A Quick and Dirty Guide to Writing Practical Compiler Extensions with Metal

Andy Chou, Dawson Engler, Benjamin Chelf, and Seth Hallem
{acc, engler, bchelf, shallem}@cs.stanford.edu

1 Introduction

This introduction describes how to write practical extensions with Metal. It is not meant to be a complete specification of the language but rather a practical guide on how to make extensions that work on real code.

See our research papers for more information on the MC project [3, 1] and an introduction to Metal [2].

2 Hello world from start to finish

Extensions are written in *metal*, which provides support for matching code (using patterns), transitioning to states (discussed below), and performing actions (written in C). Following is *metal*'s version of "hello world": it searches for calls to `printf` with the format string ``` ``hello world'' ``` and prints out ``` ``hello world!'' ``` every time it hits one:

```
sm hello {
   start: { printf("hello world\n"); } ==>
   {
        printf("hello world!\n");
   };
}
```

The first line of the extension is boilerplate: `sm` states that this is an extension called `hello`. The extension is a degenerate state machine that has one state, `start`, and no transitions out of it. Most extensions have multiple states as well (necessarily) transitions. The main interesting things are (1) the literal pattern it uses:

```
{ printf("hello world\n"); }
```

to find all `printf` calls and (2) the action, which runs what it matched:

```
{
     printf("hello world!\n");
};
```

Both patterns and actions can be (nearly) arbitrary C code. Patterns will match code, whereas actions are run at each match.

If we run the extension on a variant of the C hello world program:

```
int main(void) {
     printf("hello world\n");
     printf("foo");
     return 0;
}
```

it produces the output:

```
hello world!
```

Note that it does not match the second `printf` call because the string, ``foo'' does not match. Neither does it match `return 0;` since this is not a call.

A useful extension feature is pattern variables, which allow you to name patterns. Here is a modified version of our `hello` extension:

```
sm hello2 {
   /* pattern variable */
   pat hello = { printf("hello world\n"); };

   start: hello ==>
   {
        printf("hello world!\n");
   };
}
```

Note that we do not put braces around the pattern variable name.

Pattern variables can also capture several patterns, which are joined by the "|" operator. E.g.,:

```
pat hello = { printf("hello world\n"); }
          | { puts("hello"); }
          ;
```

This variable will match if we hit either the `printf` or `puts` call.

2.1 Compiling and running

The extension can be compiled using the `mcc` compiler:

```
mcc hello.m
```

The output of the *metal* compiler is in three files [1]:

1. `hello.h`: a header file for `hello.c`.

2. `hello.c`: source code for the compiler extension. This file compiles into a shared library object, `hello.so`, which will be dynamically linked into *xgcc* when it compiles the program to be checked. This file contains code to perform the transitions of the state machine and also the code for the actions.

3. `hello.p`: pattern file containing all patterns to be matched. This file is included when the extension is run, and it contains information to allow *xgcc* to find the `hello.so` library. The primary purpose of this file is to provide fragments of code that represent the patterns in the SM. The patterns are wrapped in specially named functions, and when *xgcc* compiles these functions, the patterns are parsed into gcc trees. Since these trees have basically the same form as code that matches the pattern, pattern matching can be done with a straightforward tree comparison.

`hello.h` and `hello.c` are compiled into a shared library object file `hello.so`:

```
gcc -DIN_GCC -fPIC -shared -g -I/home/acc/mc/mc/include -I/home/acc/mc/mc/include/extensions -c hello.c
gcc -DIN_GCC -fPIC -shared -g hello.o -o hello.so
```

[1] You can usually ignore these files, other than telling the compiler where to get them.

The `hello.p` file is included when *xgcc* is run over the source of the program to be checked. For example, if we have code to check in `test-hello.c` the compile line looks like:

```
xgcc -c -include hello.p -I/home/acc/mc/mc/include -I/home/acc/mc/mc/include/extensions -fmc test-hello
```

Note that the `Makefile` distributed with the extensions will usually take care of all of this automatically, as long as you tell it the name of your extension (in this case `hello.m`).

Execution model. Conceptually, by default extensions follow all paths in the code. This means that if we introduce an if statement:

```
int hello2(void) {
     int x;
     if(x)
             x++:
     printf("hello world\n");
     return 0;
}
```

we'd expect that the extension would print out `hello world` twice since there are two paths through this code (one path that assumes `x` is true, the other where `x` is false). However, if you run the extension, it will only print out one message — this is because for speed the implementation caches what states it has seen at every point and if it gets to the same statement in the same state it stops traversing that path. Without this optimization, many extensions would be unusable, since the number of paths increases roughly exponentially with the size of code. In the case above, because our extension never transitions out of the `start` state, it will stop running the second time it hits the `printf` call.

As we'll discuss later, with more interesting extensions this pruning may not happen so you can get identical messages depending on how you got to a statement.

2.2 Checking for floating point

Checkers do not need to be elaborate to be useful. We discuss one such example below. Kernel code has environment restrictions user-level code does not. One example is that OS code cannot typically use floating point — the system call may not have saved floating point state, or the co-processor may not be enabled. The following extension searches for floating point uses and emits an error:

```
sm float_check {
   decl { double } d;
   decl { float } f;

   start: { d } | { f } ==>
       { mc_note(mc_stmt, "ERROR:FLOAT: can't use fp in kernel"); }
       ;
}
```

The extension has a single state `start` and no transitions. It uses two patterns that are simply the variables `d` and `f`, which will will match any variable or expression of type `double` or `float` respectively.

When run on code such as:

```
/* linux 2.4.4:drivers/video/sis/sis_main.c */
/* 760 */  double drate = 0, hrate = 0;
           ...
/* 778 */  hrate = drate / htotal;
/* 779 */  ivideo.refresh_rate =
                (unsigned int) (hrate / vtotal * 2 + 0.5);
```

We get the error messages:

```
drivers/video/sis/sis_main.c:760:do_set_var: ERROR:FLOAT: can't use fp in kernel
drivers/video/sis/sis_main.c:778:do_set_var: ERROR:FLOAT: can't use fp in kernel
drivers/video/sis/sis_main.c:779:do_set_var: ERROR:FLOAT: can't use fp in kernel
```

3 Pattern Matching

3.1 Hole variables

When writing extensions it's often necessary to match patterns on an entire set of expressions. For example, suppose you wanted to match calls to `printf` that have an arbitrary argument string, not just "Hello, world." Hole variables, declared with `decl`, allow you to do this:

```
sm hello {
    decl { char * } str;
    pat hello = { printf(str); };

 start: hello ==> {
     printf("Hello, world!\n");
 };
}
```

Here, `str` is a hole variable that matches any expression of type `char *`. Then, patterns can use this hole variable in their definition whenever they need to match an arbitrary `char *` expression. For example, on the following input:

```
int main(void) {
    char *s = "bar";
    printf("Hello, world\n");
    printf("foo");
    printf(&s[1]);
    printf("%d", 0);
    return 0;
}
```

The extension outputs:

```
Hello, world!
Hello, world!
Hello, world!
```

The extension matches the first three `printf`'s, each of which take a single `char *` argument; the third pattern does not match because it takes two arguments.

The keyword `literal` can be prepended to any standard C type when specifying a hole variable. The `literal` must appear outside of the brackets as in the following example:

```
decl literal {int} x;
```

The meaning of the keyword `literal` is that the hole variable should only match an expression of the enclosed type if the expression is a constant of that type. Thus, the hole variable `x` as specified in our example will only match integer constants. If the C type specified within the brackets is not a basic type, then adding the keyword `literal` makes no sense and will not be interpreted correctly.

Special types	Meaning
any_expr	any legal expression
any_scalar	any legal scalar
any_pointer	any legal pointer
any_stmt	any legal statement
any_arguments	any set of function arguments
any_fn_call	any function call
Special patterns	**Meaning**
end_of_path	match the end of a path
$epsilon$	matches any token
Special states	**Meaning**
$stop$	stop the current path or state variable
killed	a state variable in this state will not be reinstated, even after a kill

Table 1: Representative *metal* keywords and their meaning.

3.2 Special forms

Holes are based on types, so they are limited by what can be expressed in the underlying type system of C. There are many cases where this is too restrictive; for example, many extensions need to match on any function call, but with only type-based hole variables this is difficult to do. To address this problem, *metal* provides a set of special forms that expand or restrict what patterns will match. Table 1 shows the special forms.

