were suspicious lost values. One of the worst (and most interesting) was from a commercial system which had the equivalent of the following code:

```
c= p->buf[0][3];
c= p->buf[0][3];
```

At first glance, this seems like a harmless obvious copy-and-paste error. It turned out that the redundancy flags a much more devious bug. The array buf actually pointed to a "memory mapped" region of kernel memory. Unlike normal memory, reads and writes to this region cause the CPU to issue I/O commands to a hardware device. Thus,

TABLE 3
Bugs Found by the Dead Code Checker on
Linux 2.4.5-ac8, OpenBSD 3.2, and PostgreSQL 7.2

System	Bugs	False
Linux 2.4.5-ac8	66	26
OpenBSD 3.2	11	4
PostgreSQL 7.2	0	0

the reads are not idempotent, and the two of them in a row rather than just one can cause very different results to happen. However, the above code does have a real (but silent) error—in the variant of C that this code was written, pointers to device memory must be declared as "volatile." Otherwise, the compiler is free to optimize duplicate reads away, especially since, in this case, there were no pointer stores that could change their values. Dangerously, in the above case, buf was declared as a normal pointer rather than a volatile one, allowing the compiler to optimize as it wished. Fortunately, the error had not been triggered because the GNU C compiler that was being used had a weak optimizer that conservatively did not optimize expressions that had many levels of indirection. However, the use of a more aggressive compiler or a later version of gcc could have caused this extremely difficult to track down bug to surface.

4 Dead Code

The checker in this section flags dead code (see Table 3). Since programmers generally write code to run it, dead code catches logical errors signaled by false beliefs that unreachable code can execute.

The core of the dead code checker is a straightforward mark-and-sweep algorithm. For each routine, it 1) marks all blocks reachable from the routine's entry node and 2) traverses all blocks in the routine, flagging any that are not marked. The checker has three modifications to this basic algorithm. First, it truncates all paths that reach functions that would not return. Examples include "panic," "abort," and "BUG," which are used by Linux to signal a terminal kernel error and reboot the system—

```
/* linux2.4.1/net/ipv6/raw.c:rawv6_getsockopt */
switch (optname) {
  case IPV6_CHECKSUM:
     if (opt->checksum == 0)
           val = -1;
     else
           val = opt->offset;
     /* BUG: always falls through */
  default:
     return -ENOPROTOOPT;
  }
  len=min(sizeof(int),len);
  . . .
```

Fig. 4. Unintentional switch "fall through" causing the code to always return an error. This maps to the low-level redundancy that the value assigned to val is never used.

```
/* linux2.4.1/drivers/char/rio/rioparam.c:RIOParam */
if (retval == RIO_FAIL) {
   rio_spin_unlock_irqrestore(&PortP->portSem, flags);
   pseterr(EINTR);     /* BUG: returns */
   func_exit();
   return RIO_FAIL;
}
```

Fig. 5. Unexpected return: The call pseterr is a macro that returns its argument value as an error. Unfortunately, the programmer does not realize this and inserts subsequent operations, which are flagged by our dead code checker. There were many other similar misuses of the same macro.

code dominated by such calls cannot run. Second, we suppress error messages for dead code caused by constant conditions involving macros or the sizeof operator, such as

```
/* case 1: debugging statement that is turned 'off' */
#define DEBUG 0
if (DEBUG) printf("in_foo");
/* case 2: gcc coverts the condition to 0
      on 32-bit architectures */
if (sizeof(int) == 64)
      printf("64 bit architecture\n");
```

The former frequently signals code "commented out" by using a false condition, while the latter is used to carry out architecture dependent operations. We also annotate error messages when the flagged dead code is only a break or return. These are commonly a result of defensive programming. Finally, we suppress dead code in macros.

Despite its simplicity, dead code analysis found a high number of clearly serious errors. Three of the errors caught by the redundant assignment checker are also caught by the dead code detector: 1) the single iteration loop in Fig. 2, 2) the mistaken statement terminator in Fig. 3, and 3) the unintentional fall through in Fig. 4.

Fig. 5 gives the most frequent copy-and-paste error. Here, the macro "pseterr" has a return statement in its definition, but the programmer does not realize it. Thus, at all seven call sites that use the macro, there is dead code after the macro that the programmer intended to have executed.

Fig. 6 gives another common error—a single-iteration loop that always terminates because it contains an if-else statement that breaks out of the loop on both branches. It is hard to believe that this code was ever tested. Fig. 7 gives a variation on this, where one branch of the if statement breaks out of the loop but the other uses C's continue statement, which skips the rest of the loop body. Thus, none of the code thereafter can be executed.

Type errors can also result in dead code. Fig. 8 shows an example from OpenBSD. Here, the function tv_channel returns -1 on error, but, since temp is an unsigned variable, the error handling code in the true branch of the if statement is never executed. An obvious fix is to declare temp as int. As before, none of the four compilers we tested warned about this suspicious typing mistake.

```
/* linux2.4.1/drivers/scsi/53c7,8xx.c:
    return_outstanding_commands */
for (c = hostdata->running_list; c;
     c = (struct NCR53c7x0_cmd *) c->next) {
     if (c->cmd->SCp.buffer) {
          printk ("...");
          break;
     } else {
          printk ("Duh? ...");
          break;
     }
     /* BUG: cannot be reached */
     c->cmd->SCp.buffer =
          (struct scatterlist *) list;
     ...
```

Fig. 6. Broken loop: The first if-else statement of the loop contains a break on both paths, causing the loop to always abort, without ever executing the subsequent code it contains.

