Using Redundancies to Find Errors

Yichen Xie and Dawson Engler
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

ABSTRACT

This paper explores the idea that redundant operations, like type errors, commonly flag correctness errors. We experimentally test this idea by writing and applying four redundancy checkers to the Linux operating system, finding many errors. We then use these errors to demonstrate that redundancies, even when harmless, strongly correlate with the presence of traditional hard errors (e.g., null pointer dereferences, unreleased locks). Finally we show that how flagging redundant operations gives a way to make specifications "fail stop" by detecting dangerous omissions.

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 di erent 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 perform an action, it was because they believed it served some purpose. Spurious operations violate this belief and are likely errors. For example, impossible Boolean conditions can signal mistaken expressions; critical sections without shared state can signal the use of the wrong variable; variables written but not read can signal an unintentionally lost result. At the least, these conditions signal conceptual confusion, which we would also expect to correlate with hard errors — deadlocks, null pointer dereferences, etc. — even for harmless redundancies.

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

We wrote four checkers that flagged potentially dangerous redundancies: (1) idempotent operations, (2) assignments that were never read, (3) dead code, and (4) conditional branches that were never taken. The errors found would largely be missed by traditional type systems and checkers. For example, as Section 2 shows, assignment of variables to themselves can signal mistakes, yet such assignments will type check in any 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 e ectively find errors, our checkers must separate such redundancies from those induced by error.

We wrote our redundancy checkers in the *xgcc* extensible compiler system [16], 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 e ective flagging redundant operations is at finding dangerous errors by applying the above four checkers to the Linux operating system. This is a good test since Linux is a large, widely-used source code base (we check roughly 1.6 million lines of it). As such, it serves as a known experimental base. Also, because it has been written by many people, it is representative of many di erent 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 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 measuring how well redundancies predict these errors compared to chance. In our tests, files that have redundancy errors are roughly 45% to 100% more likely to have hard errors compared to files drawn by chance. This di erence holds across the di erent 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

SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

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 correctness errors.

2. Experimentally validating this idea by writing and applying four 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 give a way to make specifications "fail stop" by detecting dangerous 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 errors that we were reasonably sure about. We have had close to two years of experience with Linux bugs, so we have reasonable confidence that our false positive rate of bugs that we diagnose, while non-zero, is probably less than 5%.

Section 2 through Section 5 present our four checkers. Section 6 correlates the errors they found with traditional hard errors. Section 7 discusses how to check for completeness using redundancies. Section 8 discusses related work. Finally, Section 9 concludes.

2. IDEMPOTENT OPERATIONS

System	Bugs	Minor	False
Linux 2.4.5-ac8	7	6	3

Table 1: Bugs found by the idempotent checker in Linux version 2.4.5-ac8.

