How to Write System-specific, Static Checkers in Metal

Benjamin Chelf, Dawson Engler, and Seth Hallem
Stanford University

1. INTRODUCTION

This paper gives an overview of the *metal* language, which we have designed to make it easy to construct system-specific, static analyses. We call these analyses *extensions* because they act as the input to a generic analysis engine that runs the static analysis over a given source base. We also interchangeably refer to them as *checkers* because they check that a user-specified property holds in the source base and report any violations of that property. Note that checkers may not detect all violations of a specified property. Their goal is to find as many violations as possible with a minimum of false positives.

We describe seven checkers in total; most are less than 50 lines of code, but, in aggregate, find hundreds of errors in a typical large system. In addition to language details, we give a feel for how to build good checkers: exploiting high-level knowledge, ranking errors, suppressing false positives. Further, we show how to write checkers that extract rules to check from the source code itself without the assistance of the programmer.

The common thread among these analyses is that they all exploit the fact that many abstract program restrictions map clearly to source code actions [5]. While *metal* extensions are executed much like a traditional dataflow analysis, they can easily be augmented in ways outside the scope of traditional approaches, such as using statistical analysis to discover rules [6].

To check a rule, an extension does two things: (1) recognizes interesting source code actions relevant to a given rule and (2) checks that these actions satisfy some rule-specific constraints. We refer to the former as the alphabet of the extension because it defines a "language" that is the interface between the extension and the code that it is analyzing. The latter is most often a finite state machine that defines the legal and illegal sequences in the alphabet. Because *metal* does allow flexibility in several stylized ways, extensions are not limited to finite-state properties although the majority of the properties we check are temporal safety properties.

Metal extensions are executed by the analysis engine, *xgcc*. *xgcc* can execute both interprocedural and intraprocedural analyses. All analyses are flow-sensitive and, in the interprocedural case, context-sensitive. The algorithm we use to execute *metal* extensions shares many properties with the standard dataflow analysis techniques in the literature [1, 12], although our techniques do diverge in many ways that we describe in detail in [10].

PASTE '02, November 18–19, 2002, Charleston, SC, USA.

2. METAL CRASH COURSE

This section illustrates the main features of *metal* using three example checkers. We discuss the main features of all *metal* extensions, then we discuss two simple checkers in detail. We finish by revisiting some of the components in the extension, and by taking a look at a more complex extension.

All extensions have two main parts: the language of discourse and the state machine (SM). The language of discourse defines the interesting features in the code. This language is defined using *metal* patterns, which are either templates written in the source language or calls to boolean-valued C functions that identify the interesting constructs. The SM part of the extension follows one of two templates: track program-wide properties or track object-specific properties. We call extensions that follow the former template *global* state machines, and those that follow the latter *variable-specific* state machines. (The term *variable-specific* is something of a misnomer because these extensions can track arbitrary expressions in addition to variables.)

2.1 Global Checkers: Linux Interrupts

A global extension tracks transitions in a single state machine that corresponds to a program-wide property such as "interrupts are disabled." A global extension defines the SM as a list of states, each of which guards a list of transitions that can potentially execute when the SM is in the guarding state. The first state in this list of states is implicitly defined as the initial state of the SM. Each transition is specified with a pattern, an optional destination state, and an optional C code *action*. The pattern identifies a source construct that, when encountered in the source base, will cause the transition to execute. The destination state provides a new state for the SM; if it is omitted, the SM state remains the same after the transition executes. The action is an arbitrary block of C code that most often will either execute a state transition or print a message.

Figure 1 shows a simplified global checker for Linux that warns when disabled interrupts are not restored. A call to `cli()` disables interrupts; a call to `sti()` enables them.

```
.module macros.m

sm cli_sti {
  enabled:
     { cli(); } ==> disabled
   | { sti(); } ==> stop,
        { err("Double sti"); }
    ;
  disabled:
     { sti(); } ==> enabled
   | { cli(); } ==> stop,
        { err("Double cli"); }
   | $end_of_path$ ==>
        { err("Did not reverse"); }
    ;
}
```

Figure 1: Interprocedural interrupt checker: uses a global SM to track whether disabled interrupts are re-enabled.

```
int err(void) {
   if(random())
      sti();   // goes to enabled state
}
int err_caller() {
   cli(); // goes to disabled state
   err(); // returns in two states
          //    { enabled, disabled }
}         // end-of-path: ERROR:Did not reverse

int ok(void) {
   cli();  // goes to disabled state
}
int ok_caller() {
   ok();  // returns in disabled state
   sti(); // goes to enabled state
}         // end of path: success
```

Figure 2: Example code and transitions for interrupt checker.

These two functions are the alphabet of the extension. Conceptually, the extension finds violations by checking that each call to disable interrupts has a matching enable call on all outgoing paths. As refinements, the extension warns of duplicate calls to these functions or non-sequitur calls (e.g., re-enabling without disabling). Section 2.5 describes a more sophisticated version.

The extension tracks the interrupt status using two states, `enabled` (the initial state) and `disabled`. The first state has two transitions. The first, actionless, transition triggers when a call to `cli()` is identified in the source code. It changes the SM state to `disabled`. The second transition triggers on a call to `sti()`, prints an error message, and transitions to the special `stop` state thereby stopping the analysis of the current path through the program. If no pattern matches, the SM remains in the same state and continues down the current code path. The three transition rules for the `disabled` state are similar. The one novelty is the pattern `$end_of_path$`, which triggers a transition when a program path ends.