5 Redundant Conditionals

The checker in this section uses path-sensitive analysis to detect redundant (always true or always false) conditionals in branch statements, such as if, while, switch, etc. (see Table 4). They cannot affect the program state or control flow. Thus, their presence is a likely indicator of errors. To avoid double reporting bugs found in the previous section, we only flag nonconstant conditional expressions that are not evaluated by the dead code checker.

The implementation of the checker uses the false path pruning (FPP) feature in the *xgcc* system [15]. FPP was originally designed to eliminate false positives from infeasible paths. It prunes a subset of logically inconsistent paths by keeping track of assignments and conditionals along the way. The implementation of FPP consists of three separate modules, each keeping track of one class of program properties as described below:

1. The first module maintains a mapping from variables to integer constants. It tracks assignments (e.g., x = 1; x = x * 2;) and conditional branch statements (e.g., if (x == 5) {...}) along each execution path to derive variable-constant bindings.
2. The second module keeps track of a known set of predicates that must hold true along the current program path. These predicates are collected from conditional branches (e.g., in the true branch of if (x != NULL) {...}, we will collect x != NULL into the known predicate set), and they will be used later to test the validity of subsequent control predicates (e.g., if we encounter if (x != NULL) later in the program, we will prune the else branch). Note if the value of variable x is killed (either directly by assignments or indirectly through pointers), we remove any conditional in the known predicate set that references x.
3. The third module keeps track of constant bounds of variables. We derive these bounds from conditional statements. For example, in the true branch of if (x <= 5), we enter 5 as x's upper bound. We prune paths that contradict with the recorded bound information (e.g., the true branch of if (x > 7)).

```
/* linux2.4.5-ac8/net/decnet/dn_table.c:
        dn_fib_table_lookup */
for(f = dz_chain(k, dz); f; f = f->fn_next) {
     if (!dn_key_leq(k, f->fn_key))
          break;
     else
          continue;

     /* BUG: cannot be reached */
     f->fn_state |= DN_S_ACCESSED;

     if (f->fn_state&DN_S_ZOMBIE)
          continue;
     if (f->fn_scope < key->scope)
          continue;
```

Fig. 7. Useless loop body: Similarly to Fig. 6, this loop has a broken if-else statement. One branch aborts the loop, the other uses C's continue statement to skip the body and begin another iteration.

```
/* openbsd3.2/sys/dev/pci/bktr/bktr_core.c:tuner_ioctl */
unsigned temp;
...
temp = tv_channel( bktr, (int)*(unsigned long *)arg );
if ( temp < 0 ) { /* gcc frontend turns this into 0 */
     /* BUG: cannot be reached */
     temp_mute( bktr, FALSE );
     return( EINVAL );
}
*(unsigned long *)arg = temp;
```

Fig. 8. Unsigned variable tested for negativity.

These three simple modules were able to capture enough information that allows us to find interesting errors in Linux and OpenBSD.

With FPP, the checker works as follows: For each function, it traverses the control flow graph (CFG) with FPP and marks all reachable CFG edges. At the end of the analysis, it emits conditionals associated to untraversed ones as errors.

Macros and concurrency are the two major sources of false positives. To suppress those, we discard warnings that take place within macros and ignore conditionals that involve global, static, or volatile variables whose values might be changed by another thread.

The checker is able to find hundreds of redundant conditionals in Linux and OpenBSD. We classify them into three major categories which we describe below.

TABLE 4
Bugs Found by the Redundant Conditionals Checker in Linux 2.4.5-ac8, OpenBSD 3.2, PostgreSQL 7.2

System	Bugs	False	Uninspected
Linux 2.4.5-ac8	49	52	169
OpenBSD 3.2	64	33	316
PostgreSQL 7.2	0	0	0

```
1  /* linux2.4.5-ac8/fs/fat/inode.c */
2  error = 0;
3  if (!error) { /* causes unnecessary code complexity */
4      sbi->fat_bits = fat32 ? 32 :
5          (fat ? fat :
6          (sbi->clusters > MSDOS_FAT12 ? 16 : 12));
```

Fig. 9. Nonsensical programming style: The check at line 3 is clearly redundant.

The first class of errors, which are the least serious of the three, are labeled as "nonsensical programming style." Fig. 9 shows a representative example from Linux 2.4.5-ac8. The if statement at line 3 is clearly redundant and leaves one to wonder about the purpose of such a check. These errors, although harmless, signal a confused programmer. The latter conjecture is supported by the statistical analysis described in Section 8.

Fig. 10 shows a more problematic case. The second if statement is redundant because the first one has already taken care of the case where slave is false. The fact that a different error code is returned signals a possible bug in this code.

The second class of errors are again seemingly harmless on the surface, but, when we give a more careful look at the surrounding code, we often find serious errors. The while loop in Fig. 11 is obviously trying to recover from hardware errors encountered when reading network packets. But, since the variable err is not updated in the loop body, it would never become SUCCESS and, thus, the loop body will never be executed more than once, which is suspicious. This signals a possible error where the programmer forgets to update err in the large chunk of recovery code in the loop. This bug would be difficult to detect by testing, because it is in an error handling branch that is only executed when the hardware fails in a certain way.

The third class of errors is clearly composed of serious bugs. Fig. 12 shows an example from Linux 2.4.5-ac8. As we can see, the second and third if statements carry out entirely different actions on identical conditions. Apparently, the programmer has cut-and-pasted the conditional without changing one of the two NODE_LOGGED_OUTs into a more likely fourth possibility: NODE_NOT_PRESENT.

```
1  /* linux2.4.5-ac8/drivers/net/tokenring/smctr.c:
2                              smctr_rx_frame */
3  while((status = tp->rx_fcb_curr[queue]
4                  ->frame_status) != SUCCESS)
5  {
6      err = HARDWARE_FAILED;
7      ... /* large chunk of apparent recovery code,
8              with no updates to err */
9      if (err != SUCCESS)
10         break;
11 }
```

Fig. 11. Redundant conditional that suggests a serious program error.