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`). 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. Four of these were apparent typos in variable assignments. The clearest example was the following code, where the programmer makes a mistake while copying structure `sa` to structure `da`:

```
/* 2.4.1/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 redundant errors catch cases that type systems miss. This code — an assignment of a variable to itself — will type check in all languages we know of, yet clearly contains an error. Two of the other errors were caused by integer overflow (or'ing an 8-bit variable by a constant that only had bits set in the upper 16 bits). The final one 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 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 non-zero additions.

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

Macros are the main source of potential false positives. They represent logical actions that may not map to a concrete action. 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 errors on these lines.

3. REDUNDANT ASSIGNMENTS

System	Bugs	False	Uninspected
Linux 2.4.5-ac8	129	26	1840
xgcc	13	1	0

Table 2: Bugs found by the redundant assignment checker in Linux version 2.4.5-ac8 and the *xgcc* system used in this paper. There were 1840 uninspected errors for variables assigned but never used in Linux — we expect a large number of these will be actual errors given the low number of false positives in our inspected results.

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 global analysis. At each assignment it follows the variable forward on all paths. It emits an error message if the variable is read on no path before either exiting scope or being assigned another value. As we show, in many cases such lost values signal real errors, where control flow followed unexpected paths, results that were computed were not returned, etc.

The checker finds thousands of redundant assignments in a system the size of Linux. Since it was so e ective, we minimized the chance of false positives by radically restricting the variables it would follow to non-global variables that were not aliased in any way.

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

1. Variables assigned values that are not read. Empirically, these errors tend to be the most serious, since they flag unintentionally lost results.

2. Variables assigned a non-constant that is then overwritten without being read. 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.

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 use. We track the value and emit it when reporting the error so that messages using a common defensive value can be easily suppressed.

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 produced by macros. Its main remaining vulnerability are to values assigned and then passed to debugging macros that are turned o :

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

Typically there are a small number of such macros, which we manually turn back on.

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. Thus we rank errors based on (1) the line distance between the assignment and reassignment and (2) the number of conditions on the path. Close errors are most likely; farther errors become more arguably defensive programming.

The errors. This checker found more errors than all the other checkers 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).

Figure 1 shows a representative example of the first pattern. 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 calling application 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 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. In both of these cases, the fact that logically the code contains errors is readily flagged by looking for variables assigned but not used.

The second class of errors comes from calculations that are aborted by unexpected control flow. Figure 2 gives one example: 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. Figure 3 gives 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 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. The presence of these errors led us to write the dead-code checker in the next section.

```
/* 2.4.1/net/decnet/af_decnet.c:dn_wait_run */
do {
  ...
  if (signal_pending(current)) {
     err = -ERESTARTSYS; /* BUG: lost value */
     break;
  }
  SOCK_SLEEP_PRE(sk)
  if (scp->state != DN_RUN)
       schedule();
  SOCK_SLEEP_POST(sk)
} while(scp->state != DN_RUN);
return 0;
```

Figure 1: Lost return value caught by flagging the redundant assignment to `err`.

```
/* 2.4.1/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;
}
```

Figure 2: A single-iteration loop caught by flagging the redundant assignment `next = entry→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.

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

```
/* 2.4.5-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 di erent case shows a potential confusion about how C's iteration works:

```
/* 2.4.1/drivers/scsi/scsi.c: */
SCnext = SCpnt->bh_next;
for (; SCpnt; SCpnt = SCnext) {
    SCnext = SCpnt->bh_next;
```

Where 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:

```
/* 2.4.5-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;
   ...
```

Figure 3: Catastrophic return caught by the redundant assignment to c2. The 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`.

```
/* 2.4.1/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);
  ...
```

Figure 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.

System	Bugs	False
Linux 2.4.5-ac8	66	26

Table 3: Bugs found by the dead code checker on Linux version 2.4.5-ac8.

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

At first glance this seems like an obvious copy-and-paste error. It turned out that the redundancy flags a much more devious error. The array `buf` actually pointed to a "memory mapped" region of kernel memory. Unlike normal memory, reads and writes to this memory cause the CPU to issue I/O commands to a hardware device. Thus, the reads are not idempotent, and the two of them in a row rather than just one can cause very di erent 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 memory mapped IO 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 later version gcc could have caused this extremely di cult to track down bug to surface.

4. DEAD CODE

The checker in this section flags dead code. Since programmers generally write code to run it, dead code catches logical errors signaled by false beliefs that an impossible path 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. It 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 — code dominated by such calls cannot run. Second, we suppress error messages for dead code caused by constant conditions, such as

```
if(0)
    printf("in foo");
```

since these frequently signaled code "commented out" by using a false condition. We also annotate error messages when the code they flag is a single statement that contains a break or return. These are commonly a result of defensive programming. Finally, we suppress dead code caused by 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 extension: (1) the single iteration loop in Figure 2, (2) the mistaken statement terminator in Figure 3, and (3) the unintentional fall through in Figure 4.

Figure 5 gives the most frequent copy-and-paste error. Here the macro "`pseterr`" returns, but the caller does not realize it. Thus, at all seven call sites that use the macro, there is dead code after the macro that the client intended to have executed.

```
/* 2.4.1/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;
}
```

Figure 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 mistaken uses of the same macro.