The special types are used to declare hole variables, which can then be used in patterns. For example, to match any function call with any arguments two special types are used:

```
sm hello {
    decl any_fn_call f;
    decl any_arguments args;

    pat hello = { f(args); };

 start: hello ==> {
     printf("Hello, world!\n");
 };
}
```

This extension will match all four calls to `printf` in the hello world example, even the one with two arguments. This usage of `any_fn_call` is syntactic sugar. In general, to specify a function that matches exactly N arguments, pass N as the first parameter to the call and specify the following N arguments as holes:

```
sm hello {
    decl any_fn_call f;
    decl { char * } str;
    decl { int } num;

    pat hello = { f(2, str, num); };

 start: hello ==> {
     printf("Hello, world!\n");
 };
```

```
   ``a+b''            ``f(a+b)''                  ``*(a+b) = 0''
  ---------      --------------------------      ----------------

     +              function_call                   modify_expr
    / \                /   \                           /    \
   a   b              f   list_node                 deref    0
                            /   \                     |
                           +    nil                   +
                          / \                        / \
                         a   b                      a   b
```

Figure 1: A few sample abstract syntax trees (ASTs).

```
}
```

Using this pattern, the SM will only match the final `printf` in the example, which has one `char *` and one `int` argument.

In addition to special types, there are two special patterns: `$epsilon$`, which will match every tree, and `$end_of_path$`, which matches when the SM hits an exit point of the function (i.e. when there is no more code along a path). `$end_of_path$` is especially useful for SMs that check for something that should (or shouldn't) happen at every exit point of a function, for example freeing memory that was allocated or releasing locks that were acquired.

Callouts.

3.3 Exactly what will patterns match?

It's not always obvious what a pattern matches. To get a feel for what patterns will match, it is helpful to understand at a high level how pattern matching happens.

As described in Section 1, each pattern is parsed by `gcc` into a tree. These trees represent statements, expressions, types, and all other language constructs. A few examples are shown in Figure 3.3. A very simple pattern matcher could be implemented recursively by the following algorithm (pseudo-code):

```
int match(mc_tree pattern, mc_tree code) {
        if(tree_type(pattern) != tree_type(code))
                return 0;
        if(num_children(pattern) != num_children(code)
                return 0;
        for(i = 0; i < num_children(pattern); i++)
                if(!match(child(pattern, i), child(code, i)))
                        return 0;
        return 1;
}
```

Intuitively, two trees match if they are the same type of node and their children all match. To extend this algorithm to handle hole variables, the pattern matcher simply needs to notice when a pattern is a declared hole. When this happens, the hole will match anything of the same type in the `code` tree, regardless of the exact tree type or children. For instance, consider the pattern shown in Figure 3.3. Suppose this pattern is in an SM running over code that includes a call to "f(a+b)." The matcher will match on the `function_call_node` and then recursively match the children. The function `f` matches, and the `list_node` matches. Then, the hole variable, which has type `int`, is matched against the expression "a+b." If this expression also has type `int`, the pattern will match,

```
decl { int } x;
pat p = { f(x) };

        ‘‘p’’
 --------------------
    function_call
        /   \
       f   list_node
             /   \
       HOLE:int   nil
```

Figure 2: A pattern may have “holes” in its AST. These holes have types and will match any expression with that type.

and the entire match succeeds, thus matching “f(x)” with “f(a+b).” This mechanism can be easily extended to add special forms as well.

The basic matching algorithm may seem simple enough, but there are many details to watch out for; keep the following things in mind when writing patterns:

1. Statements vs. expressions. Recall that expressions have values and possibly side effects, but statements only have side effects. If you put a semicolon into your pattern, it becomes a statement and will not match any subexpressions. For example, “f(x)” will match the function call subexpression in “f(0) + 1” but “f(x);” will not.

2. **end_of_path** matches at least once for each exit point of the function. More precisely, it matches once for each state of the SM at each exit point. It also matches when a per-variable state is killed (see Section 5).

3. Synonyms. Sometimes it’s the meaning of a pattern that’s important rather than an exact syntax match. For example, variable declarations with initializers like “`int a = 0;`” are essentially assignment expressions of the form “`a = 0`.” Table 2 shows a list of all synonyms built into the matcher. The first two synonyms match almost-equivalent expressions: assignments are like initializers, and structure field references through a pointer are like dereferences of the pointer itself (this is good enough for checking NULL pointers but is not strictly correct in general). The third synonym allows for composition: an assignment and a test for equality can be specified as separate patterns and they will both match. The assignment will match first and then the test. This synonym makes the NULL checker cleaner.

4. Including the right header files. Recall that patterns are parsed into trees at the same time the code is compiled, so these trees should be the same as the ones that are built from the code to be checked. This relies on your specifying the correct include files in your SM:

   ```
   { #include "my-includes.h" }
   sm { ... stuff ... }
   ```

   If the proper headers are not included then you will likely get a compile error, but this is not guaranteed. One insidious problem is macros: these can dramatically change the form of a pattern if they are not included correctly.

Pattern	Synonym
a = b	int a = b
*p	p->any_field
v = a with v == b	((v = a) == b)

Table 2: Pattern synonyms. The first two synonyms match almost-equivalent expressions: assignments are like initializers, and structure field references through a pointer are like dereferences of the pointer itself. The third synonym means that an assignment and a test for equality can be specified as separate patterns and they will both match: first the assignment and then the test.

3.4 Precedence when matching multiple patterns

When an SM has multiple patterns that can match in a given state, which one will trigger a match? One could easily imagine several schemes: longest match (greedy), match in order the patterns are specified in the SM (user-specified precedence), or some other ordering. Our system uses a combination of these two schemes to determine precedence. The matcher will choose the longest pattern that matches in a given state, unless the shorter patterns that match appear before the longer pattern in the SM definition. If this is the case, both the shorter patterns and the longer patterns will match.

The size of a match is the number of tree nodes in that match. Thus, for example, a match of the pattern `foo()` is smaller than a match of the pattern `(v = foo()) == 0)`. If the pattern `foo()` appears after the pattern `(v = foo()) == 0` in the SM definition, then whenever the latter pattern matches, the former pattern will not. If the ordering of these two patterns was reversed in the SM definition, then any match of the larger pattern would also be a match of the smaller pattern, and both matches would be signalled to the SM.

4 States and Transitions: Interrupt Checker

More typically, extensions are loosely written as state machines (SMs), which consist of a set of states and some transitions between them. Graphically, a state machine looks like a directed graph that may contain cycles, self-loops, and multiple edges between the same nodes. In *metal*, transitions are triggered when the state machine matches a piece of code; we can represent this graphically by labeling each edge with the associated pattern.

The extension below illustrates how to use these features. It checks that procedures in the Linux kernel that disable interrupts (using a call to `cli()`) subsequently re-enable them on all paths (using a call to `sti()`). It gives an error on duplicate calls to these functions, or if the procedure appears to exit with interrupts disabled. (A slightly more complex version of this checker found errors in several Linux device drivers [3].)

```
{ #include "linux-includes.h" }
sm check_interrupts {
  // Variables
  // used in patterns
  decl { unsigned } flags;

  // Patterns
  // to specify enable/disable functions.
  pat enable = { sti(); }
```

```
            | { restore_flags(flags); } ;
  pat disable = { cli(); };

  // States
  // The first state is the initial state.
  is_enabled: disable ==> is_disabled
     | enable ==> { err("double enable"); }
     ;
  is_disabled: enable ==> is_enabled
     | disable ==> { err("double disable"); }
     // Special pattern that matches when the SM
     // hits the end of any path in this state.
     | $end_of_path$ ==>
        { err("exiting w/intr disabled!"); }
     ;
}
```

The given SM uses three states, `enabled`, `disabled`, and `all`, to track the current interrupt state for each path. By default, SMs start in the state mentioned in the first transition definition (here `enable`). As it begins in this state, the system checks in lexical order to see if any of its patterns match in order, if so, it "consumes" the input that matched, executes the optionally provided action and sets the state to the optionally specified new state (the token after the `==>` operator).

For the above SM, `enable` searches for all interrupt disabling using the literal pattern "`{ cli(); }`," which, if it succeeds, causes the SM instance on this path to transition to the `disabled` state. Similarly, calls to `sti()` are caught and cause an error (multiple enablings are questionable at best). By omitting the state token for this transition, the SM will remain in the current state.

