Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

Abstract

A major obstacle to finding program errors in a real system is knowing what correctness rules the system must obey. These rules are often undocumented or specified in an ad hoc manner. This paper demonstrates techniques that automatically extract such checking information from the source code itself, rather than the programmer, thereby avoiding the need for a priori knowledge of system rules.

The cornerstone of our approach is inferring programmer "beliefs" that we then cross-check for contradictions. Beliefs are facts implied by code: a dereference of a pointer, `p`, implies a belief that `p` is non-null, a call to "`unlock(l)`" implies that `l` was locked, etc. For beliefs we know the programmer must hold, such as the pointer dereference above, we immediately flag contradictions as errors. For beliefs that the programmer may hold, we can assume these beliefs hold and use a statistical analysis to rank the resulting errors from most to least likely. For example, a call to "`spin_lock`" followed once by a call to "`spin_unlock`" implies that the programmer may have paired these calls by coincidence. If the pairing happens 999 out of 1000 times, though, then it is probably a valid belief and the sole deviation a probable error. The key feature of this approach is that it requires no a priori knowledge of truth: *if two beliefs contradict, we know that one is an error without knowing what the correct belief is.*

Conceptually, our checkers extract beliefs by tailoring rule "templates" to a system – for example, finding all functions that fit the rule template "<*a*> must be paired with <*b*>." We have developed six checkers that follow this conceptual framework. They find hundreds of bugs in real systems such as Linux and OpenBSD. From our experience, they give a dramatic reduction in the manual effort needed to check a large system. Compared to our previous work [9], these template checkers find ten to one hundred times more rule instances and derive properties we found impractical to specify manually.

1 Introduction

We want to find as many serious bugs as possible. In our experience, the biggest obstacle to finding bugs is not the need for sophisticated techniques nor the lack of either bugs or correctness constraints. Simple techniques find many bugs and systems are filled with both rules and errors. Instead, the biggest obstacle to finding many bugs is simply knowing what rules to check. Manually discovering any significant number of rules a system must obey is a dispiriting adventure, especially when it must be repeated for each new release of the system. In a large open source project such as Linux, most rules evolve from the uncoordinated effort of hundreds or thousands of developers. The end result is an ad hoc collection of conventions encoded in millions of lines of code with almost no documentation.

Since manually finding rules is difficult, we instead focus on techniques to automatically extract rules from source code without a priori knowledge of the system. We want to find what is incorrect without knowing what is correct. This problem has two well-known solutions: contradictions and common behavior. How can we detect a lie? We can cross-check statements from many witnesses. If two contradict, we know at least one is wrong *without knowing the truth*. Similarly, how can we divine accepted behavior? We can look at examples. If one person acts in a given way, it may be correct behavior or it may be a coincidence. If thousands of people all do the same action, we know the majority is probably right, and any contradictory action is probably wrong *without knowing the correct behavior*.

Our approach collects sets of programmer *beliefs*, which are then checked for contradictions. Beliefs are facts about the system implied by the code. We examine two types of beliefs: MUST beliefs and MAY beliefs. MUST beliefs are directly implied by the code, and there is no doubt that the programmer has that belief. A pointer dereference implies that a programmer must believe the pointer is non-null (assuming they want safe code). MAY beliefs are cases where we observe code features that suggest a belief but may instead be a coincidence. A call to "`a`" followed by a call to "`b`" implies the programmer may believe they must be paired, but it could be a coincidence.

Once we have a set of beliefs, we do two things. For a set of MUST beliefs, we look for contradictions. Any contradiction implies the existence of an error in the code. For a set including MAY beliefs, we must separate valid beliefs from coincidences. We start by assuming all MAY beliefs are MUST beliefs and look for violations (errors) of these beliefs. We then use a statistical analysis to rank each error by the probability of its beliefs. If a particular belief is observed in 999 out of 1000 cases, then it is probably a valid belief. If the belief happens only once, it is probably a coincidence.

We apply the above approach by combining it with our prior work [9]. That work used system-specific static analyses to find errors with a fixed set of manually found and specified rules (e.g.,"`spin_lock(l)` must be paired with `spin_unlock(l)`"). It leveraged the fact that abstract rules commonly map to fairly simple source code sequences. For example, one can check the rule above by inspecting each path after a call to "`spin_lock(l)`" to ensure that the path contains a call to "`spin_unlock(l)`." While effective, this previous work was limited by the need to find rules manually. This paper describes how to derive rule instances automatically: our system infers the pairing rule above directly from the source code.

Experience indicates that this approach is far better than the alternative of manual, text-based search to find relevant rule instances. The analyses in this paper automatically derive all the rule instances previously hand-specified in [9], as well as an additional factor of ten to one hundred more. Further, we now check properties that we formerly gave up on (see Section 7). We demonstrate that the approach works well on complex, real code by using it to find hundreds of errors in the Linux and OpenBSD operating systems. Many of our bugs have resulted in kernel patches.

Section 2 discusses related work. Sections 3–5 give an overview of the approach, and Sections 6–9 apply it to find errors. Section 10 concludes.

2 Related Work

There are many methods for finding errors. The most widely used, testing and manual inspection, suffer from the exponential number of code paths in real systems and the erratic nature of human judgment. Below, we compare our approach to other methods of finding errors in software: type systems, specification-based checking, and high-level compilation. We close by comparing our work with two systems that dynamically infer invariants.

Type systems. Language type systems probably find more bugs on a daily basis than any other approach. However, many program restrictions—especially temporal or context-dependent restrictions—are too rich for an underlying type system or are simply not expressed in it. While there has been some work on richer frameworks such as TypeState [23], Vault [6], and aspect-oriented programming [17], these still miss many systems relations and require programmer participation. Further, from a tool perspective, all language approaches require invasive, strenuous rewrites to get results. In contrast, our approach transparently infers richer, system-specific invariants without requiring the use of a specific language or ideology for code construction.

Traditional type systems require programmers to lace a fixed type system throughout their code. We take the opposite approach of inferring an ad hoc type system implicit in programs and then putting this into the compiler. As a side-effect, we show that code features believed to require specification can be pulled from the source directly (see Section 7).

Specifications. Another approach is to specify code and then check this specification for errors. An extreme example of this approach is formal verification. It gains richness by allowing the programmer to express invariants in a general specification, which is then checked using a model checker [19, 22], theorem provers, or checkers [13, 20]. While formal verification can find deep errors, it is so difficult and costly that it is rarely used for software. Further, specifications do not necessarily mirror the code they abstract and suffer from missing features and over-simplifications in practice. While recent work has begun attacking these problems [5, 15, 18], verifying software is still extremely rare. The SLAM project [2] is a promising variation on this approach. It extracts and then model checks a Boolean variable program skeleton from C code. However, it requires considerably more effort than our approach, and appears to check a more limited set of properties.

Recent work has developed less heavyweight checkers, notably the extended static type checking (ESC) project [7], which checks interface-level specifications and LCLint [11], which statically checks programmer source annotations. However, these approaches still require more effort than those in this paper. The specifications required by these approaches scale with code size. In contrast, our analyses cost a fixed amount to construct but then repay this cost by automatically extracting checking information from large input codes. In a sense, our work is complementary to these other approaches, since the information extracted by our analyses can be used to check that specifications correctly describe code.

The Houdini assistant to ESC [12] is one effort to decrease the manual labor of annotation-based approaches. Houdini uses annotation templates to automatically derive ESC annotations, then uses those annotations to statically find runtime errors in Java programs. One difference between our approach and theirs is that we allow for much noisier samples when deriving our rule templates, then we use statistical analysis to rank the derived rules.

High-level Compilation. Many projects have embedded hard-wired application-level information in compilers to find errors [1, 3, 4, 8, 21, 24]. These projects find a fixed set of errors, whereas we derive new checks from the source itself, allowing detection of a broader range of errors. The checking information we extract could serve as inputs to suitably modified versions of these other checkers.

Dynamic invariant inference. The two most significant projects in this area are Daikon and Eraser. Daikon is the most similar project to ours in terms of deriving program rules [10]. Daikon dynamically monitors program execution to reconstruct simple algebraic invariants. It starts with a set of mostly linear building block hypotheses (that a variable is a constant, that it is always less than or greater than another variable) and validates each hypothesis against each execution trace. If a trace violates a hypothesis, the hypothesis is discarded. Compared to static analysis, dynamic monitoring has the advantage that noise and undecidability is less of a concern: by definition, an executed path is possible, and at runtime, all values can be determined. However, the accuracy of dynamic monitoring has a cost. Daikon is primarily intended to help understand programs. It has found very few errors and would have several significant difficulties in doing so: it can only see executed paths, requires test cases that adequately exercise the code it monitors, and can only observe how code works in the tested environment. Static analysis does not have any of these problems.

In terms of desire to find bugs, the Eraser system is most similar to our work [21]. Eraser dynamically detects data races by monitoring which locks protect which variables. Inconsistent locking is flagged as an error. Eraser has been effective at finding real bugs [21]. However, because it is dynamic it has similar limitations to Daikon: it only sees a limited number of paths, requires the ability to monitor code, and can only flag an error when a path is executed.

Of course, dynamic information can be quite useful. In future work we intend to explore how static analysis can be augmented with dynamic monitoring. One possibility is using profile data to rank bugs.

