Bugs as Deviant Behavior

A General Approach to Inferring Errors in Systems Code

Engler et al., Stanford University

Published at SOSP 2001

The core problem

- To find **bugs**, you need a notion of what is/isn't a bug
- In other words, you need rules for "**correct behavior**"
- In large systems, many rules are undocumented (and constantly evolving)
- Manually enumerating rules is slow, incomplete, and doesn't scale.

The core problem

- To find **bugs**, you need a notion of what is/isn't a bug
- In other words, you need rules for "**correct behavior**"
- In large systems, many rules are undocumented (and constantly evolving)
- Manually enumerating rules is slow, incomplete, and doesn't scale.

So: can we find serious bugs without writing a specification first?

Previous methods

Manual review

- Humans don't scale

Type systems

- Compiler-enforced rules about data types
- Many kernel rules are **temporal** ("must call B after A", "check error before use")

Specification-Based Checking (e.g., Formal Verification)

- Powerful, but prohibitively expensive, labor-intensive, and rarely used

Testing

- Only finds bugs on executed paths; many OS bugs are in rare paths

The question.

- How can we find what is incorrect without knowing what is correct?

The question.

- **How can we find what is incorrect without knowing what is correct?**

Answer: use **logic** and **statistics**

Key move: treat a bug as deviant behavior relative to the surrounding codebase

Two powerful tools

- **Contradictions**: if two beliefs conflict, at least one is wrong (what are beliefs?)

  $P \wedge \neg P$

- **Common behavior**: if 1000 sites do X and 1 site doesn't, the outlier is suspicious

Contradictions: more "definitive" (we **MUST** have made a mistake here)
Common behavior: a "probabilistic" approach (we **MAY** have made a mistake here)

Key vocabulary: "beliefs"

- Belief = a fact implied by code (e.g., dereference p ⇒ p is non-null)
- **MUST belief**: the programmer must hold it (directly implied)
- **MAY belief**: pattern might be real… or coincidence

The system extracts beliefs, then cross-checks them for violations.

Example: null pointer consistency (MUST)

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

- L1: **card == NULL** implies card is NULL at the true path of the check.
- L3: **card->contrnr** implies card is NOT NULL at the true path of the check.
- We got a contradiction!
- A **consistency checker** can find such errors by associating the pointer p with a belief set and flagging cases where beliefs contradict.

General Internal Inconsistency

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.

General Internal Inconsistency

In simpler words:

- Associate each slot instance (e.g. every pointer p) with a belief set.
- Code actions add beliefs (e.g., deref => not-null).
- Conditionals propagate beliefs on true/false paths.
- When paths join, merge beliefs; contradictions => bug.

Example: statistical lock inference (MAY)

```
1   lock l;
2   int a, b;
3
4   void foo() {
5       lock(l);
6       a = a + b;
7       unlock(l);
8       b = b + 1;
9   }
10  void bar() {
11      lock(l);
12      a = a + 1;
13      unlock(l);
14  }
15  void baz() {
16      a = a + 1;
17      unlock(l);
18      b = b - 1;
19      a = a / 5;
20  }
```

- **a** is used four times, three times with the lock held, and once without.
- **b** is indifferently protected with the lock: not protected twice, and protected once in a plausibly coincidental situation.
- Intuitively, **a** is much more plausibly protected by the lock than **b**.

General Statistical Analysis

Conceptually, a **statistical checker** is an internal **consistency checker** with three modifications:

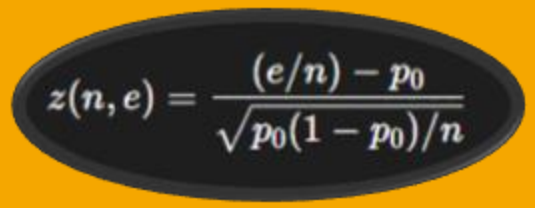
1. It applies the check to all potential slot instance combinations. I.e., it assumes that all combinations are MUST beliefs.
2. It indicates how often a specific slot instance combination was checked and how often it failed the check (errors).
3. It is augmented with a function, rank, that uses the count information above to rank the errors from all slot combinations from most to least plausible.