Fig. 13 shows another serious error. The author obviously wanted to insert sp into a doubly-linked list that starts from q->q_first, but the while loop clearly does nothing other than setting srb_p to NULL, which is nonsensical. The checker detects this error by inferring that the ensuing if statement is redundant. An apparent fix is to replace the while condition (srb_p) with (srb_p && srb_p->next). This bug can be dangerous and hard to detect, because it quietly discards everything that was in the original list and constructs a new one with sp as its sole element. As a matter of fact, the same error is still present in the latest stable 2.4.20 release of the Linux kernel source as of this writing.

6 Redundant Null-Checks

The checker described in this section uses redundancies to flag misunderstandings of function interfaces in Linux. Certain functions, such as kmalloc, return a NULL pointer on failure. Callers of these functions need to check the validity of their return values before they can safely dereference them. In prior work [11], we described an algorithm that automatically derives the set of potential NULL-returning functions in Linux. Here, we use the

```
1  /* linux2.4.5-ac8/drivers/net/wan/sbni.c:sbni_ioctl */
2  slave = dev_get_by_name(tmpstr);
3  if(!(slave && slave->flags & IFF_UP &&
4       dev->flags & IFF_UP))
5  {
6    ... /* print some error message, back out */
7    return -EINVAL;
8  }
9  if (slave) {    ... }
10 /* BUG: !slave is impossible */
11 else {
12   ... /* print some error message */
13   return -ENOENT;
14 }
```

Fig. 10. Nonsensical programming style: The check of slave at line 9 is guaranteed to be true and also notice the difference in return value.

```
1  /* linux2.4.1/drivers/fc/iph5526.c:
2                      rscn_handler */
3  if ((login_state == NODE_LOGGED_IN) ||
4      (login_state == NODE_PROCESS_LOGGED_IN)) {
5      ...
6  }
7  else
8  if (login_state == NODE_LOGGED_OUT)
9      tx_adisc(fi, ELS_ADISC, node_id,
10         OX_ID_FIRST_SEQUENCE);
11 else
12 /* BUG: redundant conditional */
13 if (login_state == NODE_LOGGED_OUT)
14     tx_logi(fi, ELS_PLOGI, node_id);
```

Fig. 12. Redundant conditionals that signal errors: a conditional expression being placed in the else branch of another identical one.

```
/* linux2.4.5-ac8/drivers/scsi/qla1280.c:
                              qla1280_putq_t */
srb_p = q->q_first;
while (srb_p )
   srb_p = srb_p->s_next;

if (srb_p) { /* BUG: this branch is never taken*/
   sp->s_prev = srb_p->s_prev;
   if (srb_p->s_prev)
      srb_p->s_prev->s_next = sp;
   else
      q->q_first = sp;
   srb_p->s_prev = sp;
   sp->s_next = srb_p;
} else {
   sp->s_prev = q->q_last;
   q->q_last->s_next = sp;
   q->q_last = sp;
}
```

Fig. 13. A serious error in a linked list insertion implementation: srb_p is always null after the while loop (which appears to be checking the wrong Boolean condition).

```
1 /* linux2.5.53/fs/ntfs/attrib.c:ntfs_merge_run_lists */
2 if (!slots) {
3     /* FIXME/TODO: We need to have the extra
4      * memory already! (AIA) */
5     drl = ntfs_rl_realloc(drl, ds, ds + 1);
6     if (!drl) /* BUG: drl is never NULL */
7         goto critical_error;
8 }
```

Fig. 14. Redundant NULL-check of drl signals a more serious problem: return values of ntfs_rl_realloc should in fact be checked with IS_ERR. A NULL-check will never catch the error case.

logical opposite of that algorithm to flag functions whose return values should never be checked against NULL. A naive view is that, at worst, such redundant checks are minor performance mistakes. In practice, we found they can flag two dangerous situations.

1. The programmer believes that a function can fail when it cannot. If they misunderstand the function's interface at this basic level, they likely misunderstand other aspects.
2. The programmer correctly believes that a function can fail, but misunderstands how to check for failure. Linux and other systems have functions that indicate failure in some way other than returning a null pointer.

For each pointer-returning function f, the checker tracks two counts:

1. The number of call sites where the pointer returned by f was checked against null before use.
2. The number of call sites where the returned pointer was not checked against null before use.

A function f whose result is often checked against NULL implies the belief that f could potentially return NULL. Conversely, many uses with few NULL-checks implies the belief that the function should not be checked against NULL. We use the z-statistic [14] to rank functions from most to least likely to return NULL based on these counts. Return values from functions with highest z-values should probably be checked before use, whereas NULL-checks on those from the lowest ranked functions are most likely redundant.

Fig. 14 shows one of the two bugs we found in a recent release of Linux. Here, the redundant NULL-check at line 6 signals the problem: The programmer has obviously misunderstood the interface to the function ntfs_rl_realloc, which, as shown below,

```
/* 2.5.53/fs/ntfs/attrib.c:ntfs_rl_realloc */
. . .
new_rl = ntfs_malloc_nofs(new_size);
if (unlikely(!new_rl))
   returnERR PTR(-ENOMEM);
. . .
```

will never return NULL. Instead, on memory exhaustion, it will return what is essentially ((void*)-ENOMEM), which should be checked using the special IS_ERR macro. When ntfs_rl_realloc fails, the null check will fail too and the code will dereference the returned value, which will likely correspond to a valid physical address, causing a very difficult-to-diagnose memory corruption bug. Unsurprisingly, the same error appeared again at another location in this author's code.