3 Methodology

This section introduces our approach and terminology for finding bugs. The goal of our approach is to extract beliefs from code and to check for violated beliefs.

We restrict our attention to beliefs that fit generic rule *templates*. An example template is "$<a>$ must be paired with $<b>$." In this example, the bracketed letters $<a>$ and $<b>$ represent positions in our template that the extraction process should fill with concrete elements from the code. We call these positions *slots* and code elements that fill slots *slot instances*. Possible slot instances for slots $<a>$ and $<b>$ could be the function calls `lock` and `unlock` respectively.

The remainder of this section explains how we apply a template to a new, unknown system and end up with hundreds of automatically-detected bugs. We begin with a detailed example of a null-pointer-use checker. This example introduces a general approach that we call *internal consistency*. We then present a detailed description of a locking-discipline checker, which introduces an approach we call *statistical analysis*. We conclude by describing the system we use to implement our checkers and the systems that we check.

3.1 Example: null pointer consistency

This subsection illustrates how internal consistency can find errors by applying it to one of the simplest possible problems: detecting null-pointer uses statically. Consider the following code fragment, which compares the pointer `card` against null and then dereferences it:

```
    /* 2.4.1:drivers/isdn/avmb1/capidrv.c: */
1: if (card == NULL) {
2:      printk(KERN_ERR "capidrv-%d: ... %d!\n",
3:                card->contrnr, id);
4: }
```

At line 1, the check `card == NULL` implies the belief that `card` is null on the true path of the conditional. However, at line 3 the dereference `card→contrnr` implies the belief that `card` is not null: a contradiction. A consistency checker can find such errors by associating every pointer, `p`, with a *belief set* and flagging cases where beliefs contradict. For our example, `p`'s belief set could contain nothing (nothing is known about `p`), "null" (`p` is definitely null), "not null" (`p` is definitely not null), or both "null" and "not null" (`p` could be either). Any code element, or *action*, implying a belief that contradicts `p`'s current belief set is an error.

Note that, while not relevant for the error above, the comparison action at line 1 also implies that `p`'s belief set should contain both "null" and "not null" before line 1. Otherwise, this check is pointless. This implied belief set is useful in a different piece of code:

```
   /* 2.4.7:drivers/char/mxser.c */
1:int mxser_write(struct tty_struct *tty, ...) {
2:  struct mxser_struct *info = tty->driver_data;
3:  unsigned long flags;
4:
5:  if (!tty || !info->xmit_buf)
6:     return (0);
7:  ...
```

At line 2, `tty→driver_data` dereferences `tty`, but at line 5 the check `!tty` implies `tty` could be null. Either the check is impossible and should be deleted, or the code has a potential error and should be fixed. The following beliefs are inferred on each line:

Line 1: entry to `mxser_write`. Assuming we do not have inter-procedural information, the checker sets `tty`'s belief set to "unknown," otherwise we set it to its value at the caller.

Line 2: the checker sets `tty`'s belief set to "not null."

Lines 3 and 4 have no impact on the belief set. We say a belief set is *propagated* when it moves from one action to another. In this case, the belief set after line 2 is propagated forward through lines 3 and 4 to line 5.

Line 5: implies a belief set for `tty` containing both "null" and "not null." However, the only path to this condition has a belief set of "not null," which contradicts the implied belief set.

We formalize the framework for internal consistency checkers below.

3.2 General internal consistency

Consistency checkers are defined by five things:

1. The rule template T.
2. The valid slot instances for T.
3. The code actions that imply beliefs.
4. The rules for how beliefs combine, including the rules for contradictions.
5. The rules for belief propagation.

The rule template T determines what property the checker tests. The checker's job is to find and check valid slot instances for the template, T. For example, the null-pointer checker's template is "do not dereference null pointer $<p>$," and all pointers are potentially valid slot instances for $<p>$. Each slot instance has an associated belief set. At each action, we consider how that action effects the belief sets for each slot instance. For the checker above, if an action is a dereference of a pointer `p`, the action can either (1) signal an error if `p`'s belief set contains the belief "null," or (2) add the belief "not null" to `p`'s belief set. If an action implies a belief, we must also consider how that belief propagates to other actions. A comparison, `p == NULL`, propagates the belief that `p` is "null" to all subsequent actions on its true branch, the belief that `p` is "not null" to all subsequent actions on its false branch, and the belief that `p` could be either "null" or "not null" when these paths join. In general, beliefs can propagate forward, backward, from caller to callee, between functions that implement the same abstract interface either within the same program or across programs, or to any other piece of *related code*. We give a more complete discussion of related code in Section 4.2.

More formally, for any slot instance v, we denote its belief set as B_v. The null checker associates each pointer p with a belief set B_p that can take on the values $B_p = \{null\}$ (p is definitely null), $B_p = \{notnull\}$ (p is definitely not null), or $B_p = \{null, notnull\}$ (p could be either null or not null). By convention an empty belief set, $B_p = \emptyset$, means nothing is known about p.

Most actions have no impact on the current belief sets other than propagating them forward unaltered to the next statement. The null checker above had two actions that imply beliefs: dereferences and comparisons. These actions affect the belief set of one valid slot instance, i.e. the pointer, p, that is dereferenced or compared to null. A dereference of pointer p implies the belief *notnull* ($B_p = \{notnull\}$) and gives an error if p's belief set contains *null* ($null \in B_p$). Comparison implies two things. First, p's belief set prior to the comparison contains both null and notnull ($B_p = \{null, notnull\}$). An error is given if the beliefs are known more precisely (error if $B_p = \{null\}$ or $B_p = \{notnull\}$)). Second, after the conditional, p is null on the true path ($B_p = \{null\}$), and not null on the false ($B_p = \{notnull\}$).

One complication when propagating beliefs is what happens when different paths join. The null checker takes the union of all beliefs on the joining paths. For the first example, `card`'s belief set is $B_p = \{null\}$ on the true path after the comparison `card == NULL`, $B_p = \{notnull\}$ on the false path, but becomes $B_p = \{null, notnull\}$ when the paths join after line 4.

3.3 Example: statistical lock inference

This subsection illustrates how statistical analysis can find errors in sets of MAY beliefs. We use statistical analysis to rank MAY belief errors from most to least probable.

Consider the problem of detecting when a shared variable v is accessed without its associated lock l held. If we know which locks protect which variables, we can readily check this rule using static analysis. Unfortunately, most systems do not specify "lock <l> protects variable <v>." However, we can derive this specification from the code by seeing what variables are "usually" protected by locks. If v is almost always protected by l, it may be worth flagging cases where it is not.

Consider the contrived code example in Figure 1 with two shared variables, `a` and `b`, that may or may not be protected by a lock `l`. Here, `a` is used four

```
 1: lock l;           // Lock
 2: int a, b;         // Variables potentially
                      // protected by l
 3: void foo() {
 4:     lock(l);      // Enter critical section
 5:     a = a + b;    // MAY: a,b protected by l
 6:     unlock(l);    // Exit critical section
 7:     b = b + 1;    // MUST: b not protected by l
 8: }
 9: void bar() {
10:     lock(l);
11:     a = a + 1;    // MAY: a protected by l
12:     unlock(l);
13: }
14: void baz() {
15:     a = a + 1;    // MAY: a protected by l
16:     unlock(l);
17:     b = b - 1;    // MUST: b not protected by l
18:     a = a / 5;    // MUST: a not protected by l
19: }
```

Figure 1: A contrived, useful-only-for-illustration example of locks and variables

times, three times with `l` held, and once without `l` held. In contrast, `b` is indifferently protected with `l`: not protected twice, and protected once in a plausibly-coincidental situation. Intuitively, `a` is much more plausibly protected by `l` than `b`. This belief is further strengthened by the fact that `a` is the only variable accessed in the critical section at line 11 – either the acquisition of l is spurious (locks must protect *something*) and should be fixed, or the programmer believes `l` protects `a`.

Our checking problem reduces to inferring if code *believes* l protects v. If a use of variable v protected by lock l implied the MUST belief that l protects v, then we could check the rule above using internal consistency. However, the protected access could simply be a coincidence, since accessing unprotected variables in critical sections is harmless. Thus, we can only infer that code *may* believe l protects v. We call this type of belief a MAY belief.

How can we check MAY beliefs? In all cases we consider in this paper, the MAY belief reduces to whether or not candidate slot instances should be checked with a rule T. For the example above, should a given variable `a` and lock `l` be checked with the template "variable `a` must be protected by lock `l`?" An effective way to determine if such a MAY belief is plausible is simply to act on it: check the belief using internal consistency and record how often the belief satisfied its rule versus gave an error. We can use these counts to rank errors from most to least credible (essentially ranking the belief by how often it was true versus its negation). The more checks a belief passes, the more credible the (few) violations of it are, and the higher these errors are ranked. The highest ranked errors will be those with the most examples and fewest counter-examples, the middle will be beliefs violated much more often, and the bottom errors will be from almost-always violated beliefs. When inspecting results, we can start at the top of this list and work our way down until the noise from coincidental beliefs is too high, at which point we can stop.