The `disable` state is roughly the inverse of `enable`, with the refinement that it uses the special pattern `$end_of_path$` to detect when the end of current path is reached in this state.

The final state `all` is predefined by *metal* to specify transitions that can happen in any state. The alternative would be writing the transition in every rule. These default transitions are only run after all lexically preceding rules have failed to match. The SM uses this state along with a transition to the special `$stop$` state (which stops the traversal of a given path) to eliminate false positives by stopping a path traversal if it hits a function that will cause the kernel to self-destruct, since such paths are likely to not obey system restrictions.

All states are considered to be accepting by default; rejecting states should define a transition with an empty pattern and an action that calls an error function.

This check finds hundreds of errors, such as:

```
/* From linux 2.4.1: drivers/isdn/isdn_ppp.c */
int
isdn_ppp_bind(isdn_net_local * lp)
{
        save_flags(flags);
        cli();
        ...
        if (i >= ISDN_MAX_CHANNELS) {
                restore_flags(flags);
                printk(KERN_WARNING "isdn_ppp_bind: Can't find a (free)"
                                     "connection to the ipppd daemon.\n");
                return -1;
        }
```

```
        ...
        if (unit < 0) {
                printk(KERN_ERR "isdn_ppp_bind: illegal interface name "
                                "%s.\n", lp->name);
                return -1;
        }
        ...
        restore_flags(flags);
}
```

5 Per-variable states

This section presents extensions that use per-variable state. In our system, per-variable state refers to an implicit state that is attached to each variable of interest in a function. Variables of interest are identified through a pattern match. If a *metal* variable is declared with the special keyword `state`, it signals that if a pattern containing this variable is matched under the proper conditions, the system should attach a state to the AST element corresponding to that variable in the match.

The following extension finds when freed variables are used by (1) binding each freed variable to a `v.freed` state, and (2) in this state using standard patterns to catch dereferences.

```
sm simple_free_checker {
   state decl any_pointer v;

   start: { free(v)  } ==> v.freed
    ;
   v.freed: { *(any *)v } ==>
          { mc_note(mc_stmt, "Use-after-free of '%s'!", mc_identifier(v)); };
}
```

We declare `v` as a `state decl` to indicate that the system should attach a state to instances of `v` that are interesting. `v` is of type `any_pointer` since we want to catch all freed pointer errors. We create an instance of `v` with an attached state by matching a pattern in a global state that contains the variable `v` and transitioning to a local state prefixed with the `v.` notation. In the SM above, the only global state is `start`, and the only global pattern is `free (v)`. Since the first transition goes from the `start` state to the `v.freed` state, a new variable/state pair is created at every free call. The variable/state pair created attaches a state to the AST element representing the argument to `free` in the source code. This global to local transition is specified using the same notation as we used to specify purely global state transitions with the addition of the `v.` preceding the local state name:

```
   start: { free(v)  } ==> v.freed
```

Once the variable/state pair is created, the new pair starts in the local state `freed`. Any pattern in the state `v.freed` is now active for this instance of `v`. Here we just have a single pattern: `*(any *)v` that will match all dereferences of an instance of `v`.

A few somewhat subtle things to note:

1. The dereferencing pattern `*(any *)v` has to be used rather than the more natural `*v` because of the details of how patterns are implemented. This is unfortunate, but not a big deal in practice — the compiler will warn you if you forget.

2. There can be multiple instances of `v` in action concurrently. One instance is created at each call to `free`. Each of these instances are independent. For example, the code:

```
free(p); /* p = freed.
            instance of v with AST for p is created. starts
            in freed state. */
free(q); /* q = freed.
            instance of v with AST for q is created. starts in
            freed state. */
*p;     /* use after free */
*q;     /* use after free */
```

Will give two errors. The first error refers to the use after free of `p`, the second to the use after free of `q`. The AST elements correponding to `p` and `q` both have state, and those states are stored and transitioned independently based on usages of `p` and `q` respectively.

3. Per-variable patterns override global patterns: if an instance, `p`, of the `state decl v` already exists, a duplicate will not be created if `free(p)` appears in the code a second time. In addition, the current instance of `p` will not be transitioned to the `freed` state because that would restart the analysis of `p`. This prevents existing variable/state pairs from undergoing state transitions other than those that are specified by the actions within the currently active local state (in this case, the pattern `*(any *)v` in the `v.freed` state).

4. The system will "kill" a variable with state whenever the corresponding AST element is overwritten in the source code. E.g., neither of these two will give errors:

```
free(p);/* p = freed */
p = q;  /* kill of p */
*p;     /* no use after free error */

/* increment of i kills the expression a[i] */
for(i = 0; i < n; i++)
        free(a[i]);     /* no use after free error */
```

When a variable with attached state is assigned a new value in the source, we no longer know anything about that variable's state. We handle this by killing the variable. Once a variable is killed, the global states are reactivated for that particular variable. Thus, if `p` is passed as an argument to `free` again, we now know something about `p`'s state, and we give it the local state `freed`. This killing happens transparently within the system. To explicitly kill a variable within an SM, the SM can transition the variable to the state `v.stop`.

5. End-of-path will match both when a path ends (either because the path hits the end of the function of the path is cut short by caching) and when an AST node with per-variable state is killed.

6. If the state variable (in this case `v`) appears in a pattern that cannot create a new instance of `v`, the pattern matches only if the variable corresponding to `v` in the code matches an already existing instance of `v`. For example, the following dereference of `q` will not match because the only instance of the state variable `v` is the variable `p`:

```
free(p);
*q;             /* does not match */
```

Local states (like `v.freed`) can also have patterns active in those states that do not contain the variable `v` — thus, the pattern can match whether or not an instance of `v` appears in the source code. For example, if we modified the above SM with the following additions:

```
decl any_fn_call fcall;
decl any_args args;

pat fn_call = {fcall (args)};

/* ... */

v.freed: fn_call ==> { /* ... */ };
```

the pattern `fn_call` would match on any function calls as long as there were some variables in the `v.freed` state. If there are no variables in the `v.freed` state, the `fn_call` pattern would not be active at all. One additional feature is that the action for patterns of this type do have access to the instance of `v` that activated the pattern. If we ran our modified extension over the code

```
free(p);
foo (); /* matches here because p is in the freed state */
```

the action represented by the `{ /* ... */}` in the modified extension could use the *metal* variable `v`. In this case, `v` would actually refer to the AST node for the variable `p`.

5.0.1 A more refined `free` checker

This section does an in-depth examination of how to make the previous naive checker closer to being industrial strength.

The previous checker misses a lot of errors, since it only warns about dereferences. For example, it will miss the following double free:

```
free(p);
free(p);   /* really really bad, but not caught. */
```

The biggest single improvement is that we want to warn about all uses of freed variables, rather than just dereferences. Unfortunately, if we change the SM in the obvious way by adding the pattern:

```
v.freed: { v } ==>
    { mc_note(mc_stmt, "Use-after-free of '%s'!", mc_identifier(v)); };
```

We will get an enormous number of false positives, since there are some uses of freed variables that are legal. You would discover this weakness in your extension by running the extension over Linux and getting irritated by the number of false positives. To fix it, you would notice certain idiomatic usages of freed variables that are acceptable and add special patterns to allow these uses. Patterns that come up a lot in Linux are comparisions and debugging prints:

```
free(p);

/* safe: just prints the pointer value of p */
printf("pointer %p", p);
...
/* safe: just compares p */
if(p != x)
        free(x);

/* will cause a false match */
p = q;
```

To suppress these we add the following patterns:

```
decl any_pointer x;

/* ... */

v.freed:
   { v == x } | { v != x }
| { v = x }
| ${ is_debug_call(mc_stmt }
       ==> { /* reduce false pos */ }
 | { *(any *)v } | { v } ==> v.stop,
       { mc_note(mc_stmt, "Use-after-free of '%s'!", mc_identifier(v)); };
```

Here, the first two patterns will suppress usages of `v` in comparisons to any other pointer (represented with the wildcard `x`). The third pattern will match when `v` is assigned another value — in that case, the current matcher would otherwise give us a false warning because `v` is used during the assignment, even though the end result is that it is killed. The final pattern suppresses warnings when a freed variable is passed as an argument to printing routines. It is implemented as:

```
/* return 1 if it's a potential debugging call */
static int is_debug_call(mc_tree t) {
     char *n;

      if(!mc_is_call(t) || !(n = mc_identifier(t)))
           return 0;
      return strstr(n, "print") != 0;
}
```

As a final refinement, in practice we have found that if code uses a freed variable once, it will likely use it more than once. To stop a cascade of error messages, we transition `v` to the special *xgcc* pre-defined state, `v.stop`, which stops all subsequent transitions. Note that if the variable is freed again, any illegal usages of the variable after that second `free` will be flagged because the variable is "restarted" by the second call to `free`.