Metal extensions are applied depth-first to the control flow graph (CFG) for a function, which is computed from the abstract syntax tree (AST). Applying an extension depth-first means that it is applied to each program point down a single path through the CFG. Each program point is a single node in the AST. After analyzing a single path through the CFG, we backtrack to the last branch point, reset the state of the state machine to the state at that branch point, and resume the analysis at a different successor from that point. When the analysis encounters a function call, it retrieves the CFG for the callee and begins analyzing the callee. The analysis does not return to the caller until all paths through the callee are analyzed (see [10]).

The analysis uses a fixed-point computation to determine when to stop traversing loops in the CFG. The details of this computation are described in [10]. The practical implication of this fixed point computation is that as long as the state machine is deterministic, the extension will analyze each program point in every possible state reachable at that program point.[1]

In the case of the interrupt checker, the analysis engine transparently propagates the current global interrupt state across procedure calls, and propagates any state changes back to the caller. The extension will find one error in Figure 2. It will traverse the two (trivial) call trees: the first starting with the entry function `err_caller`, reports an error; the second, starting with the entry function `ok_caller`, passes the check.

2.2 Variable-Specific Checkers: Null Pointers

A variable-specific extension is comprised of a series of state machines, each of which tracks the state attached to a single program object. "Program object" is broadly defined to include any expression that has an associated state, including structure fields, arithmetic expressions, pointer dereferences, etc. Typical examples of variable-specific properties include "freed pointers should not be used" and "null pointers should not be dereferenced."

A variable-specific extension is logically separated into two parts: creation transitions and state transitions. The creation transitions tell the analysis when to begin tracking a new object and are guarded by the identifier "`start`." They are active throughout the analysis, although a creation transition is suppressed if we are already tracking a state machine associated with the created variable. The state transitions describe the state machine that each program object must follow. They are guarded by *bound* states.

Figure 3 shows a simple variable-specific extension (the null checker) that flags when a pointer returned from the Linux memory allocator, `kmalloc`, is used without being checked against `NULL`. The extension will find one error in Figure 4.

Each variable-specific extension defines a single typed identifier that is used to refer to the program object that a single SM is tracking. In the null checker, the identifier `v` is used to refer to the tracked object. The keyword `state` in the declaration of `v` on line 2 indicates that `v` will refer to the tracked object. The states in a variable-specific state machine are bound to the identifier that refers to the tracked object. For example, in the null checker, the states are `unknown`, `stop`, and `null`, each of which is bound to the identifier `v` using the syntax `v.unknown`, `v.stop`, and `v.null` respectively.

When a variable is in the `unknown` state, it means that the value of that variable is the result of a call to the allocation

[1] There are some subtleties due to recursion that make this statement false under the conditions described in [10].

```
1 : sm null_checker local {
2 :   state killvars decl any_pointer v;
3 :   decl any_expr x,y;
4 :
5 :   // Various uses.
6 :   pat use = { *(any *)v }
7 :           || { memset(v, x, y) }
8 :           || { v + x }
9 :           || { v - x }
10:           ;
11:   start: { v = kmalloc(x,y) } ==> v.unknown
12:     ;
13:   v.unknown:
14:      { (v == NULL) } ==> true=v.null, false=v.stop
15:    | { (v != NULL) } ==> true=v.stop, false=v.null
16:    ;
17:   v.null, v.unknown: use ==> v.stop,
18:      { v_err("NULL", v, "Using \"$name\" illegally!"); }
19:    ;
20: }
```

Figure 3: Null Checker: interprocedurally tracks whether pointers returned by the Linux allocator are then used without being checked against `NULL`.

```
1 : struct foo { int *p; };
2 :
3 : // p->p and q->p are in unknown state.
4 : void contrived(struct foo *p, struct foo *q) {
5 :   if(q->p)      // Null check: q->p goes to stop.
6 :       *q->p;    // OK.
7 :   else
8 :      *q->p;     // ERROR: q->p is null
9 :   *p->p;        // ERROR: p->p is unknown
10: }
11:
12: void contrived_caller(int x, struct foo *p, struct foo *q) {
13:    // Put "p->p" in unknown state.
14:    p->p = kmalloc(sizeof *p->p,GFP_KERNEL);
15:    if(x) {
16:      p = q; // Kill p, thus kills "p->p".
17:      *p->p; // No error
18:    } else {
19:        // Put "q->p" in uknown state.
20:        q->p = kmalloc(sizeof *p->p,GFP_KERNEL);
21:        contrived(p, q);
22:    }
23: }
```

Figure 4: Contrived null checker example code. Because the *xgcc* system does flow and context sensitive interprocedural analysis the null checker from Figure 3 will flag errors on lines 8 and 9.

routine `kmalloc`, and we do not know if that call returned `NULL` or not. If the tracked variable is in the `null` state, it means that we know that the value of that variable is `NULL` along the current path. Finally, if the variable is in the `stop` state, it means that we know the value is definitely not `NULL` along the current path and we can stop tracking that variable.

The null checker's alphabet includes three types of code constructs: allocations, comparisons, and uses. Allocations are identified in the null checker with the pattern "{`v = kmalloc(x,y)`}." The transition on line 11 specifies that whenever a pointer variable "`v`" is assigned the result of a call to "`kmalloc`", a new state machine should be created to track the returned result. In addition, the identifier "`v.unknown`" on the right side of the "`==>`" operator indicates that this tracked object should begin in the `unknown` state. Note that the identifier `v` is declared on line 2 with the type `any_pointer`. This means that the pattern will match any calls to `kmalloc` where the return value is captured, regardless of the base type of the pointer variable that captures that returned value.