General Statistical Analysis

In simpler words:

For the previous statistical lock inference example, the checker would consider all variable-lock pairs (v, l) as valid instances

It would then rank them based on "errors" or "checks" of the assumed rule template according to a function (here: z statistic)



- n = population size = number of times the rule *could have been checked* ("checks")
- c = counterexamples = number of failures ("errors")
- e = examples = successful checks = n - c
- assume baseline probability p0 = 0.9 (rule holds “most of the time” by default)
- observed success rate = e/n

Simply: heavily reward patterns that have both massive sample sizes and very few errors

RECAP: how rules become checkers

- They restrict to generic rule templates

(e.g., "<a> must be paired with <b>")

- Templates have slots; concrete code elements fill slots ("slot instances")

- Checkers: find slot instances, propagate beliefs, report contradictions/violations.

Implementation

- The authors use metal, a high level state machine (SM) language for writing system specific compiler extensions
- These extensions are dynamically linked into xgcc, an extended version of the gcc. 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

```
1   sm internal_null_checker {
2     state decl any_pointer v;
3
4     /* Initial start state: match any pointer
5        compared to NULL in code, put it in a 'null'
6        state on true path, ignore it on false path. */
7     start:
8       { (v == NULL) } ==> true=v.null, false=v.stop
9       { (v != NULL) } ==> true=v.stop, false=v.null
10    ;
11
12    /* Give an error if a pointer in the null state
13       is dereferenced in code. */
14    v.null:
15      { *v } ==> { err("Dereferencing NULL ptr!"); }
16    ;
17  }
```

Implementation Intuition

- Compile each function to GCC IR (control-flow graph)
- Run metal code over the CFG (state machines)
- States encode beliefs (e.g., lock held / not held)
- Transitions triggered by events (deref, compare, return)
- Log: CHECK events and ERROR events

Case study 1: internal null consistency

Three error families:

- **Check-then-use** (null on path, later dereferenced)
- **Use-then-check** (dereferenced, later checked as if could be null)

  Rule template: "do not dereference a null pointer <p>"

- **Redundant checks** (checking when value is already known)

  Rule template: "do not test a pointer <p> whose value is known."

  Flags places where programmers are clearly confused ⇒ possible bugs

Case study 1: internal null consistency

Three error families:

- **Check-then-use** (null on path, later dereferenced)
- **Use-then-check** (dereferenced, later checked as if could be null)

  Rule template: "do not dereference a null pointer <p>"

- **Redundant checks** (checking when value is already known)

  Rule template: "do not test a pointer <p> whose value is known."

  Flags places where programmers are clearly confused ⇒ possible bugs

This checker stays CFG local

Case study 1: Results

On Linux 2.4.7:

- check-then-use: 79 bugs (26 false positives)
- use-then-check: 102 bugs (4 false positives)
- redundant-checks: 24 bugs (10 false positives)

/ drivers / video / tdfxfb.c

```
1878        fb_info.bufbase_virt = ioremap_nocache(fb_info.bufbase_phys, fb_info.bufbase_size);
1879        if(!fb_info.regbase_virt) {
1880          printk("fb: Can't remap %s framebuffer.\n", name);
1881          iounmap(fb_info.regbase_virt);
1882          return -ENXIO;
1883        }
```

Case study 2: dangerous user pointers

Kernel must **not** trust user pointers.

- **Belief A**: raw dereference ⇒ safe kernel pointer.
- **Belief B**: passed to copy_to/from_user ⇒ unsafe user pointer.
- If both beliefs hold for the same pointer ⇒ bug.

Mostly system-independent; small system-specific lists (routines to handle user pointers, dereferencing routines that should be ignored, etc)

Case study 2: dangerous user pointers

Two-pass interprocedural approach

Derivation pass:

- list of functions/parameters that pass their parameter to a routine like copy_from_user / copyin
- a list of functions/parameters that **dereference**

BUG:

- B dereferences its i-th parameter
- A passes a tainted value into B's i-th parameter.