For the code above, we would treat both MAY beliefs, "`l` protects `a`" and "`l` protects `b`," as MUST beliefs. Before checking whether a lock protects a variable, though, we must first determine whether the lock is held at all. Beliefs about locks propagate both forward and backward from `lock` and `unlock` actions: `lock(l)` implies a belief that `l` was not locked before, but locked afterwards, and `unlock(l)` implies a belief that `l` was locked before, but unlocked afterwards. (As a side-effect, this checker could catch double-lock and double-unlock errors.)

Using the lock belief sets, we can then record for each variable (1) how it was checked with the rule (once for each access: four times for `a`, three for `b`) and (2) how many times the variable failed the check. I.e., the number of times it was accessed where $B_l = \{unlocked\}$ (one for `a`, two for `b`). Since `a` is usually protected by `l`, the unprotected access at line 18 is probably a valid error. Since `b` has more errors than correct uses, we would usually discard it. (Programmers are usually right. If they are not, then we have much bigger concerns than a few concurrency bugs.) More generally, we use the "hypothesis test statistic" to rank errors based on the ratio of successful checks to errors. This statistic favors samples with more evidence, and a higher ratio of examples to counter-examples. We discuss this statistic further in Section 5.

3.4 General statistical analysis

For this paper, the only MAY beliefs that concern us are whether a particular set of slot instances can be checked with a rule template T. Thus, conceptually, a statistical checker is an internal consistency checker with three modifications:

1. It applies the check to all potential slot instance combinations. I.e., it assumes that all combinations are MUST beliefs.

2. It indicates how often a specific slot instance combination was checked and how often it failed the check (errors).

3. It is augmented with a function, *rank*, that uses the count information above to rank the errors from all slot combinations from most to least plausible.

For the lock checker above, this would mean that the checker would consider all variable-lock pairs (v, l) as valid instances. For each pair (v, l), it emits an error message at each location where v was used without lock l held, and a "check" message each time v was accessed. For the code above, there are two possible slot combinations, `(a, l)` and `(b, l)`. The instance `(a, l)` has four check messages (lines 5, 11, 15, 18) and one error (line 18). The instance `(b, l)` has three check messages (lines 5, 7, 17) and two errors (lines 7 and 17).

There are two practical differences between internal consistency and statistical checkers. First, for good results, statistical analysis needs a large enough set of cases. In contrast, an internal consistency checker can give definitive errors with only two contradictory cases. Second, to make the universe of slot instances more manageable, a statistical checker may use an optional pre-processing pass that filters the universe of possible slot instances down to those that are at least moderately plausible.

This technique applies to many types of system rules. While internal consistency flags all cases where there are conflicting beliefs as errors, statistical analysis can be used even when the set of checks and errors is noisy.

```
sm internal_null_checker {
 state decl any_pointer v;

 /* Initial start state: match any pointer
    compared to NULL in code, put it in a 'null'
    state on true path, ignore it on false path. */
 start:
    { (v == NULL) } ==> true=v.null, false=v.stop
  | { (v != NULL) } ==> true=v.stop, false=v.null
  ;
   /* Give an error if a pointer in the null state
      is dereferenced in code. */
 v.null:
   { *v } ==> { err("Dereferencing NULL ptr!"); }
   ;
}
```

Figure 2: A simple *metal* extension that flags when pointers compared to null are dereferenced.

3.5 How we implement checkers

We write our analyses in *metal* (see Figure 2), a high-level state machine (SM) language for writing system-specific compiler extensions [9]. These extensions are dynamically linked into *xgcc*, an extended version of the GNU `gcc` compiler. After *xgcc* translates each input function into its internal representation, the extensions are applied down each execution path in that function. The system memoizes extension results, making the analyses usually roughly linear in code length.

Metal can be viewed as syntactically similar to a "yacc" specification. Typically, SMs use patterns to search for interesting source code features, which cause transitions between states when matched. Patterns are written in an extended version of the base language (GNU C), and can match almost arbitrary language constructs such as declarations, expressions, and statements. Expressing patterns in the base language makes them both flexible and easy to use, since they closely mirror the source constructs they describe.

The system is described in more detail elsewhere [9]. For our purposes, the main features of extensions are that they are small and simple — most range between 50 and 200 lines of code, and are easily written by system implementers rather than compiler writers. Many of the errors we find leverage the fact that our analyses can be aggressively system-specific.

A key feature of how we inspect errors is that we first rank them (roughly) by ease-of-diagnosis as well as likelihood of false positives. Our ranking criteria places local errors over global ones, errors that span few source lines or conditionals over ones with many, serious errors over minor ones, etc. We then inspect errors starting at the top of this list and work our way down. When the false positive rate is "too high," we stop. Thus, while our checkers report many errors, we rarely inspect all of them.

Static analysis is scalable, precise, and immediate. Once the fixed cost of writing an analysis pass is paid, the analysis is automatic (scalability), it can say exactly what file and line led to an error and why (precision), and it does not require executing code (immediacy). Further, static analysis finds bugs in code that *cannot* be run. This feature is important for OS code, the bulk of which resides in device drivers. A typical site will have have less than ten (rather than hundreds) of the required devices.

3.6 What systems we check

We have applied our extensions to OpenBSD and Linux. The bulk of our work focuses on two Linux snapshots: "2.4.1" and "2.4.7." Version 2.4.1 was released roughly when the first draft of this paper was written; 2.4.7 roughly when the final draft was completed. Which version we check is determined by which was current when the checker being described was written. Thus, both represent a hard test: live errors, unknown until we found them. We occasionally select example errors from other more intermediate snapshots, but we mainly report results from released versions for reproducibility. Finally, we also apply several of our checkers to OpenBSD 2.8 to check generality.

The main caveat with all of our results is that we are not Linux or OpenBSD implementers, and could get fooled by spurious couplings or non-obvious ways that apparent errors were correct. We have countered this by releasing almost all of our bugs to the main kernel implementers. Many have resulted in immediate kernel patches.

The next two sections continue the discussion of methodology, describing MAY and MUST beliefs in more detail. The rest of the paper evaluates the methodology with case studies.

4 Internal Consistency

Internal consistency finds errors by propagating MUST beliefs inferred at one code location to related locations. Any belief conflict is an error. We introduced internal consistency by describing a null pointer checker in Section 3. In this section we describe other applications of the same general technique and provide a more detailed description of the technique itself. Section 6 and Section 7 then present two case studies of using it to find errors.

Table 1 gives a set of example questions that can be answered using MUST beliefs. For example, as discussed in Section 7, we can use code beliefs to determine if a pointer, `p`, is a kernel pointer or a dangerous user pointer. If `p` is dereferenced, the kernel must believe it is a safe kernel pointer. If it passes `p` to a special "paranoid" routine, it must believe `p` is an unsafe user pointer. It is an error if `p` has both beliefs.

Consistency within a single function is the simplest form of these checkers: if a function `f` treats pointer `p` as an unsafe pointer once, it must always treat `p` as unsafe. Consistency checkers can go beyond self-consistency, though. Code can be grouped into equivalence classes of related code that must share the same belief set. We can then propagate beliefs held in one member to all members in the equivalence class. This gives us a powerful lever: a single location that holds a valid MUST belief lets us find errors in any code we can propagate that belief to. Therefore, we have two primary objectives: (1) finding MUST beliefs, and (2) relating code. The more beliefs found, the more applicable the checker. The more code we can relate, the further we can propagate beliefs, and thus the more likely it is we will find at least one location that holds a valid MUST belief. We discuss each of these two objectives briefly below.

4.1 Inferring MUST beliefs

We infer MUST beliefs in two ways: (1) direct observation and (2) implied pre- and post-conditions. Direct observation uses standard compiler analyses to compute what code *must* believe by tracking actions that reveal code state. The null-pointer checker described in Section 3, for example, can use two direct observations: setting a pointer `p` to null, and checking if `p` is null. The first is an explicit state change, while the second is an observation of state. After changing state, the programmer must believe the changes took effect. Similarly, after observing state, the programmer must believe the observation is true. Note that beliefs inferred from direct observation are validated in that we can compute their truth ourselves.

The second method of inferring beliefs is based on the fact that many actions in code have pre- and post-conditions. For example, division by `z` implies a belief that `z` is non-zero, and deallocation of a pointer, `p`, implies a belief that it was dynamically allocated (pre-condition) and will not be used after the deallocation (post-condition). If we further assume that code intends to do useful work, we can infer that code believes that actions are not redundant. In Section 3 we observed that a check of `p` against null implies a belief that the check was not spurious. Similarly, a mutation such as setting `p` to `q` implies a belief that `p` could have been different from `q`. As Section 6 and Section 8 demonstrate, flagging such redundancies points out where programmers are confused and hence have made errors.

4.2 Relating code

Code can be related both at an implementation level, when there is an execution path from action a to action b, or at an abstraction level, when a and b are related by a common interface abstraction or other semantic boundary. We consider each below.

Code related by implementation. An execution path from a to b allows us to cross-check a's beliefs with b's, typically using standard compiler analysis. In addition to obvious beliefs about shared data, we can also cross-check their assumed execution context and fault models. For example, if a calls b, b usually inherits a's fault model: if a checks foo for failure, b must as well (b must be at least as careful as a). Conversely, a inherits the faults of b: if b can fail, a can as well.

Code related abstractly. If a and b are implementations of the same abstract routine, interface, or abstract data type, we can cross-check any beliefs that this relationship implies.