Here is the final version of the checker:

```
sm final_free_checker {
   state decl any_pointer v;
   decl any_pointer x;

   start: { free(v)  } ==> v.freed
    ;
   v.freed: { v == x } | { v != x }
   | { v = x }
   | ${ is_debug_call(mc_stmt) }
          ==> { /* reduce false pos */ }
    | { *(any *)v } | { v } ==> v.stop,
          { mc_note(mc_stmt, "Use-after-free of '%s'!", mc_identifier(v)); }
  ;
}
```

While the final checker is more heavyweight than the initial free checker it is also significantly more effective. This checker was able to find the following security hole in Linux 2.4.1:

```
/* 2.4.1:drivers/block/cciss.c:cciss_ioctl:
    Use-after-free of 'buff'! set by 'kfree':664 */
    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);
    if (buff != NULL)
       kfree(buff);
```

Here, if the `copy_to_user` fails (which is depends on the user's behavior), both `c` and `buff` will be freed twice. Thus, malevolent users can cause breathtakingly bad corruption of kernel memory at will.

The free checker as written is a very conservative, purely local checker. As a result, it will reject many legal programs. For example, it will complain about any freed pointers passed as arguments, whether or not they are subsequently used. However, in real life, such code is not written (or not blessed if it is) — any program that passes around freed pointers is a victim of bad style, whether or not it actually uses them.

This observation is a common feature that many of our checkers exploit aggressively: programs that are easy for people to understand tend to be easy for for our extensions to analyze as well.

6 *metal* callouts

Many of the questions you may want to answer cannot be expressed directly in C code. *metal* provides a set of callouts that can be used to interogate the AST tree that it traverses during matching. Syntactically, a callout looks just like a pattern except that it is prefixed with `$`:

```
${ any_boolean_expression() } ==> { /* ... */ }
```

It's contained C code is run instead of being matched, and should return 1 on match and 0 on failure. For example, a callout that will match all nodes would just contain the constant 1:

```
/* match all nodes */
${ 1 } ==> { mc_note(mc_stmt, "node!"); }
```

Callouts allow you to see what a tree node is (a for loop, a variable, a procedure call, etc.) as well as it's type (integer, unsigned, etc.) or other attributes (variable names). These can be used both during matching and during the actions.

The following extension illustrates using callouts both in patterns as well as in actions. It warns when stack variables are "too large". OSes usually have fixed sized stacks (e.g., the linux stack is about 8K). A large local variable can easily exceed this amount of space. To check this, the following extension grabs all variable declarations and, if they are allocated on the stack, warns if they are larger than 1024 bytes:

```
sm stack_check flow_insensitive  {
   /* find all ``decl statements'' that are not static variabls
      or constants (neither are allocated on the stack) */
   start: ${ mc_is_decl_stmt(mc_stmt)
     && !mc_is_static(mc_decl_stmt_decl(mc_stmt))
             && !mc_is_readonly(mc_decl_stmt_decl(mc_stmt)) } ==>
        {
                int n = mc_type_size(mc_decl_type((mc_stmt)));
                if(n >= 1024)
                        mc_note(mc_stmt, "ERROR: variable is %d bytes!", n);
```

```
        }
  };
}
```

The callout pattern:

1. grabs local declarations using `mc_is_decl_stmt`;
2. tests if they are constants using `mc_is_readonly`;
3. tests if they have static storage using `mc_is_readonly`;

The action then extracts the declaration statement type using `mc_decl_type` and gets it size by calling `mc_type_size`.

It will warn about innocuous looking but dangerous situations such as:

```
2.4.4/drivers/sound/emu10k1/audio.c:906:emu10k1_audio_ioctl:
 ERROR: var 'buf' is 4016 bytes!
        case SNDCTL_COPR_LOAD:
                {
                        copr_buffer buf;
```

There were about 32 similar cases in the linux 2.4.4 kernel.

7 Example: A then B on error paths

A common bug in kernel code is to perform side-effects (allocating memory, acquiring locks, etc), hit an error condition, and then forget to reverse these side-effects before returning an error code. These errors are common, in part, because are many operations to reverse, it's difficult to test error paths, and such errors tend to be silent (losing storage) rather than causing hard segmentation faults or deadlocks.

This section describes a checker to catch such errors. One of its more interesting aspects is how it suppresses false positives — with simple steps we can make an unusable checker into one that is very effective.

The rule we want to check is that "A must be followed by B on error paths." Idiomatically, we can find an error path by looking for returns of either negative integers or "`NULL`" pointers. The "A then B" check is similer to the null checker already discussed. Figure **??** gives a first simple version that tracks memory allocated with `malloc` is deallocated using `free`. Its main structure:

1. On every `malloc` call of any size (`v = malloc(sz)`) we put the assigned-to variable `v` in an unknown state.
2. If this variable ever hits a free (`free(v)`) we stop following it.
3. If `v` is compared to null or not null we stop following it on the true and false branches respectively.
4. If we hit a return that is identifiable as an error path (negative integer, or a return of `NULL`) we give an error.

When run on the test case:

```
{ #include <malloc.h> }

sm_header {
           /* Assume < 0 and null returns are errors */
           static int is_err_ret(mc_tree retv) {
                if(mc_is_integer_cst(retv) && mc_int_val(retv) < 0)
                 return 1;
                if(mc_is_pointer_cst(retv) && !mc_pointer_cst(retv))
                        return 1;
                return 0;
           }
}

sm inv_checker {
   decl any_scalar retv;
   state decl any_pointer v;
   decl any_scalar sz;

   /* track every allocation */
   start: { v = malloc(sz) } ==> v.unknown;

   v.unknown, v.not_null:
      /* when we hit a free, ignore the variable */
      { free(v); } ==> v.stop
      /* kill the variable on null paths */
    | { (v == 0) } ==> true = v.stop, false = v.not_null
    | { (v != 0) } ==> true = v.not_null, false = v.stop
    ;

     /* if we hit an error return with a variable still active,
        emit an error message */
   v.unknown, v.not_null:
     { return -retv; } ==>
        {
         mc_note(mc_stmt, "Did not free '%s' on error path!",
                                                      mc_identifier(v));
        }
     | { return retv; } ==>
        {
           if(is_err_ret(retv))
                mc_note(mc_stmt, "Did not free '%s' on error path!",
mc_identifier(v));
        }
;
}
```

```
int free_error0(void) {
   int *x = malloc(sizeof *x); /* x=unknown */
   if(x)
      return -1;   /* error: lost x! [x=not_null] */
   else
      return -1;   /* x=stop */
}
```

The assignment to x will put it in the unknown state. The subsequent comparision to null will put it in not_null on the true branch and subsequently give an error when we hit the return of -1 (since x will be lost). On the false branch, x will be in the special stop state, effectively killing it, effectively killing it.

Of course, we can have multiple variables active at once:

```
int free_error1(void) {
    int *x = malloc(sizeof *x);
    int *y = malloc(sizeof *x);
    int *z = malloc(sizeof *x);

    /* lost all pointers!
        [x=unknown, y=unknown, z=unknown] */
    return -1;
}
```

As it stands, the checker will produce a huge number of false positives because of several idioms:

1. Pointers that are passed to cleanup routines that free them.

```
x = malloc(sz);
if(some condition) {
      cleanup(x);
      return -1;
}
```

2. Pointers that, while not freed, are not overwritten if the routine is retried.

```
if(!p)
       p = malloc(sizeof *p);
```

3. Global variables that are cleaned up in non-obvious ways.

Because there are so many errors, we can afford to be aggressive in only reporting the most clear cases. This is a good safety measure becuase the cost of an incorrect fix compared to the benefit of a correct one is enormous: the worst thing that happens with the error is that it will lose storage, whereas if we mistakenly free memory that is still in use, data will be non-deterministically corrupted. Such errors tend to be the worst possible ones to introduce since they are extraordinarily difficult to track down with testing.