7 Predicting Hard Errors with Redundancies

In this section, we show the correlation between redundant errors and hard bugs that can crash a system. The redundant errors are collected from the redundant assignment checker, the dead code checker, and the redundant conditional checker.[3] The hard bugs were collected from Linux 2.4.1 with checkers described in [8]. These bugs include use of freed memory, dereferences of null pointers, potential deadlocks, unreleased locks, and security violations (e.g., the use of an untrusted value as an array index). They have been reported to and largely confirmed by Linux developers. We show that there is a strong correlation between these two error populations using a statistical technique called the contingency table method [6]. Further, we show that a file containing a redundant error is roughly 45 to 100 percent more likely to have a hard error than one selected at random. These results indicate that 1) files with redundant errors are good audit candidates and 2) redundancy correlates with confused programmers who will probably make a series of mistakes.

7.1 Methodology

This section describes the statistical methods used to measure the association between program redundancies and hard errors. Our analysis is based on the 2×2 contingency table [6] method. It is a standard statistical

3. We exclude the idempotent operation and redundant NULL-check results because the total number of bugs is too small to be statistically significant.

TABLE 5
Contingency Table: Redundant Assignments vs. Hard Bugs

Redundant	Hard Bugs		
Assignments	Yes	No	Totals
Yes	345	435	780
No	206	1069	1275
Totals	551	1504	2055

$T = 194.37$, p-value $= 0.00$

There are 345 files with both error types, 435 files with a redundant assignments and no hard bugs, 206 files with a hard bug and no redundant assignments, and 1,069 files with no bugs of either type. A T-statistic *value above four gives a* p-value *of less than 0.05, which strongly suggests the two events are not independent. The observed* T *value of 194.37 gives a* p-value *of essentially 0, noticeably better than the standard threshold. Intuitively, the correlation between error types can be seen in that the ratio of 345/435 is considerably larger than the ratio 206/1,069—if the events were independent, we expect these two ratios to be approximately equal.*

tool for studying the association between two different attributes of a population. In our case, the population is the set of files we have checked in Linux, and the two attributes are: 1) whether a file contains redundancies and 2) whether it contains hard errors.

In the contingency table approach, the population is cross-classified into four categories based on two attributes, say **A** and **B**, of the population. We obtain counts (o_{ij}) in each category and tabulate the results as follows:

The values in the margin ($n_{1\cdot}, n_{2\cdot}, n_{\cdot 1}, n_{\cdot 2}$) are row and column totals, and $n_{\cdot\cdot}$ is the grand total. The null hypothesis H_0 of this test is that **A** and **B** are mutually independent, i.e., knowing **A** gives no additional information about **B**. More precisely, if H_0 holds, we expect that:[4]

$$\frac{o_{11}}{o_{11}+o_{12}} \approx \frac{o_{21}}{o_{21}+o_{22}} \approx \frac{n_{\cdot 1}}{n_{\cdot 1}+n_{\cdot 2}}.$$

We can then compute expected values (e_{ij}) for the four cells in the table as follows:

$$e_{ij} = \frac{n_{i\cdot} n_{\cdot j}}{n_{\cdot\cdot}}.$$

We use a "chi-squared" test statistic [14]:

$$T = \sum_{i,j \in \{1,2\}} \frac{(o_{ij} - e_{ij})^2}{e_{ij}}$$

to measure how far the observed values (o_{ij}) deviates from the expected values (e_{ij}). Using the T statistic, we can derive the the probability of observing o_{ij} if the null hypothesis H_0 is true. This probability is called the p-value.[5] The smaller the p-value, the stronger the evidence against H_0, thus the stronger the correlation between attributes **A** and **B**.

TABLE 6
Contingency Table: Dead Code vs. Hard Bugs

	Hard Bugs		
Dead Code	Yes	No	Totals
Yes	133	135	268
No	418	1369	1787
Totals	551	1504	2055

$T = 81.74$, p-value $= 0.00$

TABLE 7
Contingency Table: Redundant Conditionals vs. Hard Bugs

Redundant	Hard Bugs		
Conditionals	Yes	No	Totals
Yes	75	79	154
No	476	1425	1901
Totals	551	1504	2055

$T = 40.65$, p-value $= 0.00$

7.2 Data Acquisition and Test Results

In our previous work [8], we used the *xgcc* system to check 2,055 files in Linux 2.4.1. These focused on serious system crashing hard bug and collected more than 1,800 serious hard bugs in 551 files. The types of bugs we checked for included null pointer dereference, deadlocks, and missed security checks. We use these bugs to represent the class of serious hard errors and derive correlation with program redundancies.

We cross-classify the program files in the Linux kernel into the following four categories and obtain counts in each:

1. o_{11}: number of files with both redundancies and hard errors.
2. o_{12}: number of files with redundancies, but not hard errors.
3. o_{21}: number of files with hard errors, but not redundancies.
4. o_{22}: number of files with neither redundancies nor hard errors.

We can then carry out the test described in Section 7.1 for the redundant assignment checker, dead code checker, and redundant conditional checker.

The result of the tests are given in Tables 5, 6, 7, and 8. Note that, in the aggregate case, the total number of redundancies is less than the sum of number of redundancies from each of the four checkers. That is because we avoid double counting files that are flagged by two or more checkers. As we can see, the correlation between redundancies and hard errors are extremely high, with p-values being approximately 0 in all four cases. The results strongly suggest that redundancies often signal confused programmers and, therefore, are a good predictor for hard, serious errors.