If a and b implement the same interface, they must assume the same execution context and fault model. In addition, they must also have the same argument restrictions, produce the same error behavior, etc. Example contradictions in these categories include: a exits with interrupts disabled, b with them enabled; a checks its first argument p against null, b dereferences p directly; a returns positive integers to signal errors, b re-

Template	Action	Belief
"Is <P> a null pointer?"	`*p`	Is not null.
Section 6	`p == null?`	null on true, not-null on false.
"Is <P> a dangerous user pointer?"	`p` passed to `copyout` or `copyin`	Is a dangerous user pointer.
Section 7	`*p`	Is a safe system pointer.
"Must `IS_ERR` be used to check routine <F>'s returned result?" Section 8.3	Checked with `IS_ERR`	Must always use `IS_ERR`.
	Not checked with `IS_ERR`	Must never use `IS_ERR`.

Table 1: Questions that can be inferred using internal consistency. Ranking the results is not necessary because a single contradicted belief must be an error. A nice feature: contradictions let us check code without knowing its context or state.

turns negative integers. We can even perform checks across programs, such as checking that different Unix implementations of POSIX system calls have the same argument checks and return the same error codes. Finally, if a and b are equivalent, this implies we can also (symmetrically) cross-check the different pieces of code that use a against those that use b.

How can we tell when we can relate code at an abstract level? One way, of course, is by divine intervention: if we know a and b are the same, we can cross-check them. Lacking this knowledge, we must find these relationships automatically. One simple technique is to relate the same routine to itself through time across different versions. Once the implementation becomes stable, we can check that any modifications do not violate invariants implied by the old code. Another way to relate code is to exploit common idioms that imply that two implementations provide the same abstract interface. A common idiom is that routines whose addresses are assigned to the same function pointer or passed as arguments to the same function tend to implement the same abstract interface. Our most important use of this trick is to cross-check the many implementations of the same interface within a single OS, such as different file systems that export the same virtual file interface to the host OS and different device drivers that all implement an interrupt handler (see Section 7).

5 Statistical analysis

Statistical checkers find errors in MAY beliefs. They use statistical analysis to filter out coincidences from MAY beliefs by observing typical behavior over many examples.

We sort the errors from statistical analysis by their ranking according to the z statistic for proportions [14]:

$$z(n, e) = (e/n - p_0)/\sqrt{(p_0 * (1 - p_0)/n}$$

where n is the population size (the number of checks), c the number of counter examples (errors), e the number of examples (successful checks: $n - c$), p_0 the probability of the examples and $(1 - p_0)$ the probability of the counter-examples. This statistic measures the number of standard errors away the observed ratio is from an expected ratio. We typically assume a random distribution with probability $p_0 = 0.9$. The ranking, z, increases as n grows and the number of counter-examples decreases. Intuitively, the probability of an observed result also increases with the number of samples. For the purposes of bug finding, perfect fits are relatively uninteresting. Error cases reside where there are at least some number of counter-examples. Given enough samples, derivation can infer a wide range of rule instances. Table 2 gives a set of example questions that can be answered using statistical checkers.

There are a couple of things to note about this ranking approach. First, it can be augmented with additional features. One useful addition is code trustworthiness: code with few errors is more reliable for examples of correct practice than code with many.

Second, it has an interesting inversion property. If $z(N, E)$ ranks instances that satisfy a template T, then it is commonly useful to also rank $z(N, N - E)$, which computes $\neg T$. Often, if template T is useful, its negation $\neg T$ is useful as well. We call this the *inverse principle*.

Third, statistical analysis, like internal consistency, can exploit the non-spurious principle. Many properties must be true for at least one element: a lock must protect some variable or routine; a security check must protect some sensitive action. For such cases, an empty template slot signals a derivation error. For example, if the lock checker in Section 3.3 finds that a lock `l` has no variable `v` such that the ratio of checks to errors for `(l, v)` gives an acceptable rank ($z(checks, checks-errors)$), then we know there is a problem: either our analysis does not understand lock bindings, or the program has a serious set of errors. In general, this idea can be trivially applied to the statistical analysis of any property that must have at least one member. In some cases, we can also use it to immediately promote a MAY belief to a MUST belief without any statistical analysis. For example, a critical section that only accesses a single shared variable implies that the code must believe that the variable is protected by the critical section's lock.

5.1 Handling noise

One concern when deciding if MAY beliefs are true is noise from both coincidences and imperfect analysis. There are three key features we use to counter such noise: large samples, ranking error messages, and human-level operations.

First, we can easily gather large representative sets of behavioral samples by basing our approach on static analysis, which allows us to examine all paths. While these paths are inherently noisy, there are so many that we can derive many patterns and only use the most promising candidates.

Second, we can counter noise in our error messages by using the z statistic value to rank errors from most to least credible. We can then inspect these errors and stop our search once the false positive rate is deemed too high. This step is in some sense the most crucial. A naive use of the z statistic would be to use it to rank beliefs rather than errors. Our initial approach did just that: we selected a threshold t, calculated z for each belief, and treated those beliefs above the threshold as MUST beliefs. We then checked rules using these beliefs and threw all of the resultant errors in the same pool. The problem with this approach is its sensitivity to t's value. If t is too low, we drown in false positives. If it is too high, we do not find anything interesting. However, ranking error messages rather than beliefs completely avoids these problems: we can start inspecting at the top where the most extreme cases are (and thus the false positive rate is lowest). Noise will increase steadily as we go down the list. When it is too high, we stop. Switching to this approach made a notable difference in building effective checkers.

Finally, our analyses are also aided by the fact that code must be understood by humans. Important operations are usually gifted with a special function call, set of data types, and even specific naming conventions. In fact, we can commonly use these *latent specifications* to cull out the most easily understood results (e.g., when deriving paired functions to give priority to pairs with the substrings "`lock`," "`unlock`", "`acquire`," "`release`", etc.) We discuss this further below.

5.2 Latent specifications

Latent specifications are features designed to communicate intent to other programmers and idioms that implicitly denote restrictions and context. Because they are encoded directly in program text, extensions can easily access them to determine where and what to check, as well as what conditions hold at various points in the code. Leveraging these encodings makes our approach more robust than if it required that programmers write specifications or annotate their code, since in practice, it is an event worth remarking when they do. Statistical analysis checks (and in some cases internal consistency checks) leverage latent specifications to filter results and to suppress false positives.

The most primitive latent specifications are naming conventions. Familiar substrings include "lock," "unlock," "alloc," "free," "release," "assert," "fatal," "panic," "spl" (to manipulate interrupt levels), "sys_" (to signal system calls), "_intr" (to flag interrupt handlers), "brelse" (to release buffer cache blocks), and "ioctl" (as an annotation for buggy code). Our statistical analysis passes use these as auxiliary information when flagging potentially interesting functions.

At a slightly higher-level, most code has cross-cutting idioms that encode meaning. For example, error paths are commonly signaled with the return of a null pointer or a negative (or positive) integer. These annotations allow checkers to detect failure paths at callers, and error paths within callees.

Code is also interlaced with executable specifications. For example, debugging assertions precisely state what conditions must hold when a routine runs. Another example is routines such as `BUG` in Linux and `panic` in BSD. These calls serve as precise annotations marking which paths cannot be executed (since the machine will have rebooted after either call). Our checkers use them to suppress error messages on such paths.

Finally, specifications can be completely transparent but shared across all code in a given domain. Examples include the popular rules that null pointers should not be dereferenced and that circular locking is bad. A compiler extension can directly encode this information.

6 Internal Null Consistency

The next four sections are case studies evaluating our approach: this section and the next focus on internal consistency, while Section 8 and Section 9 focus on statistical analysis.

This section implements a generalized version of the internal consistency checker in Section 3. It finds pointer errors by flagging three types of contradictory or redundant beliefs:

1. Check-then-use: a pointer `p` believed to be null is subsequently dereferenced.
2. Use-then-check: a pointer `p` is dereferenced but subsequently checked against null. Note that this is only an error if no other path leading to the check has the opposite belief that `p` is null.
3. Redundant checks: a pointer known to be null or known to be not null is subsequently checked against null (or not null). As above, all paths leading to the check must have the same known value of `p`.

Check-then-use and use-then-check violate the rule "do not dereference a null pointer $<p>$." They tend to be hard errors that cause system crashes. Redundant checks violate the rule "do not test a pointer $<p>$ whose value is known." While violations do not cause crashes directly, they can flag places where programmers are confused.

Conceptually, as described earlier in the paper, the checker associates a belief set with each pointer `p`. The beliefs in the list can be one or more of: (1) *null*, (2) *not-null*, or (3) unknown (the empty list). The checker rules are straightforward:

1. A dereference of pointer `p` adds the belief *not-null* to `p`'s belief set. It is an error if the belief set contained *null*.
2. A pointer checked against null (or non-null) implies two beliefs. First, it propagates backwards the belief that the pointer's value is unknown (i.e., it could be either null or not null). The checker flags an error if p is known to have a more precise belief. Second, the check propagates forward the belief that `p` is null on the true path and non-null on the false path.