`kmalloc` accepts two additional arguments, one specifying the size of the allocation, and the other specifying whether the allocation can block or not (i.e., are page faults allowed). Since `kmalloc` can return `NULL` regardless of the size of the allocation or whether or not the call can block, we would like to match calls to `kmalloc` with any arbitrary expressions as its arguments. The two identifiers `x` and `y` declared on line 3 provide exactly this functionality. They are declared with the type `any_expr`, which means that when they appear in a pattern, they are effectively acting as a placeholder for an arbitrary expression. We call these placeholders *hole* variables, which we discuss in more detail in Section 2.3.

The comparison patterns identify cases where the variable is checked against `NULL`, and they use a *path-specific* transition to indicate a different state change depending on the result of the comparison. For example, if the source code checks equality of the tracked variable and `NULL`, the pattern "{`v == NULL`}" will identify this comparison, thus causing the transition on line 14 to execute. The transition specifies that on the true path from the comparison, the SM should transition to the `null` state, while on the false path, the SM should transition to the `stop` state.

Finally, the use patterns indicate illegal uses of `NULL` pointers. These include calls to `memset`, pointer dereferences, and arithmetic operations on pointers. Although this last operation is not technically incorrect, the result is probably not what the programmer intended. Note that much like `kmalloc`, `memset` takes two additional arguments whose particular values are irrelevant to the transition. Thus, we again use the placeholders `x` and `y` to indicate that any arbitrary expression can appear in those slots in the pattern. The transition on lines 17 and 18 specifies that if a pointer that is either known to be `NULL` or whose value is the unchecked result of a call to the allocator is used in an illegal way, an error message should be printed. The error message is specified in an action that calls the error reporting function `v_err` with a name for the extension as the first argument, the AST for the tracked variable as the second argument, and a format for the message as the third argument. The name of the tracked variable is substituted for the directive "`$name`."

The qualifier `killvars` that appears in the declaration of `v` on line 2 indicates that if the value of the tracked variable is updated so that we can no longer rely on the state of the SM to accurately reflect that value, we should stop tracking that variable. Note that this computation is a syntactic approximation; it does not include redefinitions through aliases.

2.3 Patterns: the Alphabet of an Extension

Metal patterns provide a simple way for extensions to identify source actions that are relevant to a particular rule. Patterns are written in an extended version of the source language (C) and can specify almost arbitrary language constructs such as declarations, expressions, and statements. Patterns are easy to use because they syntactically mirror

Hole Type	Matches
Any C type	any expression of that type
`any_expr`	any legal expression
`any_scalar`	any scalar value (int, float, etc.)
`any_pointer`	any pointer of any type
`any_arguments`	any argument list
`any_fn_call`	any function call

Table 1: Hole types and their meanings.

the source constructs that they are intended to match.

A *base* pattern in *metal* is a bracketed code fragment written in our augmented version of C. Base patterns can be composed with the logical connectives `&&` and `||`. The simplest base patterns in *metal* syntactically match the code that the extension wishes to recognize. Because we match ASTs, spaces and other lexical artifacts do not interfere with matching. For example, the base pattern {`rand()`} will match all calls to the `rand` function.

A simple pattern could not, for example, match all pointer dereferences because each dereference refers to a different pointer. The pattern on line 6 in the null checker matches all dereferences with a *metal hole* variable. Any *metal* variable declared with the keyword `decl` is a hole variable. Hole variables let patterns contain positions where any source construct of the appropriate type will match.

Hole variables in *metal* must be typed. If a hole variable is assigned a C type, the hole can be "filled" by any expression of that type. If we want to match all pointer dereferences, though, we cannot assign `v` any single C type. *Metal* introduces new *meta* types that broaden holes to an entire class of related types. The hole variable `v` is declared with the meta type `any_pointer`, which matches pointers to storage of any type. Table 1 lists the hole types and their meanings.

If the same hole variable appears multiple times in a pattern, each appearance must contain equivalent ASTs. For example, the pattern {`foo(x,x)`} matches calls of the form `foo(0,0)` and `foo(a[i],a[i])`, but not `foo(0,1)`.

A hole variable used within an action (as opposed to a pattern) refers to the AST node that matches the hole. Thus, the use of `v` on line 18 in the null checker refers to the AST for the tracked variable.

Callouts let programmers extend the matching language to express unanticipated or linguistically awkward features by writing boolean expressions in C code that determine whether a match occurs. Callouts are identified syntactically by appending the prefix `$` to a base pattern.

The degenerate callouts, `$`{0} and `$`{1}, match nothing and everything respectively. Callouts are most often used as a conjunct that refines a more general pattern. For example,

```
{ fn(args) } && ${ mc_is_call_to(fn, "gets") }
```

refines a pattern that matches all function calls to one that only matches calls to `gets`. The variable `fn` is a hole variable of type `any_fn_call`, and the variable `args` is a hole with type `any_arguments`. This pattern could have been written as literal C code as well.

Used alone, callout functions can only refer to the current program point, `mc_stmt`, and any global state either within the extension or within *xgcc*. Used as a conjunct or disjunct with other patterns, the callout can refer to the ASTs matching the hole variables used in these patterns as arguments (see `fn` in the example above). *xgcc* provides an extensive library of functions useful as callouts.