7.3 Predicting Hard Errors

In addition to the qualitative measure of correlation, we want to know quantitatively how much more likely it is that

4. To see this is true, consider 100 white balls in an urn. We first randomly draw 40 of them and put a red mark on them. We put them back in the urn. Then, we randomly draw 80 of them and put a blue mark on them. Obviously, we should expect roughly 80 percent of the 40 balls with red marks to have blue marks, as should we expect roughly 80 percent of the remaining 60 balls without the red mark to have a blue mark.

5. Technically, under H_0, T has a χ^2 distribution with one degree of freedom. The p-value can be looked up in the cumulative distribution table of the χ^2_1 distribution. For example, if T is larger than 4, the p-value will go below 5 percent.

TABLE 8
Contingency Table: Program Redundancies (Aggregate) vs. Hard Bugs

	Hard Bugs		
Aggregate	Yes	No	Totals
Yes	372	573	945
No	179	931	1110
Totals	551	1504	2055

$T = 140.48$, p-value $= 0.00$

we will find hard errors in a file that contains one or more redundant operations. More precisely, let E be the event that a given source file contains one or more hard errors and R be the event that it has one or more forms of redundant operations, we can compute a confidence interval for $T' = (P(E|R) - P(E))/P(E)$, which measures how much more likely we are to find hard errors in a file given the presence of redundancies.

The prior probability of hard errors is computed as follows:

$$P(E) = \frac{\text{Number of files with hard errors}}{\text{Total number of files checked}} = \frac{551}{2,055} = 0.2681.$$

We tabulate the conditional probabilities and T' values in Table 9. (Again, we excluded the idempotent operations and redundant NULL-checks because of their small bug sample.) As shown in the table, given presence of any form of redundant operation, it is roughly 45-100 percent more likely we will find an error in that file than in a randomly selected file.

8 Detecting Specification Mistakes

This section describes how to use redundant code actions to find several types of specification errors and omissions. Often program specifications give extra information that allow code to be checked: whether return values of routines must be checked against NULL, which shared variables are protected by which locks, which permission checks guard which sensitive operations, etc. A vulnerability of this approach is that if a code feature is not annotated or included in the specification, it will not be checked. We can catch such omissions by flagging redundant operations. In the above cases, and in many others, at least one of the specified actions makes little sense in isolation—critical sections without shared states are pointless as are permission checks that do not guard known sensitive actions. Thus, if code does not intend to do useless operations, then such redundancies will happen exactly when checkable actions have been missed. (At the very least, we will have caught something pointless that should be deleted.) We sketch four examples below and close with two case studies that use redundancies to find missing checkable actions.

Detecting omitted null annotations. Tools such as Splint [12] let programmers annotate functions that can return a null pointer with a "null" annotation. The tool emits an error for any unchecked use of a pointer returned from a null routine. In a real system, many functions can return null, making it easy to forget to annotate them all. We can catch such omissions using redundancies. We know only the return value of null functions should be checked. Thus, a check on a nonannotated function means that either the function: 1) should be annotated with null or 2) the function cannot return null and the programmer has misunderstood the interface. A variant of this technique has been applied with success in [11] and also in the checker described in Section 6.

Finding missed lock-variable bindings. Data race detection tools such as Warlock [24] let users explicitly bind locks to the variables they protect. The tool warns when annotated variables are accessed without their lock held. However, lock-variable bindings can easily be forgotten, causing the variable to be (silently) unchecked. We can use redundancies to catch such mistakes. Critical sections must protect *some* shared state: flagging those that do not will find either 1) useless locking (which should be deleted for better performance) or 2) places where a shared variable was not annotated.

Missed "volatile" annotations. As described in Section 4, in C, variables with unusual read/write semantics must be annotated with the "volatile" type qualifier to prevent the compiler from doing optimizations that are safe on normal variables, but incorrect on volatile ones, such as eliminating duplicate reads or writes. A missing volatile annotation is a silent error, in that the software will usually work, but only occasionally give incorrect results on certain hardware-compiler combinations. As shown, such omissions can be detected by flagging redundant operations (reads or writes) that do not make sense for nonvolatile variables.

Missed permission checks. A secure system must guard sensitive operations (such as modifying a file or killing a process) with permission checks. A tool can automatically catch such mistakes given a specification of which checks protect which operations. The large number of sensitive operations makes it easy to forget a binding. As before, we can use redundancies to find such omissions: Assuming programmers do not do redundant permission checks,

TABLE 9
Program Files with Redundancies are, on Average, Roughly 50 Percent More Likely to Contain Hard Errors

R	$R \wedge E$	R	$P(E\|R)$	$P(E\|R) - P(E)$	Standard Error	95% Confidence Interval for T'
Assign	353	889	0.3971	0.1289	0.0191	$48.11\% \pm 13.95\%$
Dead Code	30	56	0.5357	0.2676	0.0674	$99.82\% \pm 49.23\%$
Conditionals	75	154	0.4870	0.2189	0.0414	$81.65\% \pm 30.28\%$
Aggregate	372	945	0.3937	0.1255	0.0187	$46.83\% \pm 13.65\%$

finding permission check that does not guard a known sensitive operation signals an incomplete specification.

8.1 Case Study: Finding Missed Security Holes

In a separate paper [3], we describe a checker that found operating system security holes caused when an integer read from untrusted sources (network packets, system call parameters) was passed to a trusting sink (array indices, length parameters in memory copy operations) without being checked against a safe upper and lower bound. A single violation can let a malicious attacker take control of the entire system. Unfortunately, the checker is vulnerable to omissions. An omitted source means the checker will not track the data produced. An omitted sink means the checker will not warn when unsanitized data reaches it.