Figure 7 shows an updated checker that suppresses these false positives by:

1. Not tracking global variables. If the pattern `v = malloc(sz)` matches, the extension checks if v is either a local or a parameter using the trackable function. Similarly, it stops tracking any variable ever assigned to a non-trackable variable (using the pattern `v1 = v` and subsequent check that v1 is trackable).

2. Stop tracking variables passed as an argument to an unknown function. It finds these using the callout `mc_is_arg(mc_stmt)` which will return 1 if `mc_stmt` is a argument. Unfortunately, callouts do not currently bind to the per-variable, so this call will match not matter what is held in `v` or in `mc_stmt`. Thus, we have to manually check that `mc_stmt` is equivalent to `v` using the comparision `mc_tree_equal`. If this succeeds, we put `v` in the stop state. Otherwise we unmatch. NOTE: this last step is important since if local patterns match, global patterns will not be run.

3. Permentantly killing variables checked for null before being allocated. We do this using the special `killed` state. Once a variable is put in a state named `killed` it will never have a transition done on it, no matter what patterns match.

In practice these suppressions turn the checker into a finder of very easy errors to fix rather than being almost completley unusable.

The following example gives no error messages. The first assignment to `x` will not create a new per-variable state, since `x` is a global (and hence the call to `trackable` will fail). The second assignment will not cause `p` to be tracked since the prevous conditional causes it to be permentantly killed. The third assignment to `y` causes `y` to be tracked, but it is killed by the subequent call to `cleanup`. The final assignment to `y` causes it to be tracked but then killed on the assignment to the global variable `x`.

```
int *x;

int no_errors(void) {
        int *p, *y;

        x = malloc(sizeof *x);  /* will not track */
        if(x)
                return -1;

        if(p)   /* p=killed */
              p = malloc(sizeof *p);

        y = malloc(sizeof *y);   /* y=unknown */
        if(p) {
             cleanup(y);         /* y=stop */
             return -1;
        }

        y = malloc(sizeof *y);  /* y=unknown */
        x = y;                  /* y=stop */
        return -1;
}
```

8 Example: Checking Properties of Functions that Cannot Return

A lot of systems code contains functions that cannot return. When these functions are called, they terminate in a context switch (usually a switch to user context) and they cannot return to the caller. We have written two extensions based on this property. The first looks for functions that make a call to a non-returning function and then have code after that call. This code is never reached, so if it performs any useful function it is a problem. The second extension looks for functions that can

```
{ #include <malloc.h> }
sm_header {
           /* Assume < 0 and null returns are errors */
           static int is_err_ret(mc_tree retv) {
                if(mc_is_integer_cst(retv) && mc_int_val(retv) < 0)
                 return 1;
                if(mc_is_pointer_cst(retv) && !mc_pointer_cst(retv))
                       return 1;
                return 0;
           }
        /* only track parameters and locals. */
        static int trackable(mc_tree t) { return mc_is_parm(t) || mc_is_local_var(t); }
}
sm inv_checker {
   decl any_scalar retv;
   state decl any_pointer v;
   decl any_scalar sz;
   decl any_pointer v1;

   start:
        { (v == 0) } | { v != 0 } ==> { if(trackable(v)) mc_v_set_state(v, killed); }
        | { v = malloc(sz) } ==> { if(trackable(v)) mc_v_set_state(v, unknown); }
        ;
   /* Kill: (1) freed variables, (2) arguments, (3) variables assigned
           to non-trackable vars, (4) null vars.  */
   v.unknown, v.not_null:
      { free(v); } ==> v.stop
    | { v1 = v } ==> { if(!trackable(v1)) mc_v_set_state(v, stop); }
    | { (v == 0) } ==> true = v.stop, false = v.not_null
    | { (v != 0) } ==> true = v.not_null, false = v.stop
    | ${ mc_is_arg(mc_stmt) } ==>
       {
            /* callouts do not bind to variable so we have to do
               the binding ourselves. */
            if(mc_tree_equal(v, mc_stmt))
                 mc_v_set_state(v, stop);
            else
                 mc_unmatch();
        }
    ;
   v.unknown, v.not_null:
     { return -retv; } ==>
        { mc_note(mc_stmt, "Did not free '%s' on error path!", mc_identifier(v)); }
     | { return retv; } ==>
        { if(is_err_ret(retv))
                mc_note(mc_stmt, "Did not free '%s' on error path!", mc_identifier(v)); }
     ;
   v.killed: ==> {}; /* sink state */
}
```

Figure 3: Fancier on-error checker

return down some paths and not return down others. Functions that are inconsistent as to whether or not they return are generally difficult to use correctly because it is not always clear under what circumstances they will return. Although these inconsistent functions are not necessarily bugs, they give useful clues as to what callsites are suspect. By examining calls to these functions by hand, one will often find callers who did not understand that under some circumstances the function they are calling will never return.

Code After Non-returning Call Calls that cannot return can be annotated with the *gcc* attribute `noreturn`. The following code example illustrates the types of errors we would like our extension to find:

```
void pthread_exit (void *__retval) __attribute__ ((__noreturn__));

int err1 () {
  int x = 0;

  if (x < 100) {
    pthread_exit (0);
    dosomethinguseful (); /* ERROR here */
  }

  while (x) {
    dosomethingelseuseful (); /* NO ERROR here */
    x--;
  }
}

void pthread_exit (void *__retval) {
  switchtouser();
  /* Never get here */
}
```

The function `pthread_exit` is a non-returning function because it is marked with the attribute `noreturn`. The function call `dosomethinguseful` is an error because it is dominated by a call to function that cannot return. Thus, there is no circumstance in which the call will ever be executed. The call to `dosomethingelseuseful`, though, is not an error because there is a path through the code in which this call is executed. Thus, we only want to flag errors if there is some piece of code that is dominated by a non-returning call.

Here is the extension that catches this type of error:

```
sm_header {
  #define RET_FLAG 5
  #define NO_RET_FLAG 7

  static mc_tree noret_tree;
  static struct hash_table *no_return_ht;
  static int inited = 0;

#    define noret_err(s) \
            mc_note(mc_stmt, "ERROR:NORETURN:%d:%d: " s " set by '%s'",\
            mc_lineno(mc_tag (e2)),\
            mc_lineno(mc_stmt),\
            mc_identifier(mc_tag (e2)));

#define noreturn_note(s) mc_note(mc_stmt, "NOTE=NORETURN: checking '%s' " \
```

```
                                s, mc_identifier(any_fcall))

#include <libs/mc_process-annotations.h>
}

sm noret_dom {
  decl any_fn_call any_fcall;
  decl any_parm args;
  decl any_expr e;

  init {
    if (!inited) {
      inited = 1;

      no_return_ht = annot_readin (PREFIX "/calculated-no-return.txt", 0, -1, -1);
      pi_assert (no_return_ht, cannot load no return fns);
    }
  };

 start:
  { e } ==> {
    mc_flag_set (e, RET_FLAG);
  } |
  { any_fcall (args) } ==> {
    char* n;
    if ((n = mc_identifier (any_fcall))) {
      if (annot_is_def (no_return_ht, n)) {
        noreturn_note ("hit non-returning function, checking for code afterwards");
        mc_set_state (after_noret);
        mc_flag_set (any_fcall, RET_FLAG);
        noret_tree = any_fcall;
      }
    }
  };
 after_noret:
  { e } ==> {
    mc_flag_set (e, NO_RET_FLAG);
    mc_set_tag (e, noret_tree);
  };
}

sm check_noret flow_insensitive {
  decl any_expr e2;

 start:
  { e2 } ==> {
    if (mc_flag_get (e2, NO_RET_FLAG) && !mc_flag_get (e2, RET_FLAG)) {
      if (mc_lineno (e2) != mc_lineno ((mc_tree) mc_tag (e2)))
         noret_err ("code dominated by no return");
    }
  };
}
```

This analysis is divided into two separate state machines. The first state machine marks every

AST node along every path through the function with the state in which that node was visited. Thus, if the SM is applied down a path without a call to a non-returning function, every AST node along that path will be marked with the flag RET_FLAG. As soon as a call to a non-returning function is found, though, we transition to the after_noret state that tags all AST nodes along the remainder of the path with the NO_RET_FLAG.

The second pass of the analysis does a flow insensitive walk over the tree checking for AST nodes that only appeared along a non-returning path. These AST nodes are marked with the NO_RET_FLAG and not the RET_FLAG.

