# Metal error checkers

# Checking for memory problems

```
sm null_checker {
   decl { scalar } sz;      // match any scalar
   decl { const int } retv; // match const ints
   decl { any_ptr } v1;     // match any ptr
   // 'state' specifies 'v' will have a state
   state decl { any_ptr } v;

// Associate allocated memory with unknown
// state until compared to null.
start, v.all:
    // set v's state on true path to "null",
    // on false path to "not_null"
    { ((v = (any)malloc(sz)) == 0) }
       ==> true=v.null, false=v.not_null
    // vice versa
  | { ((v = (any)malloc(sz)) != 0) }
       ==> true=v.not_null, false=v.null
    // unknown state until observed.
  | { v = (any)malloc(sz) } ==> v.unknown;

// Allow comparisons on variables in
// states "unknown", "null", and "not_null."
v.unknown, v.null, v.not_null:
   { (v == 0) } ==>
       true = v.null, false = v.not_null
 | { (v != 0) } ==>
       true = v.not_null, false = v.null;

// Catch error path leaks by warning when
// a non-null, non-freed variable gets to a
// return of a negative integer.
v.unknown, v.not_null: { return retv; } ==>
     { if(mgk_int_cst(retv) < 0)
         err("Error path leak!"); };

// No dereferences of null or unknown ptrs.
v.null, v.unknown: { *(any *)v } ==>
        { err("Using ptr illegally!"); };

// Allow free of all non-freed variables.
v.unknown, v.null, v.not_null:
     { free(v); } ==> v.freed;

// Check for double free and use after free.
v.freed:
     { free(v) } ==> { err("Dup free!"); }
   | { v } ==> { err("Use-after-free!"); };

// Overwriting v's value kills its state
v.all: { v = v1 } ==> v.ok;
}
```

# Memory allocation

| | Linux | | OpenBSD | |
|---|---|---|---|---|
| **Violation** | **Bug** | **False** | **Bug** | **False** |
| No check | 79 | 9 | 49 | 2 |
| Error leak | 44 | 49 | 3 | 1 |
| Use after Free | 7 | 3 | 0 | 0 |
| Underflow | 2 | 0 | 0 | 0 |
| Total | 132 | 61 | 52 | 3 |

Table 2: Error counts for Linux and OpenBSD. The checker was applied 4268 times in Linux and 464 times in OpenBSD.

# Improper blocking

| Check | Local | Global | False Pos |
|---|---|---|---|
| Interrupts | 18 | 42 | 4 |
| Spin Lock | 21 | 42 | 4 |
| Module | 22 | $\sim 53$ | $\sim 2$ |
| Total | 61 | $\sim 137$ | $\sim 10$ |

Table 3: Results for checking if kernel routines block (1) with interrupts disabled ("Interrupts"), (2) while holding a spin lock ("Spin Lock"), or (3) in a way that causes a module race ("Module"). We divide errors into whether they needed local or global analysis. Local errors were due to direct calls to blocking functions; global errors reached a blocking routine via a multi-level call chain. The global analysis results for Module are marked as approximate since they have not been manually confirmed.

# Linux synchronization checker

| Condition | Applied | Bug | False Pos |
|---|---|---|---|
| Holding lock | ~ 5400 | 29 | 113 (90) |
| Double lock | - | 1 | 3 |
| Double unlock | - | 1 | 20 (18) |
| Intr disabled | ~ 5800 | 44 (43) | 63 (54) |
| Bottom half | ~ 180 | 4 | 12 |
| Bogus flags | ~ 3200 | 4 | 49 (24) |
| Total | - | 83 (82) | 260 (201) |

Table 4: Results of running the Linux synchronization primitives checker on kernel version 2.3.99. The **Applied** column is an estimate of the number of times the check was applied. We skipped twelve warnings that were difficult to classify. The parenthesized numbers show the changes when the two files with the most false positives are ignored.

# Errors found by Metal

| Check | Errors | False Positives | Uses | LOC |
|---|---|---|---|---|
| Side-effects(§ 4.1) | 14 | 2 | 190 | 25 |
| Static assert(§ 4.2) | 5 | 0 | 1759 | 100 |
| Stack check(§ 4.3) | 10+ | 0 | 332K | 53 |
| User-ptr(§ 5.1) | 18 | 15 | 187 | 68 |
| Allocation(§ 5.2) | 184 | 64 | 4732 | 60 |
| Block(§ 6.2) | 123 | 8 | - | 131 |
| Module(§ 6.3) | ~75 | 2 | - | 133 |
| Mutex(§ 7) | 82 | 201 | 14K | 64 |
| Total | ~511 | ~292 | - | 609 |

Table 6: The results of MC-based checkers summarized over all checks. **Error** is the number of errors found, **False Positives** is the number of false positives, **Uses** is the number of times the check was applied, and **LOC** is the number of lines of *metal* code for the extension (including comments and whitespace).