For simplicity, we implemented each error type using a different extension. The implementation is straightforward. For example, the full check-then-use checker is written in less than 50 lines (Figure 2 gives a stripped down version). It puts every pointer `p` compared to null in a "`null`" state and flags subsequent dereferences as errors. The others follow a similar pattern except that

Template (T)	Examples (E)	Population (N)
"Does lock <L> protect <V>?"	Uses of `v` protected by `l`	Uses of `v`
"Must <A> be paired with <B>?"	paths with `a` and `b` paired	paths with `a`
"Can routine <F> fail?"	Result of `f` checked before use	Result of `f` used
"Does security check <Y> protect <X>?"	`y` checked before `x`	`x`
"Does <A> reverse <B>?"	Error paths with `a` and `b` paired	Error paths with `a`
"Must <A> be called with interrupts disabled?"	`a` called with interrupts disabled	`a` called

Table 2: Templates derivable with statistical analysis; the statistical methods are necessary to counter coincidental couplings. These were ranked using $z(N, E)$. A commonly useful trick is to use $z(N, N - E)$ to derive $\neg T$.

they make sure that the error would occur on all paths before reporting it.

Unlike most checkers, the most interesting challenge for these is limiting their scope rather than broadening it: preventing beliefs from violating abstraction boundaries, suppressing impossible paths, and deciding on the boundary between "good" programming and spurious checks.

Our checkers must ensure that some beliefs do not flow across black-box abstraction barriers. For example, macros can perform context-insensitive checks, which add the `null` belief to a pointer's belief set. However, this belief is not one we can assume for the macro's user. Thus, we do not want it to propagate outside the macro since otherwise we will falsely report dereferences of the pointer as an error. Almost all false positives we observed were due to such macros. To reduce these, we modified the C pre-processor to annotate macro-produced code so we could truncate belief propagation. One counter-intuitive result is that unlike almost all other checkers, both use-then-check and redundant-check generally work best when purely local so as to prevent violations of potential abstraction boundaries.

A second problem is that the check-then-use checker is predisposed to flag cases caused by the common idiom of checking for an "impossible" condition and then calling "`panic`" (or its equivalent) to crash the machine if the condition was true:

```
if (!idle)
   panic("No idle process for CPU %d", cpu);
idle->processor = cpu;
```

Here, `panic` causes a machine reboot, so the dereference of a null `idle` is impossible. These calls are essentially latent specifications for impossible paths. To eliminate such problems, all checkers, including those in this section, pre-process the code with a 16-line extension that eliminates crash paths, thereby removing hundreds of false positives.

Finally, we must decide on a threshold for redundancy and contradiction errors. Checks separated by a few lines are likely errors, but separated by 100 could be considered robust programming practice. We arbitrarily set this threshold to be roughly 10 executable lines of code.

6.1 Results

Table 3 shows the errors found in Linux. Some of the more amusing bugs were highlighted in Section 3.1. In the check-then-use example bug, the desire to print out a helpful error message causes a kernel segmentation fault. The second example in Section 3.1 demonstrates the most common use-then-check error idiom: a dereference of a pointer in an initializer followed by a subsequent null check. This example code was cut-and-paste into *twenty* locations.

Checker	Bug	False
check-then-use	79	26
use-then-check	102	4
redundant-checks	24	10

Table 3: Results of running the internal null checker on Linux 2.4.7.

While the redundant-checks checker found far fewer errors, it did provide evidence for our hypothesis that redundancy and contradiction is correlated with general confusion. Two such redundant cases follow the example below where, after an allocation, the wrong pointer value is checked for success.

```
/* 2.4.7/drivers/video/tdfxfb.c */
fb_info.regbase_virt = ioremap_nocache(...);
if(!fb_info.regbase_virt)
   return -ENXIO;
fb_info.bufbase_virt = ioremap_nocache(...);
/* [META: meant fb_info.bufbase_virt!] */
if(!fb_info.regbase_virt) {
     iounmap(fb_info.regbase_virt);
```

Contradiction also flagged 10 suspicious locations where a contradictory pointer check of `tmp_buf` pointed out an error:

```
/* 2.4.7/drivers/char/cyclades.c */
if (!tmp_buf) {
   page = get_free_page(GFP_KERNEL);
   /* [META: missing read barrier] */
   if (tmp_buf)
      free_page(page);
```

Here, a missing cache "read barrier" will potentially allow an access to a stale pointer value held in `tmp_buf`; the other similar locations had spurious synchronization code.

6.2 Discussion

The main results of this section are: (1) the ideas that redundant and contradictory observations can be used to find errors and (2) demonstrating that even contradiction checking for simple beliefs can find many errors in real code.

The checkers in this section can be generalized to find other redundancies and contradictions. There are many opportunities for such checks since essentially every action in source code implies a set of beliefs. Example checks include warning when: (1) a critical section does *not* access some shared state; (2) a structure field is never read or its precision is under-utilized; (3) a write mutation is never read; (4) functions that cannot fail are checked; (5) general expressions in conditionals are impossible or redundant; (6) paths violate assertion conditions. One contribution of our work is the realization that traditional compiler optimization passes, such as dead code elimination and constant propagation, can become error checkers with only minor re-tooling.

7 A Security Checker

This section describes a checker that finds security errors. It uses internal consistency to check slot instances for the rule template "do not dereference user pointer <p>," and latent specifications to automatically suppress false positives from kernel "backdoors." Without these techniques, we could only check a fraction of kernel code because we could not determine which pointers were dangerous. With it, we readily found 35 security holes in Linux and OpenBSD.

7.1 The problem

Operating systems cannot safely dereference user pointers. Instead they must access the pointed-to data using special "paranoid" routines (e.g. `copyin` and `copyout` on BSD derived systems). A single unsafe dereference can crash the system or, worse, give a malicious party control of it. With a list of pointers passed from user-level, static analysis can readily find such errors. Unfortunately, from experience, the manual classification of pointers is mystifying. The worst offenders, device drivers, make up the bulk of operating systems, interact extensively with user code, but follow no discernible convention for denoting user pointers. Those routines that do follow some vague naming convention tend to have a mixture of safe pointers passed in by the kernel and unsafe pointers passed raw from the user or fabricated from input integers. Thus, if we cannot classify these dangerous pointers we will miss all security holes in the largest source of such errors.

We solve this problem by using internal consistency to derive which pointers are believed to be user pointers and then checking that they are never treated as kernel pointers (dereferenced). The rules for this checker are as follows:

1. Any pointer that is dereferenced is believed to be a safe kernel pointer.

2. Any pointer that is sent to a "paranoid" routine is believed to be a "tainted" user pointer.

3. Any pointer that is believed to be both a user pointer and a kernel pointer is an error.

The checker refines this process by also considering arguments to functions abstractly related through function pointers. If two functions f and f' are assigned to function pointer fp and f treats its ith parameter p as a user pointer, then f' must also treat its ith parameter p' as a user pointer. As with our other consistency checkers, if one use is correct, this technique can check all other related uses.

```
    /* net/atm/mpoa_proc.c */
1:  ssize_t proc_mpc_write(struct file *file,
2:                          const char *buff) {
3:     page = (char *)__get_free_page(GFP_KERNEL);
4:     if (page == NULL) return -ENOMEM;
5:     /* [Copy user data from buff into page] */
6:     retval = copy_from_user(page, buff, ...);
7:     if (retval != 0)
8:         ...
9:     /* [Should pass page instead of buff!] */
10:    retval = parse_qos(buff, incoming);
11: }
12: int parse_qos(const char *buff, int len) {
13:     /* [Unchecked use  of buff] */
14:     strncpy(cmd, buff, 3);
```

Figure 3: 2.3.99 Security error: the driver carefully copies the user memory to a safe location (in `page`) but then immediately passes the unsafe user pointer `buff` to `parse_qos`, which reads from it using `strncpy`. A striking feature is that this error is amidst a fair amount of safety-conscious boilerplate, down to the programmer using a `const` qualifier on `buff` to ensure that `buff` is not accidentally mutated.

7.2 Implementation

Below, we discuss the checker implementation, false positive suppression, and the manual effort needed to retarget the checker to a new system. We use a security hole from Linux 2.3.99 (shown in Figure 3) as a running example to clarify our description of the implementation. The checker finds this hole as follows. First, the call to to the paranoid routine `copy_from_user` (line 6) implies the belief that `buff` is tainted. Second, `buff` is passed to `parse_qos` (line 10), which then passes it to `strncpy` (line 14), which will in turn dereference it, implying `buff` is a safe kernel pointer. Since these beliefs conflict, the checker emits an error.

The checker works in two passes: a global derivation pass, which computes summaries and checks function-pointer assignment, and a local checking pass, which checks function calls and pointer dereferences using the summaries computed by the first pass.

The global pass first computes three summaries: (1) a transitive closure of all functions that taint their parameters, (2) a transitive closure of all functions that dereference their parameters, and (3) every function pointer assignment (including assignments from static structure initialization). The results of this pass are passed to the next step through three text files.

The two transitive closure operations use essentially the same technique. For the tainted list, we would like to know all functions whose parameters are eventually (through some execution path) passed to a paranoid routine. This can happen directly, as in line 6 of the above example or indirectly, as in line 2 of the following made-up function `foo`:

```
1: void foo(struct file *f, char *buff) {
2:     ssize_t sz = proc_mpc_write (f, buff);
```

The dereferencing list is computed in exactly the same manner. The function `parse_qos` dereferences its parameter `buff` directly through the call to `strncpy` at line 14. Any functions that call parse_qos passing one of their own parameters as the first argument to parse_qos are also marked. The result of these two passes are emitted as two lists (a tainting list and a dereference list) of the form *(fn, i)*, which indicates function *fn* taints or dereferences its ith parameter.