Figure 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 paths. It is hard to believe that this code was ever tested. Figure 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 at the end of the body can be executed.

```
/* 2.4.1/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;
     list = c->cmd;
     if (free) {
         c->next = hostdata->free;
         hostdata->free = c;
     }
```

Figure 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 flags redundant branch conditionals from: (1) branch statements (`if`, `while`, `for`, etc) with non-constant conditionals that always evaluate to either *true* or *false*; (2) `switch` statements with impossible `case`'s. Both cases are a result of logical inconsistency in the program and are therefore likely to be errors.

```
/* 2.4.5-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;
```

Figure 7: Useless loop body: similarly to Figure 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.

```
/* 2.4.1/drivers/net/arcnet/arc-rimi.c:
                             arcrimi_found */
/* reserve the irq */  {
     if (request_irq(dev->irq, &arcnet_interrupt ...))
                   BUGMSG(D_NORMAL,
                   "Can't get IRQ %d!\n", dev->irq);
     return -ENODEV;
}
```

Figure 8: Unexpected return: misplaced braces from the insertion of a debugging statement causes control to always return.

The checker is based on the false-path pruning (FPP) feature in the *xgcc* system. FPP was originally designed to prune away false positives arising from infeasible paths. It symbolically evaluates variable assignments and comparisons, either to constants (e.g. `x = 10, x < 100`) or to other variables (e.g. `y = x, x < y`), using a simple congruence closure algorithm [11]. It will stop the checker from checking the current execution path as soon as it detects a logical conflict.

With FPP, the checker is implemented using a simple mark-and-sweep algorithm. For each routine, it explores all feasible execution paths and marks branches (as opposed to basic blocks in Section 4) visited along the way. Then it takes the set of unmarked branches and flags conditionals associated with them as redundant.

The checker was able to find hundreds of redundant conditionals in Linux 2.4.1. The main source of false positives arises from the following two forms of macros: (1) those with embedded conditionals, and (2) constant macros that are used in conditional statements (e.g. "`if (DEBUG) {...}`," where `DEBUG` is defined to be 0). After suppressing those, we are left with three major classes of about 200 problematic cases, which we describe below.

The first class of errors is the least serious of the three that we characterize as "overly cautious programming style."

This includes cases where the programmer checks the same condition multiple times within very short program distances. We believe this could be an indication of a novice programmer and the conjecture is supported by the statistical analysis described in section 6.

Figure 9 shows a redundant check of the above type from Linux 2.4.1. Although it is almost certainly harmless, it shows the programmer has a poor grasp of the code. One might be willing to bet on the presence of a few surrounding bugs.

```
/* 2.4.1/drivers/media/video/cpia.c:cpia_mmap */
if (!cam || !cam->ops)
    return -ENODEV;
/* make this _really_ smp-safe */
if (down_interruptible(&cam->busy_lock))
    return -EINTR;
if (!cam || !cam->ops) /* REDUNDANT! */
    return -ENODEV;
```

Figure 9: Overly cautious programming style: the second check of `(!cam || !cam->ops)` is redundant.

Figure 10 shows a more problematic case. As one can see, the `else` branch of the second `if` statement will never be taken, because the first `if` condition is weaker than the negation of the second. Interestingly, the function returns di erent error codes for essentially the same error, indicating a possibly confused programmer.

```
/* 2.4.1/drivers/net/wan/sbni.c:sbni_ioctl */
slave = dev_get_by_name(tmpstr);
if(!(slave && slave->flags & IFF_UP &&
     dev->flags & IFF_UP))
{
  ... /* print some error message, back out */
  return -EINVAL;
}
if (slave) {  ... }
/* BUG: !slave is impossible */
else {
  ... /* print some error message */
  return -ENOENT;
}
```

Figure 10: Overly cautious programming style. The check of `slave` is guaranteed to be true and also notice the di erence in return value.