One additional complication is that we do not want elements of the non-returning call to appear as code after a no return. This is an artifact of the order in which flow insensitive extensions are applied to the AST. To suppress these false error reports, we save off the line of the AST node for the non-returning call in all trees we think are after a no return with mc_tag call. We then make sure that we only flag errors when the line of the code after a no return is different from the line number of the non-returning call itself.

The last element of the extension that we have not discussed is the way in which we identify non-returning calls. These calls are identified using the system's global relaxation capability. This relaxation identifies all non-returning functions and all calls that can reach a non-returning function along some call path. We extract the information generated by this relaxation with the annot_is_def function that is passed the name of the function call we have matched with the pattern any_fcall(args). If the annot_is_def function returns true, the function call we have matched can reach a non-returning function. We then set the current state of the state machine to after_noret using the call mc_set_state. The code within the block prefixed by the keyword init initializes this relaxation scheme allowing us to query the results of the relaxation.

Consistent Non-returning The second extension that deals with the noreturn annotation is the extension that looks for consistency in whether or not a function returns along all paths. This extensions uses the same annotation scheme to identify non-returning calls, but it uses this information in a different way. Here is the extension:

```
sm_header {
  #define RET_FLAG 5
  #define NO_RET_FLAG 7

  #define RETURNS 1
  #define NO_RETURN 2

  #define START 0
  #define AF_NORET 1

  static mc_tree fn_decl;
  static mc_tree noret_tree;
  static unsigned char finish_state;
  static int state = START;
  static int err_reped = 0;
  static int inited = 0;
  static struct hash_table *no_return_ht;

#    define noret_err(s) \
            mc_note(mc_stmt, "ERROR:NORETURN:%d: " s " set by '%s'",\
            mc_lineno(noret_tree),\
            mc_identifier(noret_tree));

#define noreturn_note(s) mc_note(mc_stmt, "NOTE=NORETURN: checking '%s' " \
```

```
                                  s, mc_identifier(any_fcall))

#include <libs/mc_process-annotations.h>
}

sm noret_dom {
  decl any_fn_call any_fcall;
  decl any_parm args;

  init {
    err_reped = 0;
    finish_state = 0;
    fn_decl = mc_get_cur_fn_tree ();
    pi_assert (fn_decl, cannot have fn decl as NULL);

    if (!inited) {
      inited = 1;

      no_return_ht = annot_readin (PREFIX "/calculated-no-return.txt", 0, -1, -1);
      pi_assert (no_return_ht, cannot load no return fns);
    }
  };

  final {
    if ((finish_state == NO_RETURN) &&
        (!mc_fn_is_noreturn (fn_decl)))
      mc_note (fn_decl, "ERROR:NORETURN: Function could have been marked as noreturn.");
  };

 start:
  $end_of_path$ ==> {
    finish_state |= RETURNS;

    if ((finish_state & RETURNS) &&
        (finish_state & NO_RETURN) &&
        !err_reped) {
      err_reped = 1;
      noret_err ("found paths that hit both return and no return");
    }
  } |
  { any_fcall (args) } ==> {
    char* n;
    if ((n = mc_identifier (any_fcall)) && strcmp (n, "Panic")) {
      if (annot_is_def (no_return_ht, n)) {
        noreturn_note ("hit non-returning function, checking for consistency on all paths");
        mc_set_state (after_noret);
        noret_tree = any_fcall;
      }
    }
  };
 after_noret:
  $end_of_path$ ==> {
    finish_state |= NO_RETURN;
```

```
    if ((finish_state & RETURNS) &&
        (finish_state & NO_RETURN) &&
        !err_reped) {
      err_reped = 1;
      noret_err ("found paths that hit both return and no return");
    }
  };
}
```

This extension can report two different errors. The first error reports inconsistency in the results of different paths through the function. The two states in this SM are `start` and `after_noret`. The start state has one pattern that checks all function calls to see if they can reach a non-returning call. This is the same transition as appeared in the first extension we discussed. In this case we set aside the AST tree for the non-returning call so that we can report the location of the non-returning call in our error messages. The other pattern in the `start` state is `$end_of_path`, which matches whenever the end of a path through the current source function is reached. Whenever we hit the end of a path, we mask the value `RETURNS` into the global variable `finish_state`. We then check to see if we have masked both `RETURNS` and `NO_RETURN` into this variable. If we have masked both values into `finish_state` and we have not already reported an error, we report an error because we have just found an inconsistency in the exit behavior of the current function.

The reason that we have found an inconsistency is due to the transition within the state `after_no_ret`, which also uses the pattern `$end_of_path`. If we hit the end of a path in the `after_noret` state, then we mask the value `NO_RETURN` into the global variable `finish_state` because we have found a path through the function that cannot return. Here again, we check to see if both `RETURNS` and `NO_RETURN` have been masked into the variable `finish_state` because this indicates an inconsistency in the return behavior.

In both the `start` and `after_no_ret` states, if we report an error, we set the variable `err_reped` to 1 so that no more errors are reported. We must do this because there is no good way to halt the analysis after we have found the first error.

The second type of error that this extension can report is reported within the block at the beginning of the SM labelled `final`. The code in this block checks to see if the value `RETURNS` was ever masked into the variable `finish_state`. If it was not, that means that the function does not return along any path. In this case, the function should be annotated with the attribute `noreturn`. Thus, if the current function does not have this annotation, we indicate that it should be added.

9 Example: Lock checker

Like many other operating systems, Linux uses locks to protect critical sections of code from race conditions. Most of the Linux code follows a simple locking strategy: locks are acquired and released within the same function. This locking rule is relatively easy to check because it is local and can be captured with a simple SM:

```
sm_header {
#define err(begin_stmt, end_stmt, str)        \
            mc_note(end_stmt, "ERROR from line %d: " str,  \
mc_lineno(begin_stmt))
}

sm simple_locks {
 state decl any_pointer p;
 start: { lock(p) } ==> p.LOCKED
```

```
      | { unlock(p) } ==> p.UNLOCKED
      ;
 p.UNLOCKED:
      { lock(p) } ==> p.LOCKED
    | { unlock(p) } ==> {
        err(mc_v_start_stmt(p), mc_stmt, "Double unlock");
      };

 p.LOCKED:
      { unlock(p) } ==> p.UNLOCKED
    | { lock(p) } ==> {
          err(mc_v_start_stmt(p), mc_stmt, "Double lock");
      }
    | $end_of_path$ ==> {
          err(mc_v_start_stmt(p), mc_last_stmt(), "Exiting with lock held");
      };
}
```

This version of the lock checker is relatively straightforward. The checker starts by matching calls to the `lock` and `unlock` functions, transitioning the lock variable to the `LOCKED` and `UNLOCKED` states, respectively. If a lock variable is unlocked, and a call to `lock` is matched, the machine transitions it to the `LOCKED` state. If another call to `unlock` is matched, we warn about a potential double unlock. Intuitively, if a lock is not held then we can legally lock it, but unlocking it again is probably an error.

The transitions for variables in the `LOCKED` state are duals of the ones in the `UNLOCKED` state: if a lock is held then it is legal to unlock it, but probably an error to lock it again. Finally, the transition on `$end_of_path$` is a special form that matches the end of a path in a function. Since locks are usually acquired and released in the same function, it is probably an error when a lock is held on any exit path through the function, so we give an error.

While intuitive, this lock checker gives many false positives. To see why, consider the following functions:

```
void *p;
int x;

void f1() {
    lock(p);        // transition to p.LOCKED
    unlock(p);      // transition to p.UNLOCKED, no error
}

void f2() {
    lock(p);        // transition to p.LOCKED
    if(x)
        return;     // error: exiting while holding lock p on this path
    unlock(p);      // transition to p.UNLOCKED, no error on this path
}

void f3() {
    lock(p);        // transition to p.LOCKED
                    // error: exiting while holding lock p on this path
}
```

The first function locks and unlocks properly (presumably real code would have a critical section between the lock and unlock). The second function has a locking error: if x is nonzero then the function will return with the lock held. The third function only acquires the lock and never releases it. While technically this could be an error, a more plausible scenario is that this function is meant to acquire the lock because it is itself a “non-primitive” locking function built out of the primitive `lock`. We show how to address this source of false positives in the next section.

There are other sources of false positives for the lock checker. In our experience the following are the most important:

False paths. Sometimes a condition is checked before a lock is acquired and released, for example:

```
if(c)
  lock(p);
...
if(c)
  lock(p);
```

There are only two paths through this code. Either locks are acquired and then released or never acquired at all. This particular case can be handled by MC by turning on the experimental false path pruning by setting the environment variable `FPP` to 1. MC will recognize equivalent conditions in if statements and prune the impossible paths. However, this support is still experimental and will incorrectly prune paths if, for example, the variable `c` is modified between the conditions.