Legal patterns can specify any C expression or statement (including loops, conditionals, or switch statements) with two restrictions. First, all identifiers in the pattern must be either hole variables defined in the extension or legal names in the scope of the code base being checked. Second, the C constructs used in the pattern must compile in isolation. Example illegal patterns include a single case arm without any enclosing switch statement; an isolated break; etc. All of these constructs can be matched with a callout.

2.4 Metal Transitions

At each program point encountered during the analysis, the extension iterates over all transitions that are guarded by the current SM state and executes the first transition that applies. A transition applies when its pattern matches at the current program point, and when there are no other transitions that *subsume* that transition.

One transition is subsumed by another if there is another transition that matches at an ancestor of the current program point in the AST. For example, suppose an extension includes the pattern "{`v`}," where `v` is a hole variable that matches any expression. Instead of matching at every expression in the program, the extension will only match top-level expressions. For example, in the assignment statement `x = y + z`, the extension will only match once at the expression corresponding to the entire assignment rather than matching at the expressions `y`, `z`, `y + z`, and `x`. Since not all extensions want the subsumption feature, it is enabled with the qualifier `subsume` placed after the SM declaration (on the line with the keyword `sm`, after the extension name).

Transitions can include C code actions that execute whenever the transition executes. Actions are one way that an extension can extend the basic SM abstraction. C code actions allow the extension to perform arbitrary computations whenever a transition executes. The main types of actions that we have found useful are those that perform error reporting and those that enhance the analysis machinery. In the null checker, we saw an action that reports an error. The remaining examples in this paper illustrate some additional uses for these actions.

2.5 The Rest of Metal : An Interrupt Checker

While the interprocedural interrupt checker is simple, an intraprocedural one can be much faster, both during analysis and during error inspection. Local errors often take seconds to inspect, whereas interprocedural errors can take minutes of exhausting reasoning. What we commonly do is craft a local checker to find as many errors as possible. If more depth is required, an interprocedural checker is used.

The checker in Figure 5 is an intraprocedural interrupt checker that illustrates many of the remaining *metal* features. It detects interrupt handling mistakes by finding violations of the following two properties:

1. Consistency: If one path first disables (enables) interrupts, no other path first enables (disables) them. For example, in the function `err2` in Figure 6, the true branch from the `if` statement first disables then enables interrupts, while the false branch first enables, then disables interrupts. The checker will, thus, flag an error in this function.

   The checker tracks this property by maintaining two counters that track the number of paths that start by

```
// 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_identifier(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]"); }
   ;
}
```

Figure 5: Checker that determines if interrupts are enabled and disabled consistently and idempotently.

```
int enable_disable_ok(int x) {
   cli();
   if(x != 0)
      sti(); // reversed disable [SUCCESS]
   else if(x > 0)
      sti(); // reversed disable [SUCCESS]
   else
      // No error: kill paths prunes path.
      panic("impossible value");
}
int err1(int x) {
   cli();
   if(x)
      sti(); // reversed disable [SUCCESS]
   else
      return -1; // did not reverse [FAIL]
}
// ERROR:CLI_STI: enable 1 to disable 1
int err2(int x) {
    if(x) {
      cli();
      sti(); // reversed disable [SUCCESS]
    } else {
      sti();
      cli(); // reversed enable [SUCCESS]
    }
}
int err3(int x) {
    if(x) {
      cli(); // did not reverse [FAIL]
    } else {
      sti();
      cli(); // reversed enable [SUCCESS]
    }
}
int err4(void) {
    cli(); // did not reverse [FAIL]
}
```

Figure 6: Example code with an assortment of disable and enable errors.

disabling or enabling interrupts respectively. After analyzing a function, at most one of these counters should be non-zero.

2. Idempotence: when the function exits, interrupts are at the same level as they were when it began executing. The function `err1` in Figure 6 shows a violation of this property. At the exit from this function, interrupts are either disabled or enabled depending on which path is executed. Assuming interrupts are initially enabled (otherwise we have a double disable error), the exit state is not the same as the entry state along all paths. The functions `err3` and `err4` also violate this property.

   The checker tracks idempotency by checking that any path that disables interrupts subsequently re-enables them or, conversely, if it enables them that it subsequently disables them. As part of this tracking it also flags redundant enable or disable operations.

Interrupt handling mistakes crash machines. Thus, code tends to manipulate interrupts in stylized, simplistic patterns. While overly conservative, this checker fits the most common idioms. This example, like many others, illustrates how checkers benefit from vetting human level operations: code that is easy for humans to understand tends to be easy for checkers to check.

The checker illustrates several syntactic features of *metal*:

1. `local`: Tells *xgcc* to run the extension intraprocedurally.

2. `pat`: The checker uses two *metal* pattern variables, each of which names a collection of patterns: `disable` represents the patterns that identify ways of disabling interrupts; `enable` represents the patterns that identify ways of enabling interrupts.

3. `init { ... };` : The code in the `init` block is run at the beginning of each function. The checker uses it to initialize its counting variables before the analysis starts.

4. `final { ... };` : The code in the `final` block is run at the end of each function. The extension uses it to check if at most one of its two counts are non-zero and gives an error message otherwise.

5. `sm_header { ... }` : Extensions can place arbitrary code and data in the header section, which can be accessed by the extension's callouts and actions. In the interrupt checker, the actions in the two transitions guarded by the `start` state increment the counters used to track the consistency property.

