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

Jing Huang

Background

- **problems**

  what rules will be used to check bugs?

  - undocumented
  - a ad hoc collection of conventions
  - encoded in code

  How to find the rules?

  How to use the rules to find bugs?

Background –contd.

- **previous methods**
  - **testing and manual inspection**
    - to depend on human judgment
    - to suffer from exponential number of code paths
  - **type system**
    - To require Invasive, strenuous manual work
    - To require specific languages
  - **Specifications**
    - To suffer from missing features and over-simplification
  - **Dynamic invariant inference**
    - To dynamically monitor program execution
    - To suffer from a lmited number of code paths
    - Noise is less of concern, all value can be detect

goal and process

- To extract beliefs directly from code
- To check for violated beliefs
- To suppress noise in checking results
- To find bugs based on checking

mythology **–contd.**

- General internal consistency
  - MUST belief
    - directly observation

      To change state and observe it

    - pre- and post-conditions

      To be based on the pre- and post-condition of actions in code (non-zero)

mythology **–contd.**

- General internal consistency
  - The definition of consistency checker
    - The rule template T.
    - The valid slot instances for T.
    - The code actions that imply beliefs.
    - The rules for how beliefs combine, including the rules for contradictions.
    - The rules for belief propagation.

mythology **–contd.**

- Example for internal consistency(null pointer)

```
if (card == NULL) {
printk(KERN_ERR "capidrv-%d: . .. 7,%d!\n",
card->contrnr, id) ;
}
```

  - The rule template T.

  "do not dereference null pointer <p>,“

  - The valid slot instances for T.

  pointer <card> associated with a belief set{null,notnull,empty}

mythology **–contd.**

- Example for internal consistency(null pointer)
  - The code actions that imply beliefs.
    - Compare (line1)

      nothing directly impacts
    - Deference (line 3)

      to signal error

      to add {not null} into the belief set
  - The rules for how beliefs combine, including the rules for contradictions.

  - The rules for belief propagation.
    - Compare (line1)

      to propagate belief in true branch and false branch

mythology **–contd.**

- General statistical analysis
  - Analysis object

    MAY belief
  - Analysis goal

    to promote MAY belief to MUST belief
  - The definition of consistency checker
    - To check all potential slot instance combinations and then assume that they are MUST beliefs.
    - To indicates how often a specific slot instance combination was checked and how often it failed the check (errors).
    - To use the count information above to rank the errors from most to least plausible.

mythology **–contd.**

- General statistical analysis
  - statistical analysis method

    To filter out coincidences from MAY beliefs by observing typical behaviors

    - Z-statistics

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

n: the number of checks (the population size)

e: errors (the number of counter examples)

P0: the probability of the examples (n-e)

1-p0: the probability of the counter-examples

mythology **–contd.**

- General statistical analysis
  - To suppress noise
    - To use z-statistic to rank error from most to least credible

    - To use latent specifications to filter result and determine where and what to check

      a special function call

      a set of data types

      specific naming conventions

mythology **–contd.**

- Example for statistical analysis(lock inference)

```
lock l;           // Lock
int a, b;         // Variables potentially
                  // protected by l
void foo() {
    lock(l);      // Enter critical section
    a = a + b;    // MAY: a,b protected by l
    unlock(l);    // Exit critical section
    b = b + 1;    // MUST: b not protected by l
}
void bar() {
    lock(l);
    a = a + 1;    // MAY: a protected by l
    unlock(l);
}
void baz() {
    a = a + 1;    // MAY: a protected by l
    unlock(l);
    b = b - 1;    // MUST: b not protected by l
    a = a / 5;    // MUST: a not protected by l
}
```

mythology **–contd.**

- Example for statistical analysis(lock inference)
  - The rule template T.

    variable a must be protected by lock 1?
  - To use internal consistency and record how often the belief satisfied its rule versus gave an error.
  - To use z-statistic to analyze these counts and rank errors from most to least credible
  - To define a threshold, z-value is higher than it, we regard it as MUST belief, otherwise, we give up the template.

case study

- Internal consistency

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

  - Danger user pointer

```
/* net/atm/mpoa_proc.c */
ssize_t proc_mpc_write(struct file *file,
                       const char *buff) {
    page = (char *)__get_free_page(GFP_KERNEL);
    if (page == NULL) return -ENOMEM;
    /* [Copy user data from buff into page] */
    retval = copy_from_user(page, buff, ...);
    if (retval != 0)
        ...
    /* [Should pass page instead of buff!] */
    retval = parse_qos(buff, incoming);
}
int parse_qos(const char *buff, int len) {
    /* [Unchecked use  of buff] */
    strncpy(cmd, buff, 3);
```

case study

- Statistical analysis

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

case study

- Statistical analysis
  - Failure/IS_ERR (function <f> must be checked for failure)

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

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

```
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;
```

case study

- Statistical analysis
  - no <a> after <b> (freed memory cannot be used)

```
cut->data = k m a l l o c ( . . . ) ;          →if allocating is failed
if (!ent->data)
kfree (ent) ;
goto out ;
}
out :
return ent ;
```

Conclusion

- To automatically extract programmer beliefs from the source code, and we flag belief contradictions as errors by using statistical analysis and internal consistency.
- To automatically find bugs in a system without having a prior knowledge
- To drastically decreases the manual labor required to re-target our analyses to a new system,
- To enable us to check rules that we had formerly found impractical.

like and dislike

- **Like**
  - **To use simple techniques to find bugs**
  - **Based on z-statistic, to rank MAY beliefs**
  - **To find more types of bugs than before**
  - **To provide a lot of clear analysis for detailed cases**
- **Dislike**
  - **Too many terms and too abstract description in analysis and these terms’ definition is scattered in different parts of the paper**

Thank you!

Questions?