When implementing the checker, we used the ideas in this section to detect such omissions. Given a list of known sources and sinks, the normal checking sequence is: 1) the code reads data from an unsafe source, 2) checks it, and 3) passes it to a trusting sink. Assuming programmers do not do gratuitous sanitization, a missed sink can be detected by flagging when code does steps 1 and 2, but not 3. Reading a value from a known source and sanitizing it implies the code believes the value will reach a dangerous operation. If the value does not reach a known sink, we have likely missed one in our specification. Similarly, we could (but did not) infer missed sources by doing the converse of this analysis: flagging when the OS sanitizes data we do not think is tainted and then passes it to a trusting sink.

The analysis found roughly 10 common uses of sanitized inputs in Linux 2.4.6 [3]. Nine of these uses were harmless; however, one was a security hole. Unexpectedly, this was not from a specification omission. Rather, the sink was known, but our interprocedural analysis had been overly simplistic, causing us to miss the path to it. The fact that redundancies flag errors both in the specification and in the tool itself was a nice surprise.

8.2 Case Study: Helping Static Race Detection

We have developed a static race detection tool, RacerX [10], that has been dramatically improved by explicitly using the fact that programmers do not perform redundant operations.

At a high level, the tool is based on a static lockset algorithm similar to the dynamic version used in Eraser [23]. It works roughly as follows: 1) The user supplies a list of locking functions and a source base to check; 2) the tool compiles the source base and does a context-sensitive interprocedural analysis to compute the set of locks held at all program points; 3) RacerX warns when shared variables are used without a consistent lock held.

This simple approach needs several modifications to be practical. We describe two problems below that can be countered in part by using the ideas in this paper.

First, it is extremely difficult to determine if an unprotected access is actually an bug. Many unprotected accesses are perfectly acceptable. For example, programmers intentionally do unprotected modifications of statistics variables for speed, or they may orchestrate reads and writes of shared variables to be noninterfering (e.g., a variable that has a single reader and writer may not need locking). An unprotected access is only an error if it allows an application-specific invariant to be violated. Thus, reasoning about such accesses requires understanding complex code invariants and actual interleavings (rather than potential ones), both of which are often undocumented. In our experience, a single race condition report can easily take tens of minutes to diagnose. Even at the end, it may not be possible to determine if the report is actually an error. In contrast, other types of errors found with static analysis often take seconds to diagnose (e.g., uses of freed variables, not releasing acquired locks).

We can simplify this problem using a form of redundancy analysis. The assumption that programmers do not write redundant critical sections, implies that the first, last, and only shared data accesses in a critical section are special:

- If a variable or function call is the only statement within the critical section, we have very strong evidence that the programmer thinks 1) the state should be protected in general and 2) that the acquired lock enforces this protection.
- Similarly, the first and last accesses of shared states in a critical section also receive special treatment (although to a lesser degree) since programmers often acquire a lock on the first shared state access that must be protected and release it immediately after the last one—i.e., they do not gratuitously make critical sections large.

A crucial result of these observations is that displaying such examples of where a variable or function was explicitly protected makes it very clear to a user of RacerX what exactly is being protected. Fig. 15 gives a simple example of this from Linux. There were 37 accesses to serial_out with the argument info with some sort of lock held; in contrast, there was only one unlocked use. This function-argument pair was the first statement of a critical section 11 times and the last one 17 times. Looking at the examples, it is obvious that the programmer is explicitly disabling interrupts[6] before invoking this routine. In particular, we do not have to look at the implementation of serial_out and try to reason about whether it or the device it interacts with needs to be protected with disabled interrupts. In practice, we almost always look at errors that have such features over those that do not.

A second problem with static race detection is that many seemingly multithreaded code paths are actually single-threaded. Examples include operating system interrupt handlers and initialization code that runs before other threads have been activated [23]. Warnings for accesses to shared variables on single-threaded code paths can swamp the user with false positives, at the very least hiding real errors and, in the worse case, causing the user to discard the tool.

This problem can also be countered using redundancy analysis. We assume that in general programmers do not do

6. cli() clears the Interrupt Enable Flag on x86 architectures, thus it prevents preemption and has the effect of acquiring a global kernel lock on single processor systems. sti() and restore_flags(flags) restores the Interrupt Enable Flag. They can be thought of as releasing the kernel lock previously acquired by cli().

```
/* ERROR: linux-2.5.62/drivers/char/esp.c:
 *     2313:block_til_ready: calling <serial_out-info>
 *     without "cli()"!
 */
restore_flags(flags); /* re-enable interrupts */
set_current_state(TASK_INTERRUPTIBLE);
if (tty_hung_up_p(filp)
|| !(info->flags & ASYNC_INITIALIZED)) {
     ...
}

/* non-disabled access to serial_out-info! */
serial_out(info, UART_ESI_CMD1, ESI_GET_UART_STAT);
if (serial_in(info, UART_ESI_STAT2) & UART_MSR_DCD)
          do_clocal = 1;
...

/* Example 1 drivers/char/esp.c:1206 */
save_flags(flags); cli();
/* set baud */
serial_out(info, UART_ESI_CMD1, ESI_SET_BAUD);
serial_out(info, UART_ESI_CMD2, quot >> 8);
serial_out(info, UART_ESI_CMD2, quot & 0xff);
restore_flags(flags);

...

/* Example 2: drivers/char/esp.c:1426 */
cli();
info->IER &= ~UART_IER_RDI;
serial_out(info, UART_ESI_CMD1, ESI_SET_SRV_MASK);
serial_out(info, UART_ESI_CMD2, info->IER);
serial_out(info, UART_ESI_CMD1, ESI_SET_RX_TIMEOUT);
serial_out(info, UART_ESI_CMD2, 0x00);
sti();
```

Fig. 15. Error ranked high because of redundancy analysis: There were 28 places where the routine serial_out was used as the first or last statement in a critical section.

spurious locking. We can thus infer that any concurrency operation implies that the calling code is multithreaded. These operations include locking, as well as calls to library routines that provide atomic operations such as atomic_add or test_and_set. (From this perspective, concurrency calls can be viewed as carefully inserted annotations specifying that code is multithreaded.)