After the summaries are completed, we use the three summaries to check for conflicts in function pointer assignment: it is an error for a function pointer to be assigned one function that dereferences its ith parameter and another that taints its ith parameter. I.e., we flag an error if function pointer *fp* is assigned f and f' and (f, i) is on the tainted list and (f', i) is on the dereference list.

We then run the local checking pass, which uses these three lists similarly to warn when a tainted pointer is dereferenced or passed to a routine that would dereference it. It goes over each function twice. The first pass examines all call sites, marking any pointer passed as a parameter to a tainting routine as tainted. In our example, the pointer buff is marked as tainted because it is passed to the tainting routine copy_from_user at line 6. The second pass checks all uses of tainted pointers and flags all raw dereferences of them or any call to a routine that could do a dereference. In the example, the call to parse_qos at line 10 is flagged because the tainted pointer buff is passed to the dereferencing routine parse_qos.

Note that this is a good example of how our inference approach can meld gracefully with programmer annotations. Since we use text files for summaries, annotations for which pointers are user pointers can easily be added to the file manually, or extracted from source annotations and inserted.

False positives. The largest source of false positives are kernel backdoors that check if they were called from user code or kernel code. In the latter case, they can safely dereference pointers, but such uses would be flagged by a naive checker. Fortunately, this is such a dangerous activity that kernel programmers used stylized naming conventions for the Boolean flags used to determine what context they are operating in. Both OpenBSD and Linux use variables named from_user or to_user. Our extension treats these variables as implicit specifications and tells xgcc's dataflow framework that they always evaluate to true so that the backdoor path is pruned away.

While not obvious, our other main technique for suppressing false positives is the list of dereferencing functions. The most natural checker would simply warn when a tainted value was passed as a function parameter, rather than checking if the call actually dereferenced the value. Unfortunately, the prevalence of type coercion would cause too many false positives. Device code commonly uses a value as a pointer value on one path, but as an integer on the other. In a naive checker, the first path would cause the value to be tainted, and then the second path would cause an error message if it called a function with the tainted value, even if that function used the value as an integer.

Manual labor. The checker is mostly system independent. There are three system-specific parts:

1. A text file listing the paranoid routines. There are four of these routines for BSD, 28 for Linux.

OS	Errors	False	Applied
OpenBSD 2.8	18	3	1645
Linux 2.4.1	12 (3)	16 (1)	4905
Linux 2.3.99	5	n/a	n/a

Table 4: The user-pointer checker found 35 bugs in total. It had 19 false positives and was applied roughly 6500 times in Linux 2.4.1 and OpenBSD 2.8. The numbers for Linux 2.3.99 are not available since we used an earlier version of the system. The numbers in parentheses for the 2.4.1 kernel were the errors and false positives from cross-checking functions assigned to the same function pointer.

2. A text file listing tainting or dereferencing routines that should be ignored. These suppress false positives caused by the limitations of the system we use for static analysis and are independent of our deriving approach. There are 15 of these functions for BSD, 19 for Linux.

3. A list of variables names (these can be substrings) that indicate kernel backdoors. Some form of annotation would be needed by any system; we expect that leveraging the source as we do reduces this effort to be roughly as small as it can reasonably be.

Applying the checker on a new system typically follows three stages. First, the tainting routines are specified and the checker is run over the system. Second, the results are ranked and inspected. If a given function causes too many false positives, it is added to the list of ignored routines and all related errors are skipped. Similarly, false positives from kernel backdoors cause us to add the flag to the extension's list and skip related errors. Finally, we rerun the checker; the system will re-mark errors that were already inspected.

7.3 The results

Table 4 lists how many bugs we found, the number of false positives, and how often the check was applied. All bugs have led to subsequent kernel patches. The false positive ratio is fairly low. However, given the seriousness of these bugs, a much higher rate would still have been acceptable.

In Linux, device drivers account for all but one error. The bulk of these errors were concentrated in "ioctl" calls. Bugs also tend to cluster, where assumptions that led to one mistake avalanche into several. The worst example of this was code in the "appletalk" ioctl routine which had four errors all with the same pattern of calling copy_to_user to safely copy out a user pointer rt while simultaneously calling another function, that would promptly dereference it. A representative example is:

```
/* drivers/net/appletalk/ipddp.c:ipddp_ioctl */
case SIOCFINDIPDDPRT:
  if(copy_to_user(rt, ipddp_find_route(rt),
                  sizeof(struct ipddp_route)))
          return -EFAULT;
```

Here, our analysis would taint rt because it is passed to a copy_to_user call and then warn about the call ipddp_find_route(rt), which dereferences it.

In OpenBSD the bulk of the errors were in the "System 4" compatibility layer. Most of these were due to simply reversing the arguments to the paranoid functions `copyin` and `copyout`. This error was faithfully replicated into several different places. Interestingly, in each of these places, code immediately above the errors handles the parameter passing correctly! Similar argument reversal bugs had been caught in Linux 2.3.99.

Cross-checking functions assigned to the same function pointer found three errors in 2.4.1 and one false positive. (We did not do this analysis on the other systems). Two errors came from improper implementations of routines assigned to the "`write`" method field in the `file_operations` structure; 55 of the routines assigned to this pointer treated their second parameter as tainted, but two buggy routines, `fop_write` and `mdc800_device_write` dereferenced this pointer directly. For example:

```
/* 2.4.1:fop_write:sbc60xxwdt.c: buf is tainted. */
size_t
fop_write(struct file *file, const char *buf...)
    ...
    /* now scan */
    for(ofs = 0; ofs != count; ofs++)
       if(buf[ofs] == 'V')
           wdt_expect_close = 1;
```

Amusingly, as in Section 7.2, the author uses the `const` qualifier for type safety while busily compromising system security. Although we expected to find more bugs by using function pointer equivalence, the small bug counts are reassuring: they imply most call chains treat a pointer correctly in at least one place, allowing us to check the entire call chain.

8 Inferring Failure

This section finds errors where routines are not checked or are incorrectly checked for failure. It uses statistical analysis to derive and check slot instances for the rule template "function <*f*> must be checked for failure." As we demonstrate, basing this analysis on client beliefs allows us to find restrictions indirectly represented in source code. As a result, we can find completely unanticipated errors that traditional analysis would miss.

8.1 The problem

Kernel code must check for failure at every resource exhaustion or access control point. The enormous number of such cases makes missing checks common. For example, our previous static analysis found 79 cases where Linux kernel code did not check the result of four memory allocation procedures; a similar analysis of one allocator in OpenBSD found 49 cases [9]. Any of these could lead to segmentation faults when allocation failed under high load.

Non-memory allocation functions can also fail. Such failures are frequently silent, making them worse in some ways than kernel crashes. As an example, a colleague recently wasted several days tracking a bug hidden by a single missing check in a graphics device driver that should have signaled that the driver could not allocate a range of device memory [16]. The effect (failure to display graphics) is what one would assume happens when a new card was misconfigured not, as in this case, a bug in a dauntingly opaque driver.

A traditional approach to finding all routines that could fail would be to compute the transitive closure of all routines that could return null pointers or error codes. In practice, this has two limitations. First, while easy when such values are returned directly, it becomes undecidably difficult when variables are returned. For example, in the C code "**`return foo→bar;`**" the field `bar` in structure `foo` may or may not have a null value because it was never explicitly set or cleared. From the program text we cannot tell. Second, this type of analysis can give a uselessly high number of false positives, since many routines cannot fail if pre-conditions are met (e.g., searching a list) or their failure may be outside of a program's fault model.

We use a more robust approach by deriving what routines programmers believe can plausibly fail. Missing checks on almost-always checked routines are good candidates for errors. Similarly, checks on almost-never checked routines frequently signal misunderstood interfaces.

8.2 The checkers

We wrote two checkers based on this model. The first ensures that routines returning null pointers are checked before use. The second ensures that routines that return integer error codes are checked. Both work as follows:

1. They assume that *all* functions can fail.
2. If the result of a function f is ignored or used without checks, the checker emits an error message.
3. If the result of a function f is checked before use the checker emits a "checked" message. If f has a high ratio of check to error messages, this implies that the client believes such checks are necessary.

After going over the entire system, the checker counts for each function f both the total number of errors messages ($f.err$) and the total number of checked uses ($f.chk$). It uses these counts to rank error messages via $z(f.err + f.chk, f.chk)$: errors for functions that have a high value for z are put above those with lower values. Thus, the highest ranked errors will be those for almost-always checked functions, and hence have the most probability of being real errors. Table 5 lists the number of null errors our checker found.

Note that we use statistical ranking because the beliefs we can infer above are actually MAY beliefs. We cannot tell if a unchecked call to f means (1) the programmer believes f does not need to be checked or (2) they believe f does not need to be checked *at this particular callsite*. In this case, examining many callsites allows us to (probablistically) generalize beliefs.

