

# 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

Violation	Linux		OpenBSD	
	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	~ 53	~ 2
Total	61	~ 137	~ 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	199	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	-	669

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).

# Interrupt checking

```
// Mark paths containing non-returning function as dead.
sm kill_paths local {
  decl any_fn_call call;
  decl any_args args;

  start: { call(args) } ==>
  {
    char *n = mc_identifer(call);
    if(mc_fn_is_noreturn(call))
      mc_set_path_kill(call);
    else if(n && (!strcmp(n, "panic") || !strcmp(n, "BUG")))
      mc_set_path_kill(call);
  };
}

.module macros.m // Include useful macros.

// Extension data and code: accessible from SM
// pattern matching callouts and actions.
sm_header { static int enables, disables; }

sm cli_sti_consistent local {
  decl any_expr flags;

  // Run at beginning of each function.
  init { enables = disables = 0; };

  // Run at the end of each function.
  final {
    if(!mc_nerrors() && enables != 0 && disables != 0)
      err("CLI_STI: enable %d to disable %d",
          enables, disables);
  };
}

// Pattern to match the various ways to
// disable interrupts.
pat disable =
  { cli(); }
  || { __global_cli(); }
  || { local_irq_disable(); }
  || { local_bh_disable(); }
  ;

// ... to enable interrupts.
pat enable =
  { sti(); }
  || { __global_sti(); }
  || { local_irq_enable(); }
  || { local_bh_enable(); }
  || { restore_flags(flags); }
  || { __restore_flags(flags); }
  ;

start:
  disable ==> disabled, { disables++; }
  | enable ==> enabled, { enables++; }
  ;
disabled:
  enable ==> start,
  { note("CLI_STI: reversed disable [SUCCESS]"); }
  | disable ==> stop,
  { err("CLI_STI: double disable [FAIL]"); }
  ;
enabled:
  disable ==> start,
  { note("CLI_STI: reversed enable [SUCCESS]"); }
  | enable ==> stop,
  { err("CLI_STI: double enable [FAIL]"); }
  ;
disabled, enabled: $end_of_path$ ==>
  { err("CLI_STI: did not reverse [FAIL]"); }
  ;
}
```

## Issues/ideas

- Mini-language for writing checkers
- evaluate state machines along each edge
- Interprocedural analysis
- False positives
- Ranking of results

## Mini-language for writing checkers

- pattern matching against AST
  - before macro expansion?
- Selected variables/patterns can have associated state
- state transitions based on pattern matching
- Can also evaluate arbitrary C code

## evaluate state machines along each edge

- enumerate all paths from entry to exit
- Perform DFS of CFG
- Don't recurse if finite state matches previous state at node
  - also matching information used for impossible paths, ...?

## Interprocedural analysis

- earlier papers did little or no interprocedural analysis
  - recent addition
- doesn't seem to handle function pointers or polymorphic method invocation
- Examine all functions and build call graph
- Functions with no callers are roots
  - check all roots
- Refine/restore information across procedure boundaries
  - Don't track global information?
- Cache and reuse results

## False positives

- impossible paths
- killing expressions
  - what happens to tracked state of  $a[i]$  when  $i$  is modified
- synonyms

## Ranking of results

- generic ranking
- statistical ranking of uncertain hypotheses