Case study 2: Results

- OpenBSD 2.8: 18 bugs, 3 false positives
- Linux 2.4.1: 12 bugs , 16 false positives

```
/* 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);
```

Case study 3: missing failure checks

- Many kernel functions can fail (NULL pointers, error codes).
- If the return value is almost always checked, an unchecked callsite is suspicious.

Let's make a checker!

NO propagation, but cross-function aggregation (global evidence)

Functions are global symbols

Case study 3: missing failure checks

- Many kernel functions can fail (NULL pointers, error codes).
- If the return value is almost always checked, an unchecked callsite is suspicious.

Let's make a checker!

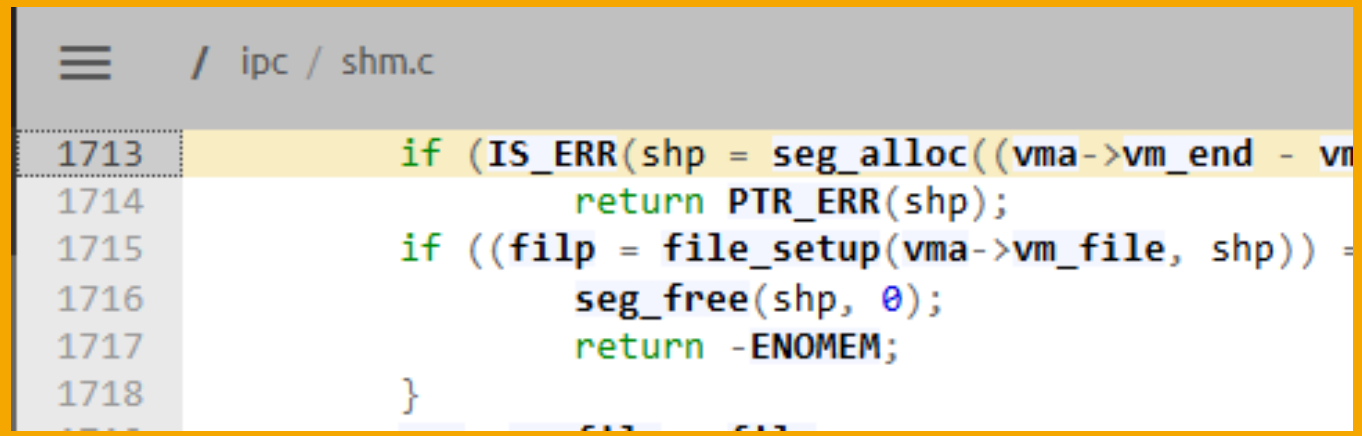
- Assume that all functions can fail.
- If the result of a function f is ignored or used without checks, the checker emits an error message.
- If the result of a function f is checked before use the checker emits a "checked" message. If f has a high ratio of check to error messages, this implies that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