The second class of errors we catch are again seemingly harmless, but when we examine them carefully, we find serious errors around them. With some guesswork and cross-referencing, we assume the `while` loop in Figure 11 is trying to recover from hardware errors encountered when reading a network packet. But since the variable `err` is never updated in the loop body, the condition `(err != SUCCESS)` is always true and the loop body is never executed more than once, which is nonsensical. This could signal a possible bug where the author forgets to update `err` in the large chunk of recovery code in the loop. This bug, if confirmed, could be di cult to detect dynamically, because it is in the error recovery code that is easy to miss in testing.

The third class of errors are clearly serious bugs. Figure 12 shows an example detected by the redundant conditional checker. As one can see, the second and third `if` statements carry out entirely di erent actions on identical conditions. Apparently, the programmer cut-and-pasted the conditional without changing one of the two `NODE_LOGGED_OUT` into a fourth possibility: `NODE_NOT_PRESENT`.

```
/* 2.4.1/drivers/net/tokenring/smctr.c:
                             smctr_rx_frame */
while((status = tp->rx_fcb_curr[queue]
                  ->frame_status)
        != SUCCESS)
{
    err = HARDWARE_FAILED;
    ... /* large chunk of apparent recovery code,
           with no updates to err */
    if (err != SUCCESS)
        break;
}
```

Figure 11: Redundant conditional that suggests a serious program error.

```
/* 2.4.1/drivers/fc/iph5526.c:
                  rscn_handler */
if ((login_state == NODE_LOGGED_IN) ||
    (login_state == NODE_PROCESS_LOGGED_IN)) {
    ...
}
else
if (login_state == NODE_LOGGED_OUT)
    tx_adisc(fi, ELS_ADISC, node_id,
             OX_ID_FIRST_SEQUENCE);
else
/* BUG: redundant conditional */
if (login_state == NODE_LOGGED_OUT)
    tx_logi(fi, ELS_PLOGI, node_id);
```

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

Figure 13 shows another serious error. One can see that the author intended to insert an element pointed to by `sp` into a doubly-linked list with head `q->q_first`, but the `while` loop really does nothing other than setting `srb_p` to `NULL`, which is nonsensical. The checker detects this error by inferring that the exit condition for the `while` loop conflicts with the true branch of the ensuing `if` statement. The obvious 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 the only element in it. As a matter of fact, the same bug is still present in the latest 2.4.19 release of the Linux kernel source as of this writing.

6. 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 come from the previous four sections. The hard

```
/* 2.4.1/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;
}
```

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

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). 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% more likely to have a hard error than a file 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.

6.1 Methodology

This subsection describes the statistical methods used to measure the association between program redundancies and hard errors. Our analysis is based the 2×2 contingency table [6] method. It is a standard statistical tool for studying the association between two di erent attributes of a population. In our case, the population is the set of files we have checked, and the two attributes are: (a) whether a file contains redundancies, and (b) whether it contains hard errors.

In the contingency table approach, the sample 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 tabularize the result as follows:

	B		
A	True	False	Totals
True	o_{11}	o_{12}	$n_{1\cdot}$
False	o_{21}	o_{22}	$n_{2\cdot}$
Totals	$n_{\cdot 1}$	$n_{\cdot 2}$	$n_{\cdot\cdot}$

The values in the margin ($n_{1\cdot}, n_{2\cdot}, n_{\cdot 1}, n_{\cdot 2}$) are row and column totals, while $n_{\cdot\cdot}$ is the grand total. The null hypothesis H_0 of this test is that the **A** and **B** are mutually independent, i.e. knowing **A** does not give us any additional information about **B**. More precisely, if H_0 holds, we are expecting that:

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

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 [15]:

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

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, which is called the p-value [2]. The smaller the p-value, the stronger the evidence against H_0, thus the stronger the correlation between attributes **A** and **B**.

6.2 Data acquisition and test results

In our previous work [8], we used the *xgcc* system to check 2055 files in Linux 2.4.1 kernel. We had focused on serious system crashing hard bug and were able to collect more than 1800 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 6.1 for the following three redundancy checkers: redundant assignment checker, dead code checker, and redundant conditional checker (the idempotent operation is excluded because of its small sample size).