6. `.module`: This command imports another extension into the current one. This extension imports the file `macros.m`, which defines the macros `err` and `note` in an `sm_header` block.

Composition: Extensions can be composed. They are run in the order they are given in the extension file. An extension can use the results of previous extensions in its own analysis. A common pattern is for an extension to use *xgcc*'s internal interface to annotate the ASTs with arbitrary data values. Subsequent extensions can retrieve and use these values.

The `kill_paths` extension defined above the interrupt extension uses the composition feature to tell *xgcc* which paths are dead because of calls to functions that cannot return. The extension matches all function calls and, if they are annotated with the GCC attribute "no-return" or are calls to `panic` or `BUG` (both reboot the machine), marks them using the *xgcc* function `mc_set_path_kill`. If *xgcc* encounters a marked call while executing the interrupt checker, it will stop traversing the current path so that no errors are reported along paths that pass through one of these calls.

Statistical ranking: Checkers can give false positives. They may be overly conservative, use approximate analysis, or check a rule that does not apply in some contexts. We use statistical analysis to counter these false reports.

We rank errors from most to least probable by counting the number of times a given check was successful versus how many times it was not. The interrupt checker shows a very crude method. First, it emits a "`SUCCESS`" message whenever the checker goes back to a clean interrupt state and a "`FAIL`" message when either a redundant operation occurs, or the idempotency property is violated. Second, a post-processing script counts the number of `SUCCESS` and `FAIL` messages for each function, and the errors in functions with many successes and few failures are ranked above those with many failures. In Figure 6, this method ranks the errors in `err2` first (two successes, no failures), then `err1` and `err3` (one success, one failure) and finally `err4` (one failure).

The underlying observation is that the most reliable error reports are based on analysis decisions that (1) lead to few violations of a property and (2) lead to many successful checks of that property.

3. CONSISTENCY CHECKING

We often do not know the actual state of the system at a given point in the analysis. However, in some instances, we can use the source code to determine what the programmer *believes* the system state to be and perform consistency checks on these beliefs. The following code illustrates some of the types of beliefs that we can infer:

```
    // action     some possible beliefs
1:  z = *p;     // p != null
2:  free(p);    // p is heap allocated
                // p will not be used
3:  unlock(l); // l was locked.
4:  w = x / z; // z != 0
                // w, x, z not guarded by l
```

Assuming the programmer does not want their code to crash, we can infer from line 1 that the programmer believes that `p` is not null. Similarly, from line 2 we infer the belief that `p` is a valid, heap-allocated pointer and will not be used subsequently. Line 3 demonstrates a belief that `l` was locked prior to that statement. In line 4, the programmer clearly believes that `z` is not 0 as it is used in the denominator of a division. Also, more subtly, we may infer that accesses to the variables `w`, `x`, and `z` do not need to be guarded by the lock `l` since it is unlocked prior to the use of these variables.

A code action generally implies its pre- and post-conditions as beliefs. We call the beliefs shown above MUST beliefs, since we know that the programmer must have them. The key feature of belief checking 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. To illustrate, consider the following code snippet.

```
1:  z = *p;
2:  if (p == NULL)
    ...
```

Line 1 demonstrates a belief that p is not null. However, in line 2, the programmer believes that p *could* be null since she checks its value. Although we do not know at this point which belief is correct, we do know that one of them must be incorrect. Either the dereference on line 1 may crash the system or the check on line 2 is redundant.

This section shows a simple example of belief checking for null pointers. The next section describes how to reason about MAY beliefs, which are inferred beliefs that the programmer may hold, but the evidence from which we inferred the belief may also be coincidental.

The null consistency checker warns if a programmer dereferences a pointer that she must believe is `NULL`. The checker infers beliefs from two code actions:

1. A dereference of a pointer, `p`, implies a belief that `p` is not null.

2. A pointer checked against `NULL` implies a belief that `p` is null on one path from the branch and non-null on the other path (which path implies which belief depends on whether the comparison is `==` or `!=`).

```
1 : .module macros.m
2 : .module kill-paths.m
3 :
4 : sm internal_null_checker subsume {
5 :    state killvars decl any_pointer v;
6 :    decl any_pointer v1;
7 :
8 :    // null comparisions
9 :    pat null_comp =
10:  { ((v = v1) == NULL) }
11:      || { (v == NULL) }
12:      ;
13:    // non-null comparisions
14:    pat not_null_comp =
15:  { ((v = v1) != NULL) }
16:      || { (v != NULL) }
17:      ;
18:
19:    // Only track pointers directly involved in a
20:    // comparision rather than all pointers.
21:    start, v.null:
22:       not_null_comp && ${ !mc_in_macro(v) } ==>
23:         true = v.stop, false = v.null
24:     | null_comp && ${ !mc_in_macro(v) } ==>
25:         true = v.null, false = v.stop
26:     ;
27:    v.null: { *(any*)v } ==> v.stop,
28:     { v_err("NULL", v, "Using \"$name\" illegally!"); }
29:     ;
30: }
```

Figure 7: Null Consistency Checker: interprocedurally tracks whether pointers checked against null are used.

As discussed above, if the inferred beliefs contradict, we know that there is an error in the source. Figure 7 shows the checker's implementation. For example, the transition on line 24 transitions the tracked object to the `null` state on the "true" path from an equality comparison, `v==NULL`, and prints an error message if it subsequently encounters a dereference along that path. This checker could also look for pointers that are dereferenced and then compared to `NULL`, but the example omits this case for brevity.