8.3 The worst error

We have hand-examined in excess of a thousand errors of various types. The following error in Linux version "2.4.0-test9" was one of the worst we have seen. It would be missed by traditional null pointer analysis but, interestingly, the checker automatically found it, even though it was not a type of error we had thought of. The error started with the following confusing false positive, where the checker emitted a message stating that the unchecked pointer `shp` was being used:

Version	Bug	False
2.4.1	52 + 102	16
OpenBSD	27 + 14	21
Total	195	37

Table 5: Errors and false positives of running the derived null checker on Linux 2.4.1 and OpenBSD 2.8. The bugs are in $a+b$ format where a are errors from derived functions previously known to return NULL (e.g., `kmalloc`), and b are errors from functions we did not know about. The OpenBSD results are for an older version of our checking system; the newer version gives 5-10 times lower false positive rates for this checker. Note that we already checked earlier versions of these OSes [9] and submitted bug reports. As a result, developers had fixed many of the easy cases.

```
/* ipc/shm.c:map_zero_setup */
if (IS_ERR(shp = seg_alloc(...)))
     return PTR_ERR(shp);
```

The false positive is caused by a very sleazy error return convention. The routine `seg_alloc` returns a valid pointer on success, but on failure, rather than return a null pointer, casts one of several integer error codes to a pointer and returns that (e.g., essentially doing "`return (void *)-ENOMEM;`"). Callers must then check the returned pointer for errors using the special call, `IS_ERR` (as this caller does), which reverses the cast and compares the value to a negative integer. The key point is that on failure `seg_alloc` returns a non-null, bogus pointer. So, if this is true, why is our null checker looking for it at all? Somewhere at least one of `seg_alloc`'s callers must have (understandably) forgotten about this idiom and checked the function's return against null! Indeed, searching the deriver log turned up the following routine:

```
/* 2.4.0-test9:ipc/shm.c:newseg
     NOTE: checking 'seg_alloc' */
if (!(shp = seg_alloc(...)))
      return -ENOMEM;
id = shm_addid(shp);
```

So what happens when `seg_alloc` fails? In this case, the null pointer check on `shp` will fail, and `newseg` will call the routine `shm_addid` with the mangled pointer `shp` as its argument. This pointer is then passed to another routine, `ipc_addid`, as the parameter "`new`", where two catastrophic things happen:

```
int ipc_addid(..., struct kern_ipc_perm* new)
  new->cuid = new->uid = current->euid;
  new->gid = new->cgid = current->egid;
  ids->entries[id].p = new;
```

First, the routine writes to the bogus memory location pointed to by `new`. Since the error bits will likely form a valid physical address, the writes to `new` will corrupt physical memory. Second, the entry is placed on an array of structures (later to be used) almost guaranteeing that this corruption will re-occur on a sporadic but continuous basis. Traditional null-pointer error checking would be completely oblivious to cases such as this, while it can be detected fairly easily by just examining inconsistencies in how callers handle function returns.

`IS_ERR` consistency checking. Not uncommonly, one type of error leads us to another. For the error above, an obvious secondary check is to verify that all routines that use a similar error-pointer "trick" are correctly handled by all callers. We use a two-pass consistency checker that enforced the restriction: If a function f's return value is ever checked using `IS_ERR`, then all callers of f must check its result this way. There were 78 such functions in the 2.4.1 kernel. While we did not find any bugs like the `seg_alloc` case above, we did find the opposite problem, where the code did an `IS_ERR` check on a function that actually returned `NULL`. This check will always fail, causing the client to think there was no error and that they can dereference the null pointer. Out of 295 checked call sites, there were five such errors and six false positives (caused by unusual coding styles). All caught errors were fixed.

8.4 Discussion

A nice feature of this analysis style is that it finds additional, unforeseen types of errors. The `IS_ERR` checking idiom is one such example — we never imagined such a bizarre error type, did not know to look for it, and when shown, did not initially understand what was going on. It seems unlikely to have been picked up with traditional, generic null-pointer analysis. We only find it because of the way we derive failing routines.

Misunderstood interface uses are another unanticipated error type that the approach found. For the checkers in this section, this manifests as routines that cannot fail but are spuriously checked by callers. We have found examples of this in all systems we have looked at. In one commercial system, there was a routine, `GetFrame`, that showed up as a mixed function in our analysis. When examined, it turned out to never return and so could not fail. Despite this there were four places that checked for failure. These checks implicitly showed the callers did not realize the routine would not return and thus assumed the code after these calls would be run. What had happened was that: (1) an initial version of the routine could fail, (2) code had been written that way, (3) the interface had changed, but the code had not been updated, and so (4) subsequent code used the initial failure checks as the correct way to treat it. A similar case shows up in OpenBSD with the `getblk` function, which cannot fail but is treated that way by clients. These checks are a nice way of flagging code written by programmers that (most likely) have a poor grasp of internal interfaces and whose code should be audited. The basic approach of finding deviant interfaces could be extended to richer examples.

In summary, this type of analysis serves as a good supplement to a more traditional compiler analysis approach. In a sense, it checks at the level of an abstract interface by viewing how clients treat a routine rather than by only examining its implementation. A possible future direction would be to use this approach to catch when an implementation violates its specification by checking for implementation actions that contradict client assumptions (e.g., a return of null when clients do not check).

9 Deriving Temporal Rules

Another type of error our approach can find are violations of temporal rules, where sequences of actions need to be considered. Some examples of temporal rules are "no $<a>$ after $<b>$" (freed memory cannot be used), "$<b>$ must follow $<a>$" (unlock must follow lock), and contextual rules such as "in context $<x>$, do $<b>$ after $<a>$" (on error paths, reverse side-effects). While there are a small number of such templates, there are many different specific operations that can fit in them.

In this section, we look at two temporal-rule templates. The first checks the "no $<a>$ after $<b>$" rule above by flagging cases where memory is used after being passed to a potential deallocation function. The second analysis checks the rule "$<b>$ must follow $<a>$" by pre-processing the code to build up traces of related function calls on local program paths, which are then examined to find sequences of operations that fit this rule. Good fits are kept, bad fits discarded.

9.1 "No $<a>$ after $<b>$": deallocation

This rule checks that freed memory is not used. However, finding all rule violations is difficult because many systems have a large set of deallocation functions, ranging from general-purpose routines, to wrappers around these routines, to a variety of ad hoc routines that manage their own internal free lists. This section describes a checker that can infer all such routine types.

The checker exploits a single, simple implication: if a function's argument is not used after the call, we can infer that the programmer *may* believe that this is a deallocation function. Since this is a MAY belief, we use the following three-step statistical process to find likely violations:

1. Blindly assume that *every* function frees *all* of its arguments.
2. For every function-argument pair, count: (1) n, the number of times we check the pair, and (2) err, the number of times the pair failed the check.
3. Segregate errors by pair, and use the z statistic to rank each pair by how often the pair was checked (n) versus how often it failed (err) (i.e., $z(n, n - err)$). This pushes the errors from the most likely pairs to the top of the list.

Unsurprisingly, checking all argument pairs is too computationally expensive in practice. We reduce this overhead by using latent specification to automatically filter the population of candidate functions to contain only those that have names suggestive of deallocation (containing the substring "free," "dealloc," etc.).

We applied this process to the 2.4.1 Linux kernel and inspected the top 14 ranked functions. There were 23 free errors, 14 of which would have been missed by our previous work. There were 11 false positives in total. Figure 4 shows one interesting example of a double-free that would not be caught by the old system. Here, if the second call to `copy_to_user` fails, the code carefully frees both `c` and `buff` and then (possibly because of a missing return) frees both variables again. This bug is particularly bad because it opens a security hole for users: they can trigger this path deterministically by calling the code with invalid pointers (which will cause `copy_to_user` to fail).

```
/* drivers/block/cciss.c:cciss_ioctl  */
if (copy_to_user(...)) {
    cmd_free(NULL, c);
    if (buff != NULL) kfree(buff);
    return( -EFAULT);
}
if (iocommand.Direction == XFER_READ)
    if (copy_to_user(...)) {
        cmd_free(NULL, c);
        kfree(buff);
    }
cmd_free(NULL, c);
if (buff != NULL) kfree(buff);
```

Figure 4: Double-free security hole: the first `copy_to_user` correctly deallocates its storage and returns with an error. The second appears to have omitted a return statement, thus allowing the code to fall through and hit a duplicate free of both `c` and `buff`.

A second disastrous bug that frees memory but then allows it to escape occurs in this routine from the "proc" file system code:

```
/* fs/proc/generic.c:proc_symlink */
ent->data = kmalloc(...);
if (!ent->data) {
    kfree(ent);
    goto out;
}
out:
 return ent;
```

If the allocation fails, then the routine will free the data pointed to by `ent`, but, instead of returning null, return `ent`. Callers checking for failure will get a non-null pointer, assume the routine succeeded, and then use this pointer. Of the remaining errors, the most exuberant was a single use-after-free error that an implementer carefully cut and paste to seven different locations.

9.2 "$<b>$ must follow $<a>$"

In this subsection, we find errors with the temporal template "$<b>$ must follow $<a>$." A function, `a`, followed by `b` implies the MAY belief that `a` must always be followed `b`. This belief is not a MUST belief because such pairings can also be coincidental. Intuitively, we can determine if this pairing is real or spurious by comparing the number of code paths with a call to `a` to the number of paths with the pair of calls `a, b`. Non-spurious couplings will have near-equal counts (errors make them slightly different).