The result of the tests are given in Tables 4, 5, 6, and 7. As we can see, the correlation between redundancies and hard

[1] 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% of the 40 balls with red marks to have blue marks, as should we expect roughly 80% of the remaining 60 balls without the red mark to have a blue mark.

[2] Technically, under H_0, T has a χ^2 distribution with one degree of freedom. 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%.

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

$T = 194.37$, p-value = 0.00

Table 4: Contingency table: Redundant Assignments vs. Hard Bugs. There are 345 files with both error types, 435 files with an assign error and no hard bugs, 206 files with a hard bug and no assignment error, and 1069 files with no bugs of either type. A T-statistic value above four gives a p-value of less than .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 this standard threshold. Intuitively, the correlation between error types can be seen in that the ratio of 345/435 is considerably larger than the ratio 206/1069 — if the events were independent, we expect these two ratios to be close.

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

$T = 81.74$, p-value = 0.00

Table 5: Contingency table: Dead code vs. Hard Bugs

errors are extremely high, with p-values being approximately 0 in all four cases. It strongly suggests that redundancies often signal confused programmers, and therefore are a good predictor for hard, serious errors.

6.3 Predicting hard errors

In addition to correlation, we want to know how much more likely it is that we will find a hard error in a file that has 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 contains one or more forms of redundant operations, we can compute a confidence interval for $T' = (P(E|R) - P(E))/P(E)$, which is a measure of how much more likely we are to find hard errors in a file given program redundancies.

The prior probability of hard errors is computed as follows:

$$\begin{aligned} P(E) &= \frac{\text{Number of files with hard errors}}{\text{Total number of files checked}} \\ &= 551/2055 = 0.2681 \end{aligned}$$

We tabularize the conditional probabilities and T' values in Table 8. (Again, we excluded the idempotent operation checker because of its small bug sample.) As shown in table, given any form of redundant operation, it is roughly 45% – 100% more likely we will find an error in that file than otherwise. Furthermore, redundancies even predict hard errors across time: we carried out the same test between redundancies found in Linux 2.4.5-ac8 and hard errors in 2.4.1 (roughly a year older) and found similar results.

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

$T = 40.65$, p-value = 0.00

Table 6: Contingency table: Redundant Conditionals vs. Hard Bugs

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

$T = 140.48$, p-value = 0.00

Table 7: Contingency table: Program Redundancies (Aggregate) vs. Hard Bugs

7. FAIL-STOP SPECIFICATION

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 a checker that uses redundancy to find when it is missing checkable actions.

Detecting omitted `null` annotations. Tools such as LCLint [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 redundancy. We know only the return value of `null` functions should be checked. Thus, a check on a non-annotated 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.

Finding missed lock-variable bindings. Data race detection tools such as Warlock [20] let users explicitly bind locks

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%

Table 8: Program files with redundancies are roughly 50% more likely to contain hard errors

to the variables they protect. The tool flags 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 redundancy 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 good 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 errors. As shown, such omissions can be detected by flagging redundant operations (reads or writes) that do not make sense for non-volatile 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 redundancy to find such omissions: assuming programmers do not do redundant permission checks, then finding permission check that does not guard a known sensitive operation signals an incomplete specification.

7.1 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, memory copy lengths) 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 flag when unsanitized data reaches the sink.

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, then 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. 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 inter-procedural analysis had been overly simplistic, causing us to miss the path to it. The fact that redundancy flags errors both in the specification and in the tool itself was a nice surprise.

8. RELATED WORK

Two existing types of analysis have focused on redundant operations: optimizing compilers and "anomoly detection" work.