One important feature of this checker is that we ignore comparisons to `NULL` that appear inside of macro expansions. Macros may perform a comparison that is irrelevant to the context in which the macro is invoked. Thus, comparisons to `NULL` within a macro should not be viewed as a programmer's belief at the program point where the macro is used. Often, beliefs must be prevented from violating abstraction boundaries. In this case, we draw the abstraction line by using the callout "`mc_in_macro(v)`," which returns true if the use of `v` is inside of a macro.

4. STATISTICAL INFERENCE

In both the null checker from Figure 3 and the interrupt checker from Figure 5, the extensions' alphabets were hardwired into the source of the extension. In the former, pointers returned by "`kmalloc`" were specified as the pointers the analysis should track. In the latter, various interrupt enabling and disabling calls, such as "`cli`" and "`sti`", were specified as interesting source actions. This section demonstrates how to automatically infer properties to check from the source code itself rather than requiring it to be hardwired into extension.

Such inference eliminates a significant bottleneck to checking large systems. Knowing what to check usually determines how many errors you can find. Manually extracting rules from a morass of missing or incorrect documentation is mindnumbingly tedious. Even when the rules are known, rule inference acts as a safety net to catch missed checkable rules. (E.g., pointers from Linux's `vmalloc` can also be null.)

The quickest way to see how to use statistical inference is by example. How can we infer which functions can return null pointers? Count the number of times the programmer compares the result against `NULL` versus the number of times the programmer uses the result without any comparison. The higher the ratio of checks to uses-without-checks, the more likely the function must be checked. How do we determine if two functions `a()` and `b()` must be paired? Count the number of times `a` appears with `b` versus the number of times each function appears alone. Functions that must be paired will have a high ratio of paired calls to unpaired calls.

The core idea behind the approach is that to determine whether a rule applies or not, we assume that it does and then count the number of times the code we are analyzing follows the rule (successes) versus the number of times the code does not (failures). The larger the skew in evidence, the more likely that the rule is a valid one.

The main question to answer is how to weigh the evidence. For example, if `a()` is paired with `b()` 9 times out of 10, how much more or less convincing is that evidence than if it was paired 90 times out of 100? A simple ratio of successes to failures does not work well because it ignores the sample size — a 9 to 10 ratio would be considered the same as a 90 to 100 even though the latter is subjectively much stronger evidence.

We use the ideas of "hypothesis testing [9]" to weigh the evidence. We can view these questions as "binary trials:" independent events that have exactly one of two discrete outcomes. Such trials occur in many settings — the success or failure of administering a drug or whether a coin came up heads or tails. To weigh such evidence, we can use the binomial formula, which computes the probability that an event had k successes out of n attempts given that the probability of success is p:

$$\binom{n}{k} \times p^k (1-p)^{n-k}$$

Intuitively, we expect that for a large number of trials, the ratio k/n should approach p and, if it does not, that this is strong evidence that the true probability is not p. Conversely, for a small number of trials, it is not surprising if k/n is far from p. (A degenerate example is a single toss of a fair coin: the frequency of heads will be 0 or 1, while the expected ratio is .5). The expected range of the divergence can be quantified using the standard deviation, which for the binomial formula is given by $\sigma = \sqrt{p \times (1-p)/n}$. The standard deviation goes to zero as n increases to infinity (we expect k/n to converge to p given an infinite number of trials).

The following measurement computes how many standard deviations away the observed ratio of success to failures is from the expected ratio for the given number of trials:

$$z = (k/n - p)/\sqrt{p \times (1-p)/n}$$

As the number of standard deviations increases, the im-

```
sm null_checker_stat local {
  state killvars decl any_pointer v;
  decl any_fn_call call;
  decl any_args args;
  decl any_expr x, y;

  // Put any pointer returned by a function in
  // unknown state and record function name in
  // data field.
  start:
     { v = call(args) } ==> v.unknown,
       { mc_v_set_data(v, mc_identifier(call)); }
   ;
  v.unknown:
    { (v == NULL) } || { (v != NULL) } ==> v.stop,
       { v_note("NULL_STAT", v,
             "Checking ptr [SUCCESS=$data]"); }
   | { *(any *)v } || { memset(v, x, y) } ==> v.stop,
       { v_err("NULL_STAT", v,
           "Using \"$name\" illegally! [FAIL=$data]"); }
   ;
}
```

Figure 8: Statistical checker to infer if a function can return a null pointer.

probability of the event does as well. This normalized measurement allows us to rank different sample sizes with different ratios from most to least probable. We do so by counting the number of successes and failures for a given trial, and ranking the failures (the errors) using the computed z value above. We call this process *z-ranking*.

The remaining question is what value to use for p. Since we assume programmers are usually right, we set $p \geq .8$ depending on how harshly we want to penalize mistakes. For example, a value of .8 corresponds to one mistake out of five attempts. Error rates equal to this will have a z value of zero; error rates better than it will have a positive value (they are a positive number of standard deviations from p); and error rates worse will have a negative value.