There is also another common type of false path: if code calls `BUG()` or `panic()`, the kernel will halt with an error message and no subsequent code will be executed. These paths can be pruned by prepending a ”kill_paths” SM before the lock checking SM:

```
sm kill_paths flow_insensitive {
   decl any_fn_call fn;
   decl any_parm args;

  start: { fn(args) } ==>
  {
        char *n = mc_identifier(fn);
        if(mc_fn_is_noreturn(fn)) {
                mc_set_path_kill(mc_stmt);
        }
        else if(n &&
        (!strcmp(n, "panic") || !strcmp(n, "bug") || !strcmp(n, "BUG"))){
                mc_set_path_kill(mc_stmt);
        }
  };
}
```

This SM notices calls to functions with conspicuous names such as `BUG` and sets a flag in the flow graph telling subsequent flow-sensitive analysis passes that the current position is equivalent to `$end_of_path$`.

Trylocks. If a lock cannot be acquired immediately the processor trying to acquire the lock will either spin in a tight loop waiting for the lock, or it will suspend the current process and switch to another. An alternative is to have `trylocks`, which either acquire the lock or return failure immediately. There are two common forms of trylocks:

```
int success = trylock1(p); // trylock form 1
void *obj = trylock2(); // trylock form 2
```

The first form takes a lock variable and returns 1 if the lock is acquired or 0 if it isn’t. The variable must be tested before the code can assume the lock is held. The second form returns a pointer to a locked object on success, or `NULL` on failure. In Linux, trylocks are seldomly used, at least when compared with regular locks.

More complex lock usage. Lock acquisition and release need not occur within the same function, and sometimes this assumption is just plain wrong. It is still possible to analyze this form of lock usage with global analysis. Fortunately, most of the Linux kernel carefully avoids using locks in a complex way. By finessing this issue, we trade off a few false positives (and perhaps false negatives) for a great deal of simplicity. Also, nonstandard lock usage is very difficult to diagnose. There may be complex corner cases that make the code right, even if it looks wrong to experienced kernel developers.

9.1 An improved lock checker

As mentioned in the previous section, some false paths can be handled by turning on experimental false path pruning, and others by including a preprocessing to eliminate path traversal after encountering fatal functions. These solutions are adequate for now, so we will not discuss false paths any further in this section.

Even with many false paths eliminated, we can still do better. The following checker eliminates many false positives caused by non-primitive locking functions:

```
sm_header {
#    define err(begin_stmt, end_stmt, str)        \
mc_note(end_stmt, "ERROR from line %d: " str, \
mc_lineno(begin_stmt))
    static int end_states;
}

sm simple_locks {
 state decl any_pointer p;
 init { end_states = 0; };

 start:
      { lock(p) } ==> p.LOCKED
    | { unlock(p) } ==> p.UNLOCKED;

 p.LOCKED:
      { lock(p) } ==> {
          err(mc_v_start_stmt(p), mc_stmt, "Double lock");
      }
    | { unlock(p) } ==> p.UNLOCKED;

 p.UNLOCKED:
      { lock(p) } ==> p.LOCKED
    | { unlock(p) } ==> {
        err(mc_v_start_stmt(p), mc_stmt, "Double unlock");
      };

 p.LOCKED, p.UNLOCKED:
      $end_of_path$ ==> {
          end_states |= (1 << mc_v_get_state(p));
          if((end_states & (1 << LOCKED)) && (end_states & (1 << UNLOCKED)))
    {
mc_tree first_stmt = (mc_tree) mc_v_start_stmt(p);
err(first_stmt, mc_last_stmt(), "Inconsistent lock");
    }
      };
```

```
}
```

This checker is based on a consistency principle. Instead of blindly looking for functions that exit holding a lock, it looks for functions that can exit with locks acquired along some paths but released along others. It does this by keeping a static variable `end_states` that records the state of the SM at every `$end_of_path$`. If a lock is `LOCKED` at the end of one path and `UNLOCKED` at the end of another, the checker gives an error.

Now consider the sample input again:

```
void *p;
int x;

void f1() {
    lock(p);        // transition to p.LOCKED
    unlock(p);      // transition to p.UNLOCKED, no error
}

void f2() {
    lock(p);        // transition to p.LOCKED
    if(x)
        return;     // p.LOCKED on this path
    unlock(p);      // error: p.UNLOCKED on this path, p.LOCKED on the other.
}

void f3() {
    lock(p);        // transition to p.LOCKED, no error
}
```

The first function is still ok because it exits in the `p.UNLOCKED` state only. The second function gives an error because it can return with `p.LOCKED` if `x` is nonzero, or `p.UNLOCKED` if x is zero. The third function does not trigger an error because it can only exit in `p.LOCKED`. This was an error case for the simple checker but it passes the consistency check: it cleanly transitions the the lock `p` from unlocked to locked.

A refinement of this checker can also handle trylocks and user-supplied lists of locking functions. Finally, notice that since the checker "discovers" functions that cleanly acquire or release locks, it can print these out and then re-use this information for future checks, thus catching more locking functions. This enhancement is currently being tweaked.

10 The MC Standard Library

The MC provides a set of library routines used to query matched trees: what is the name of a matched identifier, what is its size, what is a call's first argument (and how many arguments does it take) is a tree a for-loop, etc. Ideally, users could ask any question that one could formulate about source code. However, currently the interface is still partially completed. Below we outline a subset of the library functions. Since the interface is constantly being augmented, the header file `mc_tree.h` should be consulted for the full listing.

```
void (compile(void cspec code, int type-spec
              [, int type-size])) (void)
```

`jump` returns the cspec of a jump to `target`. This function is useful for constructing "hardcoded" finite state machines.

10.1 Tree interogation routines

`char *mc_identifier(mc_tree var)`

Returns the name of `var`, which must be some sort of variable. BUG: NULL can be returned if the routine does not know how to access the name.

`mc_tree mc_type(mc_tree t) char *mc_type_name(mc_tree type)`

`mc_type` returns the type of tree `t`, and `mc_type_name` will return its string representation. Currently does not handle all tree nodes very well.

`int mc_is_integer_cst(mc_tree t)`
`int mc_int_val(mc_tree t)`
Is `t` an integer constant and, if so, what is its value?

`int mc_is_var(mc_tree t)`
`int mc_is_local_var(mc_tree t)`
`int mc_is_global_var(mc_tree t)`
`int mc_is_parm(mc_tree t)`
`int mc_is_const(mc_tree var)`
`int mc_is_static(mc_tree var)`
Is `t` a variable, a local variable, a global, a parameter, a constant or a variable with static storage?

`int mc_is_call(mc_tree call)`
`int mc_is_arg(mc_tree arg)`
`int mc_is_call_to(mc_tree call, char *name)`
`int mc_is_call_arg(mc_tree call, mc_tree t)`
`int mc_call_is_void(mc_tree call)`
Functions to answer: Is tree `call` a function call? Is `arg` a function argument? Is `arg` an argument to `call`? Is it a call of type `void`? (The latter should be derivable from other routines, but is not.)

`int mc_is_noreturn(mc_tree call)`
`int mc_is_pure(mc_tree call)`
`int mc_is_const(mc_tree call)`
`char *mc_section_name(mc_tree call)`
Does the function invoked by `call` have the GCC modifier `no-return`, `pure`, or `const`? The last function returns the section that the function has been placed in (if any was given). It's a hack, but that's because it interrogates the GCC attributes, so doesn't have much choice.

`int mc_num_args(mc_tree call)`
`int mc_nth_arg(mc_tree call, int n)`
`int mc_get_call_name(mc_tree call)`
`int mc_get_call_arg(mc_tree call, mc_tree arg)`
`char *mc_arg_to_call(mc_tree arg)`
`mc_num_args` returns the number of arguments to `call`. `mc_nth_arg` returns the `nth` argument to `call`. `mc_get_call_name` returns NULL if it `call` is not a call or has no name, or it's name otherwise. `mc_get_call_arg` the index of argument `arg` in `call` or -1 if it is not. `mc_arg_to_call` returns the name of the function `arg` is an argument for, or NULL otherwise (which allows it to be used as an implicit check that `arg` is an argument).