Optimizing compilers commonly do dead-code elimination and common-subexpression elimination [1] which remove redundancies to improve performance. One contribution of our work is the realization that these analyses have been silently finding errors since their invention. While our analyses are closely mirror these algorithms at their core, they have several refinements. First, we operate on a higher-level representation than a typical optimizer since a large number of redundant operations are introduced due to the compilation of source constructs to the intermediate representation. Second, in order to preserve semantics of the program, compiler optimizers have to be conservative in its analysis. In contrast, since our goal is to find *possible* errors, it is perfectly reasonable to flag a redundancy even if we are only 95% sure about its legitimacy. In fact, we report all suspicious cases and sort in order of a confidence heuristic (e.g. distance between redundancies, etc) in the report. Finally, the analysis tradeo s we make di er. For example, we use a path-sensitive algorithm to suppress false paths; most optimizers omit path-sensitive analyses because their time complexity outweighs their benefit.

The second type of redundant analysis includes checking tools. Fosdick and Osterweil first applied data flow "anomaly detection" techniques in the context of software reliability. In their DAVE system [18], they used a depth first search algorithm to detect a fixed set of variable def-use type of anomalies such as uninitialized read, double definition, etc. Static approaches like this [13, 14, 18] are often path-insensitive, and therefore could report bogus errors from infeasible paths.

Dynamic techniques [17, 7] instruments the program and detect anomalies that arise during execution. However, dynamic approaches are weaker in that they can only find errors on executed paths. Further the run-time overhead and di culty in instrumenting operating systems limits the usage of this approach.

The dynamic system most similar to our work is Huang [17]. He discusses a checker similar to the assignment checker in Section 3. It tracks the lifetime of variables using a simple global analysis. At each assignment it follows the variable forward on all paths. It gives an error if the variable is read on no path before either exiting scope or being assigned another value. However, no experimental results were given. Further, because it is dynamic it seems predisposed to report large numbers of false positives in the case where a value is not read on the current executed path but would be used on some other (non-executed) path.

Other tools such as `lint`, LCLint [12], or the GNU C compiler's `-Wall` option warn about unused variables and routines and ignored return values. While these have long found redundancies in real code (we use them ourselves daily), these redundancies have been commonly viewed as harmless stylistic issues. 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 [4, 10, 5, 9, 2, 19, 21]. A main contribution of this paper is showing that redundancies signal real errors and experimentally measuring how well this holds.

9. CONCLUSION

This paper explored the hypothesis that redundancies, like type errors, flag higher-level correctness mistakes. We evaluated the approach using four checkers which we applied to the Linux operating system. 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 error types. These indicators could be used to decide where to audit a system.

10. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful comments. This work was supported by NFS award 0086160 and by DARPA contract MDA904-98-C-A933.

11. REFERENCES

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

[2] A. Aiken, M. Faehndrich, and Z. Su. Detecting races in relay ladder logic programs. In *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 1998.

[3] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *To appear in IEEE Symposium on Security and Privacy 2002*, 2002.

[4] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.

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

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

[7] F. T. Chan and T. Y. Chen. Aida–a dynamic data flow anomaly detection system for pascal programs. *Software: Practice and Experience*, 17(3):227–239, March 1987.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[9] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.

[10] D. L. Detlefs. An overview of the extended static checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996.

[11] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[12] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[13] I. R. Forman. An algebra for data flow anomaly detection. In *Proceedings of the 7th international conference on Software engineering*, pages 278–286, 1984.

[14] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305 – 330, September 1976.

[15] D. Freedman, R. Pisani, and R. Purves. *Statistics*. W.W. Norton, third edition edition, 1998.

[16] S. Hallem, B. Chelf, Y. Xie, and D.R. Engler. A system and language for building system-specific, static analyses. To appear in PLDI 2002.

[17] J. C. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, 5(3):226–236, May 1979.

[18] L. J. Osterweil and L. D. Fosdick. Dave–a validation error detection and documentation system for fortran programs. *Software: Practice and Experience*, 6(4):473–486, December 1976.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[20] N. Sterling. WARLOCK - a static data race analysis tool. In *USENIX Winter*, pages 97–106, 1993.

[21] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of bu er overrun vulnerabilities. In *The 2000 Network and Distributed Systems Security Conference. San Diego, CA*, February 2000.