4.1 Inferring Routines that Return NULL

Figure 8 shows a statistical checker that infers which functions can return null. It closely resembles the null checker from Figure 3. The two main differences are:

1. The statistical checker tracks pointers returned by any routine rather than just pointers returned from `kmalloc`. The name of the function assigned to the pointer is extracted using the *xgcc* routine `mc_identifier(call)` and then stored in the "data" part of the SM state for `v` using the routine "`mc_v_set_data`." The data part of the SM state is one of the ways in which *metal* allows the user to extend the state space in arbitrary, potentially infinite ways. The data value is treated as an opaque handle, and is concatenated with the named SM states (e.g., null, unknown, stop) to form the actual SM state.

2. The checker emits a `FAIL` message when a pointer in the `unknown` state is used. It emits a `SUCCESS` message when a pointer in the `ok` (checked) state is used. The `$data` part of the output message will be replaced with the data part of `v`'s state (i.e., the function name).

The statistical analysis requires the following steps:

```
void v_contrived(int *p, int *q) {
   q = malloc(sizeof *q);
   // Checking ptr [SUCCESS=malloc]
   if(!q)
      return;
   p = malloc(sizeof *p);
   // Using "p" illegally! [FAIL=malloc]
   memset(p, 0, sizeof *p);
   p = foo();
   *p; // Using "p" illegally! [FAIL=foo]
   q = foo();
   *q; // Using "q" illegally! [FAIL=foo]
}
```

Figure 9: Example code for the null stat checker. The error message for `malloc` will be ranked above the two for `foo`, since `malloc` has one success and one failure, while `foo` has no successes and two failures.

1. Run the extension over the code.

2. Count the number of `SUCCESS` and `FAIL` messages for each checked function and use these counts to compute a z value for the function.

3. Sort the error messages (i.e., those with `FAIL`) by the z-rank of the corresponding function.

4. The programmer inspects the errors starting from the top of the list and continuing down until the false positive rate is too high.

If the success and failure messages are emitted in a standard form, we provide a generic script that will do steps 2 and 3 for a wide range of statistical analyses.

To make the process more concrete, consider the code in Figure 9. There are four calls to functions that return a pointer: two for `malloc` and two for `foo`. The returned pointer of `malloc` is checked once before use (a success) and used once without checking (a failure). Both calls to `foo` use the return pointer without checks (two failures). Thus, the z-rank for the single `malloc` error message will be:

$$1/2 - .8/\sqrt{.8 * (1 - .8)/2} = -1.06$$

And the z value for the two error messages for `foo` will be

$$0/2 - .8/\sqrt{.8 * (1 - .8)/2} = -2.83$$

Thus, the error for `malloc` will be ranked above the errors for `foo`. (Note that in general the counts and skew are much higher.)

4.2 Inferring Deallocation Routines

This section describes how to statistically infer functions that free their arguments. We first show a deterministic checker that warns when a pointer passed to the deallocation function `kfree` is later used. We then show a statistical checker that infers such functions.

Figure 10 shows the deterministic free checker. It works as follows:

1. When a variable is passed to `kfree`, it is put in the `freed` state and a new SM is created to track that variable (line 23).

2. The only legal operations on freed pointers are (1) comparisons to other pointers, and (2) printing their

```
1 : sm_header {
2 :   // does fn contain "print" or "debug"?
3 :   static int is_debug_call(mc_tree fn) {
4 :      char *n;
5 :      if(!(n = mc_identifier(fn)))
6 :         return 0;
7 :      return strstr(n, "print") != 0
8 :          || strstr(n, "debug") != 0;
9 :   }
10: }
11: sm free_checker local subsume {
12:  state killvars decl any_pointer v;
13:  decl any_pointer x;
14:  decl any_fn_call call;
15:
16:  pat safe_uses =
17:       // don't warn about comparisions.
18:       { (v == x) } || { (v != x) }
19:       // or calls to debugging functions that
20:       // have v as any argument.
21:       || ({ call(-1, v) } && ${ is_debug_call(call) })
22:       ;
23:  start: { kfree(v) } ==> v.freed
24:    ;
25:  v.freed:
26:     safe_uses ==> { /* do nothing */ }
27:   // warn about all other uses.
28:   | { v } ==> v.stop,
29:        { v_err("FREE", v, "Using freed '$name'!"); }
30:         ;
31: }
```

Figure 10: Deterministic checker to detect when a freed variable is used.

value in debugging code. The patterns on lines 18 and 21 identify these two cases. Note that the callout to `is_debug_call` is used to refine a pattern that matches any function call with the tracked object as an argument to one that only matches debugging routines (identified by the strings "print" and "debug"). The transition on line 26 identifies safe uses of a freed pointer and does the identity transition in the SM.

3. All other uses of the freed pointer are flagged as errors by the transition on lines 28–30.

The need for the transition identifying safe pointer uses is rather subtle. There are two ways to write this extension. One way identifies all possible illegal uses of freed pointers explicitly. Since the only truly illegal use of a freed pointer is a dereference, this checker seems easy to write at first cut. Without a perfect alias analysis, though, this checker will miss many errors when pointers are assigned into arrays or complex data structures.

Instead of identifying illegal uses of freed pointers, the extension in Figure 10 flags all uses of a freed pointer as illegal *except* comparisons and debugging printouts. The subsumption feature in *metal* makes this extension easy to write. The transition on lines 28–30 flags an error on all uses of the freed pointer, but the transition on line 26 will subsume that transition if the use is actually legal.

While this checker works well in practice, systems often have many different deallocation functions, ranging from general-purpose routines to wrappers around these routines to a variety of ad hoc routines that manage their own free lists. We want to automatically detect such functions.