RacerX considers any function to be multithreaded if concurrency operations occur 1) anywhere within the function or 2) anywhere above it in the call chain.[7] Note that such operations below the function may not indicate the function itself is multithreaded. For example, it could be calling library code that always conservatively acquires locks. RacerX computes the information for 1) and 2) in two passes. First, it walks over all functions, marking them as multithreaded if they do explicit concurrency operations within the function body. Second, when doing the normal lockset computation, it also tracks if it has hit a known multithreaded function. If so, it adds this annotation to any error emitted.

Finally, static checkers have the invidious problem that, errors in their analysis, often cause silent false negatives. Redundancy analysis can help find these: Critical sections that contain no shared states imply that either the programmer made a mistake or the analysis did. We found eight errors in the RacerX implementation when we modified the tool to flag empty critical sections. There were six minor errors, one error where we mishandled arrays (and so ignored all array uses indexed by pointer variables), and a particularly nasty silent lockset caching error that caused us to miss over 20 percent of all code paths.

7. Since RacerX is a static tool, we approximate this information by a simple reachability analysis on the static call graph.

9 Related Work

In writing the five redundancy checkers, we leverage heavily from existing research on program redundancy detection and elimination. Techniques such as partial redundancy elimination [18], [20], [21] and dead code elimination algorithms [1], [17], [19] have long been used in optimizing compilers to reduce code size and improve program performance. While our analyses closely mirror these ideas at their core, there are two key differences that distinguish our approach to that used in optimizing compilers:

1. Optimizers typically operate on a low-level intermediate representation (IR) with a limited set of primitive operations on typeless pseudoregisters. In contrast, our analyses need to operate at the source level and work with (or around) the full-blown semantic complexity of C. The reason is three-fold: 1) many redundant operations are introduced in the translation from source constructs to intermediate representations, making it hard to distinguish ones that pre-exist in the source program; 2) useful diagnostic information such as types (e.g., unsigned versus signed) and variable or macro names (e.g., DEBUG) are typically discarded during the translation to IR—we find such information essential in focused suppression of false positives; 3) the translation to IR usually changes the source constructs so much that reporting sensible warning messages at the IR level is extremely difficult, if not impossible.
2. While being sound and conservative is vital in optimizing compilers, error detection tools like ours can afford (and are often required) to be flexible for the sake of usefulness and efficiency. The redundant NULL checker in Section 6 harvests function interface specifications from a statistical analysis of their usage patterns. Although the analysis is effective, it is neither sound nor conservative and, therefore, would most likely not be admissible in optimizers.
3. Compilers are typically invoked far more frequently than checking tools during the development cycle. Therefore, speed is essential for the analyses being used in the optimizer. Expensive path-sensitive algorithms are carried out sparingly, if at all, on small portions of performance critical code simply because their time complexity usually outweighs their benefit. In contrast, checking tools like ours are less frequently run and can therefore afford to use more expensive analyses. The redundant conditional checker is one such example. The less stringent

speed requirement does give us a substantial edge in detecting more classes of errors with more accuracy.

Redundancy analyses have also been used in existing checking tools. Fosdick and Osterweil first applied data flow *anomaly detection* techniques in the context of software reliability. In their DAVE system [22], they used a depth first search algorithm to detect a set of variable def-use type of anomalies such as uninitialized read, double definition, etc. However, according to our experiments, path insensitive analysis like theirs produces an overwhelming number of false positives, especially for the uninitialized read checker. We were unable to find experimental validations of their approach to make a meaningful comparison.

Recent releases of the GNU C compiler (version 3 and up) provides users with the "-Wunreachable-code" option to detect dead code in the source program. However, their analysis provides no means of controlled suppression of false positives, which we find essential in limiting the number of false warnings. Also, because of its recent inception, the dead code detection algorithm is not yet fully functional[8] as of this writing.

Dynamic techniques [7], [16] instruments the program and detects anomalies that arise during execution. However, they are weaker in that they can only find errors on executed paths. Furthermore, the runtime overhead and difficulty in instrumenting low-level operating system code limits the applicability of dynamic approaches. The effectiveness of the dynamic approach is unclear because of the lack of experimental results.

Finally, some of the errors we found overlap with ones detected by other nonredundant checkers. For example, Splint [12] warns about fall-through case branches in switch statements even when they do not cause redundancies. It could have (but did not) issued a warning for the error shown in Fig. 4.[9] However, many, if not most, of the errors we find (particularly those in Sections 5 and 6) are not found by other tools and, thus, seem worth investigating. Unfortunately, we cannot do a more direct comparision to Splint because it lacks experimental data and we were unable to use it to compile the programs we check.

10 Conclusion

This paper explored the hypothesis that redundancies, like type errors, flag higher-level correctness mistakes. We evaluated the approach using five checkers which we applied to the Linux, OpenBSD, and PostgreSQL. These simple analyses found many surprising (to us) error types. Further, they correlated well with known hard errors: Redundancies seemed to flag confused or poor programmers who were prone to other errors. These indicators could be used to identify low-quality code in otherwise high-quality systems and help managers choose audit candidates in large code bases.