Conceptually, the checker for this rule is almost identical to the deallocation checker in the previous subsection. We blindly assume that all possible function pairs must obey the rule. Then, for each a-b pair we count (1) n, the number of times we check the pair, and (2) err, the number of times the pair failed the check. We can then rank the pairs using the z statistic (i.e., $z(n, n - err)$).

In practice, we need to make two modifications. First, to control the overwhelming number of pairwise combinations, we first pre-process all possible paths to get plausible a-b pairs. Second, to reduce the inspected number of false positives, we use the z statistic to rank error messages both by pair plausibility as well as by individual error message. We discuss each modification below.

Selecting plausible a-b pairs. We reduce the number of possible pairwise combinations by automatically extracting "traces" from the source code, culling out plausible a-b pairs, and then feeding these to the checker. We only consider three idiomatic types of function traces:

```
/* type 1 */   /* type 2  */   /* type 3 */
p = foo(...);  foo(p, ...);    foo();
bar(p);        bar(p, ...);    bar();
baz(p);        baz(p, ...);    baz();
```

Type 1 traces begin when the result of a function is assigned to a variable that is then passed as the first argument to more than one subsequent call. This can happen when a handle is returned, used in some number of calls and then (possibly) released. The trace for this example would be "`foo:bar:baz`." Type 2 traces begin with the variable passed without an initial assignment. In Figure 5, the call to "`spin_lock_irqsave`" followed "`spin_unlock_irqrestore`" would lead to a type 2 trace. The third is a series of no-argument function calls. In Figure 5, the paired calls "`lock_kernel` and "`unlock_kernel`" would generate a type 3 trace. Using these idioms to filter the pairs makes the analysis manageable yet effective because they cover a wide set of uses.

We generate (a, b) pairs in three steps. First, we run the trace extractor over the kernel. The result is a file containing each unique trace; as an order-of-magnitude, the 2.4.1 kernel generates roughly 130K such traces. Second, we post-process the traces using the z statistic to select plausible a-b pairs, using latent specifications to increase the weight of functions that have names suggestive of paired functions (the substrings "lock," "unlock," "acquire," "release," "brelse," "spl," etc.). Finally, we feed the selected functions to an implementation of the checker sketched above.

Hierarchical ranking. The classic error for this rule is many paths with `a` correctly followed by `b`, and only one or two paths without it. The classic false positive is a single use of `a` and no use of `b`. Such false positives typically happen when the local analysis we use to check a-b pairs encounters a wrapper routines that never pair `a` and `b`. For example, a locking wrapper function will acquire a lock, but not release it. We want to rank error messages that fit the first idiom over errors that fit the second. We do this by computing an additional z statistic to rank the errors within a single checked function based on (1) the number of paths within that function that contain a given `a-b` pair versus (2) the number of paths that only contain `a`. Using this additional ranking for a given `a-b` pair pushes the most likely errors to the top. Importantly, the functions at the top of the list not only are most likely to contain errors, but are also likely to contain the best examples of where the programmer conscientiously attempted to pair them; these examples can help us determine if a given `a-b` pair is valid.

Thus, errors are binned according to their a-b pair, with the bins sorted by the pair's plausibility. Within each bin, errors are then sorted by their individual error plausibility. We inspect errors by starting at the top of this list. For each a-b pair, we test the validity of the first error. If it is valid, we continue inspecting errors in that bin until the false positive rate is "too high." We then skip to the next a-b pair. We continue this process

```
/* drivers/sound/esssolo1.c:solo1_midi_release */
static int solo1_midi_release(...) {
   ...
   lock_kernel();
   if (file->f_mode & FMODE_WRITE) {
       add_wait_queue(&s->midi.owait, &wait);
       for (;;) {
          __set_current_state(TASK_INTERRUPTIBLE);
          spin_lock_irqsave(&s->lock, flags);
          count = s->midi.ocnt;
          spin_unlock_irqrestore(&s->lock, flags);
          ...
          if (file->f_flags & O_NONBLOCK) {
              remove_wait_queue(...);
              set_current_state(TASK_RUNNING);
              /* did not release lock! */
              return -EBUSY;
          }
    ...
    unlock_kernel();
    return 0;
```

Figure 5: Code that acquires the global kernel lock, but does not release it on its error path, though it does properly roll back a number of other operations. This identical error was cut and paste into a total of five device drivers.

until the number of bogus a-b pairs is also deemed "too high," at which point we stop.

Results. When we applied this checker to the Linux 2.4.1 kernel, we found 23 errors, 14 of which involved functions we had not checked in prior work, and 11 false positives.

The simplest error (and one of the most insidious errors we have seen) was in the "trident" sound driver where the global kernel lock was acquired by the call to "`lock_kernel`," but not released on an error path contained within a subsequent macro ("`VALIDATE_STATE`"):

```
drivers/sound/trident.c:trident_release:
lock_kernel();
card = state->card;
dmabuf = &state->dmabuf;
VALIDATE_STATE(state);
```

All errors were similar cases where functions were not reversed on error paths. Unusually, many of these errors were in core file system code rather than in drivers. One example happened in the `ufs` file system code that acquired a lock on the file system super block (`sb`) using the function `lock_super`, but did not release it on an error path. (This error survived until we sent a bug report two days before this paper was originally submitted.)

Figure 5 gives one of the more complex errors, which acquires the master kernel lock using `lock_kernel`, but while it rolls back a number of other operations on its error path, forgets to release this lock. This error was faithfully copied into four other device drivers.

9.3 Future Work

We are currently using machine learning techniques to automatically generate temporal rules and rule templates directly from source code. Instead of performing statistical analysis on traces, we form a general model of actions and control flow using probabilistic automata (hidden Markov models and stochastic context-free grammars) with probabilities initialized from static branch prediction. Initial results lead us to believe that this will be a profitable approach.

10 Conclusion

This paper shows how to automatically find bugs in a system without having a priori knowledge of the correctness rules the system must obey. We use simple static analyses to automatically extract programmer beliefs from the source code, and we flag belief contradictions as errors. The key benefit of this approach is that it eliminates the need to understand the system in any deep way — we know that a contradicted belief must be an error, *without having to know what the actual belief should be*.

This approach is a significant improvement over our prior work that manually specified rules to check. We now specify a general template for a rule, and allow the automatic analysis to specialize the template to the checked system. This technique drastically decreases the manual labor required to re-target our analyses to a new system, and it enables us to check rules that we had formerly found impractical.

We present two general techniques for implementing these deriving analyses, and we discuss a framework and terminology for describing them. Our first technique, internal consistency, finds errors where programmers have violated beliefs that we know they must hold. Our second technique, statistical analysis, extracts beliefs from a much noisier sample where the extracted beliefs can be either valid or coincidental.

Finally, we have shown that this approach works well on real systems code. We presented six template checkers that found hundreds of errors in recent snapshots of the Linux and OpenBSD operating systems. Many of these errors resulted in kernel patches.

11 Acknowledgments

We thank David Dill for early discussions of invariant derivation. Wilson Hsieh, Justin Haugh, Ken Ashcraft and Palash Nandy provided last minute editing assistance. We thank Evan Parker for doing the bulk of the implementation work for Section 6: he built the use-then-check and redundant-check checkers, inspected all results, and modified the C pre-processor to mark where macros were expanded. We thank the generous readers of `linux-kernel` for their feedback and support, especially Alan Cox for confirming and fixing many of our bugs. We extravagantly thank Eddie Kohler and Diane Tang, who both suggested and guided multiple last minute paper surgeries in attempts to rescue clarity. Eddie simplified and gave new insights into our framework. Diane donated several exhaustive close reads. We especially thank Mark Horowitz for creating a lab environment that allows us to focus on research rather than bureaucracy. This research was supported by DARPA contract MDA904-98-C-A933 and by a grant from the Stanford Networking Research Center.

References

[1] 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.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software (LNCS 1885, Springer)*, August/September 2000.

[3] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, pages 131–152, Spring 1996.

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

[5] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.

[6] 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.

[7] D.L. Detlefs, R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. TR SRC-159, COMPAQ SRC, December 1998.

[8] P. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sørensen, and M. Tofte. AnnoDomini: From type theory to year 2000 conversion tool. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 1–14, New York, NY, 1999.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.

[10] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[11] 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.

[12] C. Flanagan, K. Rustan, and M. Leino. Houdini, an annotation assistant for esc/java. In *Symposium of Formal Methods Europe*, pages 500–517, March 2001.

[13] R. W. Floyd. *Assigning meanings to programs*, pages 19–32. J.T. Schwartz, Ed. American Mathematical Society, 1967.

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

[15] G. Holzmann and M. Smith. Software model checking: Extracting verification models from source code. In *Invited Paper. Proc. PSTV/FORTE99 Publ. Kluwer*, 1999.

[16] G. Humpreys. Personal communication. Wasted days configuring a graphics card because of a missing error check in the driver., September 2000.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[18] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[19] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.

[20] G. Nelson. *Techniques for program verification.* Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.

[21] 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.

[22] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[23] R E Strom and S Yemini. Typestate a programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 1:157–171, January 1986.

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