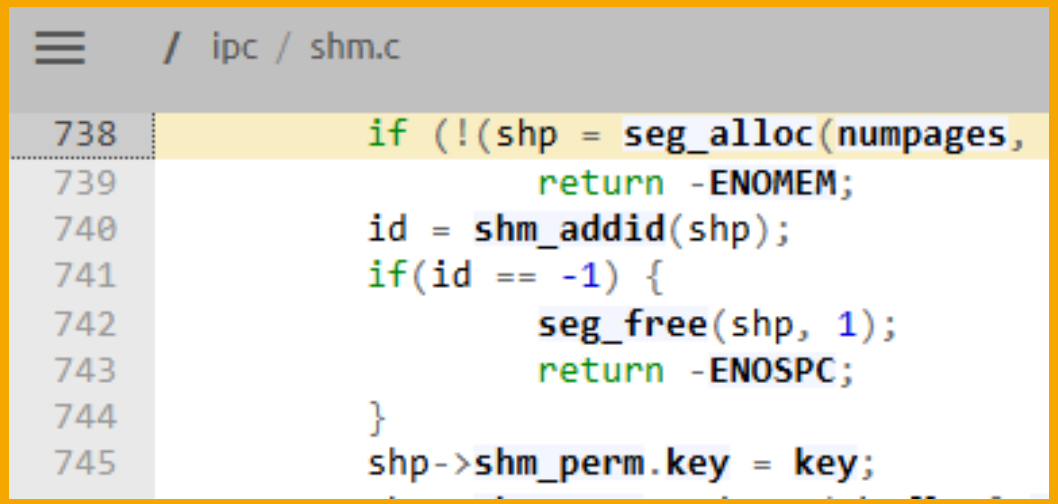
Case study 3: Results

- OpenBSD 2.8: 41 bugs, 21 false positives
- Linux 2.4.1: 154 bugs , 16 false positives

A genuinely unforeseen error type: error pointers + IS_ERR

- Some Linux code returns error codes cast as pointers
- Caller checks "== NULL" ⇒ check fails ⇒ error-pointer slips through.
- New consistency rule: if any caller uses IS_ERR for f, all callers should
- Reported: 5 such errors (plus 6 false positives) across 295 checked call sites





Case study 4: deallocation

Motivation: **ensure that freed memory is not used**

The problem? Large set of system-specific deallocation functions

Idea: infer deallocation routines automatically!

If a function's arguments are NOT used after a call, it MAY be a deallocation routine.

Use **Statistical Analysis** to find argument-function pairs that are most likely pointer-free.

Reduce the search space by only looking at functions suggestive of deallocation (containing strings like "free", "dealloc", etc)

Case study 4: Results

- Linux 2.4.1: 23 bugs , 11 false positives

/ drivers / block / cciss.c

```
if ( copy_to_user((void *) arg, &iocommand, sizeof( IOCTL_Command_struct) ) )
{
        cmd_free(NULL, c);
        if (buff != NULL)
                kfree(buff);
        return( -EFAULT);
}

if (iocommand.Request.Type.Direction == XFER_READ)
{
        /* Copy the data out of the buffer we created */
        if (copy_to_user(iocommand.buf, buff, iocommand.buf_size))
        {
                cmd_free(NULL, c);
                kfree(buff);
        }
}
cmd_free(NULL, c);
```

Case study 5: paired functions

- Goal: infer pairs (a, b) where b must follow a on all paths (like lock/unlock)
- Gather traces, rank plausible pairs based on z statistic, filter based on names suggestive of paired functions (substrings "lock", "unlock", "acquire", "release", "spl", etc.)
- Within a function: if one path misses b (often an error path) ⇒ report.

Case study 5: Results

- Linux 2.4.1: 23 bugs , 11 false positives

/ drivers / sound / trident.c

```
        lock_kernel();
        card = state->card;
        dmabuf = &state->dmabuf;
        VALIDATE_STATE(state);
```

```
#define VALIDATE_MAGIC(FOO,MAG)                                 \
({                                                              \
        if (!(FOO) || (FOO)->magic != MAG) {                    \
                printk(invalid_magic,__FUNCTION__);             \
                return -ENXIO;                                  \
        }                                                       \
})

#define VALIDATE_STATE(a) VALIDATE_MAGIC(a,TRIDENT_STATE_MAGIC)
#define VALIDATE_CARD(a) VALIDATE_MAGIC(a,TRIDENT_CARD_MAGIC)
```

Practical constraints (why false positives exist)

- Paths may be unreachable
  calls like **BUG/panic** are path cut-offs → don't report errors on those paths

- Don't let beliefs leak across abstraction barriers (**macros are not functions**)
  Annotate macro-expanded code so belief propagation can be truncated

Summary

- MUST beliefs ⇒ contradictions
  MAY beliefs ⇒ deviations ranked by z.

- Contradictions (or possible ones) are powerful; they allow us to find what is incorrect without knowing what's correct

- Statistical ranking + human inspection is an effective way to handle noise

- This technique can reveal errors you didn't know to specify

Thank you! Questions?