8. One sample error in the GCC analysis is reported at http://gcc.gnu.org/ml/gcc-prs/2003-03/msg01359.html.

9. Apparently, the parser was having trouble understanding the code and did not get far enough to check the problem part of the program.

Acknowledgments

This research was supported in part by DARPA contract MDA904-98-C-A933 and by a grant from the Stanford Networking Research Center. Dawson Engler is partially supported by a US National Science Foundation career award. The authors would like to thank Andy Chou, Godmar Back, Russel Greene, Ted Kremenek, and Ted Unangst for their help in writing this paper. They would also like to thank the editors and the anonymous reviewers for their helpful comments.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] A. Aiken, M. Fahndrich, and Z. Su, "Detecting Races in Relay Ladder Logic Programs," *Proc. First Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* Apr. 1998.

[3] K. Ashcraft and D.R. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," *Proc. IEEE Symp. Security and Privacy,* May 2002.

[4] T. Ball and S.K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *Proc. SPIN 2001 Workshop Model Checking of Software,* May 2001.

[5] W.R. Bush, J.D. Pincus, and D.J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software: Practice and Experience,* vol. 30, no. 7, pp. 775-802, June 2000.

[6] G. Casella and R.L. Berger, *Statistical Inference.* Pacific Grove, Calif.: Wadsworth Group, 2002.

[7] F.T. Chan and T.Y. Chen, "AIDA—A Dynamic Data Flow Anomaly Detection System for Pascal Programs," *Software: Practice and Experience,* vol. 17, no. 3, pp. 227-239, Mar. 1987.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D.R. Engler, "An Empirical Study of Operating Systems Errors," *Proc. 18th ACM Symp. Operating Systems Principles,* Oct. 2001.

[9] R. DeLine and M. Fahndrich, "Enforcing High-Level Protocols in Low-Level Software," *Proc. ACM SIGPLAN 2001 Conf. Programming Language Design and Implementation,* June 2001.

[10] D.R. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," *Proc. 19th ACM Symp. Operating Systems Principles,* Oct. 2003.

[11] D.R. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. 18th ACM Symp. Operating Systems Principles,* pp. 57-72, Oct. 2001.

[12] D. Evans, J. Guttag, J. Horning, and Y.M. Tan, "LCLint: A Tool for Using Specifications to Check Code," *Proc. Second ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 87-96, Dec. 1994.

[13] C. Flanagan, M.R.K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata, "Extended Static Checking for Java," *Proc. SIGPLAN '02 Conf. Programming Language Design and Implementation,* pp. 234-245, June 2002.

[14] D. Freedman, R. Pisani, and R. Purves, *Statistics,* third ed. W.W. Norton & Co., Sept. 1997.

[15] S. Hallem, B. Chelf, Y. Xie, and D.R. Engler, "A System and Language for Building System-Specific, Static Analyses," *Proc. ACM SIGPLAN 2002 Conf. Programming Language Design and Implementation,* pp. 69-82, June 2002.

[16] J.C. Huang, "Detection of Data Flow Anomaly through Program Instrumentation," *IEEE Trans. Software Eng.,* vol. 5, no. 3, pp. 226-236, May 1979.

[17] K. Kennedy, *Program Flow Analysis: Theory and Applications.* S. Muchnick and N. Jones, eds., pp. 5-54, Prentice-Hall, 1981.

[18] J. Knoop, O. Rüthing, and B. Steffen, "Lazy Code Motion," *Proc. SIGPLAN '92 Conf. Programming Language Design and Implementation,* pp. 224-234, June 1992.

[19] J. Knoop, O. Rüthing, and B. Steffen, "Partial Dead Code Elimination," *Proc. SIGPLAN '94 Conf. Programming Language Design and Implementation,* pp. 147-158, June 1994.

[20] E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Comm. ACM,* vol. 22, no. 2, pp. 96-103, Feb. 1979.

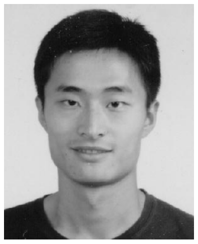
[21] E. Morel and C. Renvoise, *Program Flow Analysis: Theory and Applications.* S. Muchnick and N. Jones, eds., pp. 160-188, Prentice-Hall, 1981.

[22] L.J. Osterweil and L.D. Fosdick, "DAVE—A Validation Error Detection and Documentation System for Fortran Programs," *Software: Practice and Experience,* vol. 6, no. 4, pp. 473-486, Dec. 1976.

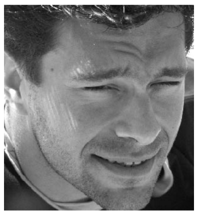
[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programming," *ACM Trans. Computer Systems,* vol. 15, no. 4, pp. 391-411, Nov. 1997.

[24] N. Sterling, "WARLOCK—A Static Data Race Analysis Tool," *Proc. USENIX Winter Technical Conf.,* pp. 97-106, Jan. 1993.

[25] D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. 2000 Network and Distributed Systems Security Conf.,* Feb. 2000.



Yichen Xie is a PhD candidate in the Computer Science Department at Stanford. He received the BS degree in computer science from Yale in 2001, and the MS degree in computer science from Stanford in 2003. His current research interests include static program analysis and software model checking.



Dawson Engler received the PhD degree from the Massachusetts Institute Technology and his undergraduate degree from the University of Arizona. He is an assistant professor and Terman fellow at Stanford. His past work has ranged from extensible operating systems to dynamic code generation. His current research focuses on developing techniques to find as many interesting software errors as possible.