In general, a pointer passed to a valid free function will not be used afterwards. We can thus infer if a function frees its argument using the same template as we used in the statistical null checker. We show the code for the statistical free checker in Figure 11. The extension works as follows:

1. Blindly assume that *every* function frees *all* of its pointer arguments.

2. Count the total number of function-argument pairs (n in the formula for the z-statistic). We can think of this number as the size of the population from which we are sampling.

3. Each time a function argument is used in a way that would be unsafe for a freed pointer, emit a failure message that indicates which population the failure belongs to. The number of such error messages (err) equals $n - k$ in the formula for the z-statistic.

4. Rank the errors for each pair using the z value computed based on n and $k = n - err$. This ranking puts the errors from the most likely pairs to the top of the list.

For simplicity, the pattern on line 22 in the statistical free checker only considers functions that take a single pointer argument as input. The pattern on line 29 also limits the number of populations we consider by only considering functions with a suggestive name. The callout to `is_free` refines the pattern on line 22 to identify these calls.

The population size for a given function (n in the z-formula) is the number of callsites. The transition on lines 22–26 records the name of the function in the data field, and emits a message at each callsite. The tag `POP=name` in the error message tells the post-processing script that the message is associated with function "name." The script counts the number of `POP` and `FAIL` messages for each function and ranks its error messages (i.e., those with `FAIL` in them) based on the z value computed from these counts. These failures are the most likely dereferences of deallocated pointers.

The similarity between the deterministic and statistical null and free checkers is not coincidental. Almost all deterministic checkers have a useful statistical analogue that can find more code actions to check.

Statistical analysis can be used to compute many other properties: which locks protect which variables, what are the bounds of an array, etc. It can also be used to learn general regular expressions from source code traces.

Surprisingly, statistical analysis is useful even for known checks. Using the methods described, the errors reported by a checker can be ranked from most probable to least probable. We have noticed that the most probable error reports are those that come from checkers that (1) flagged few errors in total and (2) had many successful checks. Clearly, this is analogous to the properties of likely rules described above. By using a similar form of z-ranking, we can rank error messages based on their likelihood of being actual errors. This ranking works well in practice. It has transformed checkers with unusably high false positive rates into effective bug finders.

5. RELATED WORK

There are several other projects that are similar in spirit to *metal* and *xgcc*, although, in most cases, the underlying

```
sm_header {
  // does fn contain suggestive name?
  static int is_free(mc_tree fn) {
      char *n;
      if(!(n = mc_identifier(fn)))
         return 0;
      return strstr(n, "free") != 0
          || strstr(n, "Free") != 0
   || strstr(n, "Dealloc") != 0
   || strstr(n, "dealloc") != 0;
   }
}
sm stat_free_checker local subsume {
 state killvars decl any_pointer v;
 decl any_pointer x;
 decl any_fn_call call;

 pat safe_uses =
      { (v == x) } || { (v != x) }
      || ({ call(-1, v) } && ${ is_debug_call(call) })
      ;
 start: { call(v) } && ${ is_free(call) } ==> v.freed,
   {  mc_v_set_data(v, mc_identifier(call));
      v_note("STAT_FREE", v,
              "freeing $name [POP=$data]"); }
   ;
 v.freed:
    safe_uses ==> { /* do nothing */ }
  | { v } ==> v.stop,
       { v_err("STAT_FREE", v,
            "Using freed '$name'! [FAIL=$data]"); }
  ;
}
```

Figure 11: Statistical checker to infer free functions.

goals and technologies are different. The PREfix project, described in [3], has the most similar goals to our own although the underlying technology is very different. Several projects aim to *verify* temporal safety properties [4, 2, 11]. These projects hope to find all violations of a given property, whereas we make no such guarantees. Several other projects check similar properties using language annotations [7, 8]. We view these approaches as complementary to our own. Annotations can be useful to analysis, but programmers do not reliably annotate their code. None of these projects offer the same flexibility in the rule description as *metal* does, but most of them do provide stronger guarantees for the results of the analysis. A more complete discussion of related work can be found in [10].

6. CONCLUSION

This paper provides an overview on how to write system-specific, static checkers in the *metal* language. *Metal* allows extension writers to concisely express a broad range of checking properties. Its key features are: (1) patterns to match interesting code constructs, (2) callouts, which can refine pattern results, (3) states, which provide a sugared way to express restrictions and (4) actions, which augment the state machine with general-purpose code. States can be either global (essentially bound to the current code path) or bound to a given expression. Extensions can be run using either a context-sensitive interprocedural analysis or intraprocedurally. Both inter- and intraprocedural analysis are flow sensitive. Extensions can be composed and can communicate using annotations.

We described how to make checkers more powerful by cross-checking program beliefs and by using statistical analysis to automatically find what to check. Both techniques allow checkers to detect errors without having to know truth.

7. ACKNOWLEDGEMENTS

We thank Junfeng Yang and Ken Ashcraft for their helpful comments. This work was supported by NFS award 0086160 and by DARPA contract MDA904-98-C-A933.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[2] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.

[3] 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.

[4] Manuvir Das, Sorin Lerner, and Mark Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[5] 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.

[6] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[7] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, and J. Saxe. Extended static check for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 246–257, 2002.

[8] J.S. Foster, T. Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

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

[10] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[11] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, January 2002.

[12] Thomas Reps, Susan Horowitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22th Annual Symposium on Principles of Programming Languages*, pages 49–61, 1995.