10.2 State machine manipulation routines

States and state variables have a variety of support routines. These routines have the restriction that they must be called within the SM body itself, and not routines that the SM calls.

`mc_set_state(state)`
`mc_set_path_state(true_state, false_state)`

Set the current global state to `state` or, for `mc_set_path_state` to `true_state` on the true path and `false_state` on the false path.

`mc_v_set_state(v, state)`
`mc_v_set_path_state(v, true_state, false_state)`

Set the current state of state variable `v` to `state` or, for `mc_v_set_path_state` to `true_state` on the true path and `false_state` on the false path.

`int mc_v_exists(v)`
`int mc_v_get_state(v)`
`mc_tree mc_v_start_stmt(v)`

Respectively: Does `v` exist as a state variable? What state is `v` in? What state did `v` start in?

`mc_v_set_data(v, void *data)`
`void *mc_v_get_data(v)`

Set and get a single `void *` pointer associated with `v`.

10.3 Interrogate enclosing function

`char* mc_function_name(void)`
`char* mc_function_tree(void)`
`char * mc_file_name(void)`
`char *mc_cur_file_name(void)`

Get the name or tree of the current function. The file routines get the name of (1) the source file the compile line contains and (2) the gets the current file that is being processed, which changes as include directives are processed.

`int mc_param_num(mc_tree param)`
`int mc_n_params(mc_tree fn)`

What parameter is `param`? How many parameters does the function `fn` have?

10.4 Annotation routines

`void mc_set_stmt_type(mc_tree s, char *type)`

Set the type of the tree `s` to `type`, which is a character string (e.g., `''ERROR''`, `''SECURITY''`). This annotation will be printed out whenver the tree or a tree containing it is the first argument to `mc_note`. It is usually used by annotation passes that search over client code for "interesting" actions and tag them. Later extensions can then transparently have these annotations emitted with their error messages.

```
unsigned mc_get_stmt_type(mc_tree s, char *type)
```

Returns `1` if the statement (or a tree it contains) has been marked with `type`.

```
char **mc_stmt_types(mc_tree sn)
```

Returns a `NULL` terminated array of strings containing all types that `s` and its subtrees have been annoted with.

```
void mc_set_block_annot(mc_block b, int x)
int mc_get_block_annot(mc_block b)
```

Each block has a single integer flag that can be set — `mc_set_block_annot` sets the flag in block `b` to `x` and `mc_get_block_annot` retrieves it. These functions should be replaced with ones similar to those used on trees.

10.5 Navigation and block interrogation routines

```
mc_tree mc_kid(mc_tree t, n)
mc_tree mc_kid0(mc_tree t) mc_tree mc_kid1(mc_tree t)
```

`mc_kid` returns the `n`th child of tree `t`; `mc_kid0` returns the 0th one, and `mc_kid1` the 1st one. `NULL` is returned if there are no kids. You probably shouldn't need these functions directly — they are almost always used to get around matcher problems.

```
mc_block mc_tree_to_blk(mc_tree t)
```

Returns the block containing tree `t`.

```
int mc_end_block(mc_tree t)
```

Is tree `t` the last on in a block?

```
void mc_set_block_visited(mc_block b)
int mc_block_visited_p(mc_block b)
```

`mc_set_block_visited` sets a bit in block `b` indicating its been visisted, `mc_block_visited_p` returns 1 if it is set, 0 otherwise. This is mainly used during block traversals to avoid infinite loops caused by backedges.

```
void cspec jump(void cspec target)
```

11 Language Reference

This section provides a quick reference manual for all of the syntactic features of *metal* that we have seen in the examples in preceding sections. The manual is organized as a series of code templates and example instantiations of those templates.

1. `{ ... }`
   When a bracketed region appears at the very top of the SM definition, this bracketed region specifies that its contents should be copied into the `.p` file created by the *metal* compiler. The effect of adding something to this region is that the code contained in the region will be added to the top of every file in the source that is compiled with the extension. This capability is generally used to add preprocessor directives to the source like a `#include` that redefines some of the macros in the source, or a `#define` that overrides one of the `#define`s in the source that was causing problems for the analysis. For example, most of our checkers that are run over Linux include the file `mc-linux.h` in a bracketed section at the beginning of the SM. This looks like:

   ```
   {
     #include <linux/mc-linux.h>
   }
   ```

   This allows us to redefine some Linux macros into functions so that we can match them with function call wildcards.

2. `sm_header { ... }`
   The construct above specifies a section of the state machine that should be copied verbatim into the `.c` translation of the SM. Thus, this section is where all helper functions that the SM uses should be defined, all global variables should be declared, etc. The code between the two curly braces can be arbitrary, compilable C code. For example, recall from Section 8 the SM that looked for functions whose returning/non-returning behavior was inconsistent contained the following header:

   ```
   sm_header {
     #define RET_FLAG 5
     #define NO_RET_FLAG 7

     #define RETURNS 1
     #define NO_RETURN 2

     #define START 0
     #define AF_NORET 1

     static mc_tree fn_decl;
     static mc_tree noret_tree;
     static unsigned char finish_state;
     static int state = START;
     static int err_reped = 0;
     static int inited = 0;
     static struct hash_table *no_return_ht;

     #    define noret_err(s) \
           mc_note(mc_stmt, "ERROR:NORETURN:%d: " s " set by '%s'",\
           mc_lineno(noret_tree),\
           mc_identifier(noret_tree));

     #define noreturn_note(s) mc_note(mc_stmt, "NOTE=NORETURN: checking '%s' " \
                                s, mc_identifier(any_fcall))

     #include <libs/mc_process-annotations.h>
     }
   ```

This header defines several constants and global variables that are used by the analysis. It also defines two helper macros, and it includes the header file for the global analysis required by this checker.

3.
```
sm <name> <analysis type> { ... }
```
The construct above is the general form of an SM definition. The SM begins with the line containing the key word **sm**, an identifier that is the name of the sm (**<name>**), and, optionally, one of the keywords that specifies the type of analysis. If no analysis type is specified, the type defaults to flow sensitive. The other types of analysis that the extension writer can specify are **flow_insensitive**, **printer**, and **splicer**. As we discussed in earlier sections, a flow insensitive walk analyzes each AST node in the function sequentially without any regard to control flow. The **printer** and **splicer** modes are intended for analyses that emit code. The **printer** emits code to standard output, whereas the **splicer** will splice the emitted code into the original source file so that the result is a compilable **.c** file. SMs of these two types can currently only have one state.

4.
```
init { ... }
```
The construct above allows the extension writer to specify code executed at the beginning of each function before the analysis begins. This construct is often useful if the extension requires some initialization before each function. It is also often used for global initialization by combining code in the **init** block with a global variable that ensures that the code within the block is only executed once. We see this usage of the **init** block frequently in analyses that use *xgcc*'s global relaxation capabilities. The following is an example init block from the no-return consistency analysis from Section 8.

```
init {
  /* These are global variables that must be reset before each
     function. */
  err_reped = 0;
  finish_state = 0;
  fn_decl = mc_get_cur_fn_tree ();
  pi_assert (fn_decl, cannot have fn decl as NULL);

  /* The following code should execute once per run of xgcc.  We
     use the static variable inited to ensure that this code is only
     executed once. */
  if (!inited) {
    inited = 1;

    no_return_ht = annot_readin (PREFIX "/calculated-no-return.txt", 0, -1, -1);
    pi_assert (no_return_ht, cannot load no return fns);
  }
};
```

5.
```
final { ... }
```
The **final** block, which has the form above, is analogous to the **init** block except that the code within this block is executed once per-function after the entire analysis has completed. It is often used to update some bookkeeping as in the no-return consistency checker from Section 8. It has the same syntactic form as the **init** block above.

6.
```
decl <wildcard> <identifier>;
```
The construct above is used to declare a *metal* variable of wildcard type. The wildcard types are **any_fn_call**, **any_expr**, **any_stmt**, **any_parm**, **any_arguments**, **any_scalar**, and **any_pointer**. The semantics of these wildcards are discussed in the Section 3. For example, the following code declares a wildcard variable **ptr** of type **any_pointer**.

```
decl any_pointer ptr;
```

7. `decl <optional keyword ''literal''> { <arbitrary C type> } <identifier>;`
   The declaration above declares a *metal* variable of arbitrary C type. This type is specified between the brackets in the declaration. For example:

   ```
   decl { int* } intp;
   ```

   declares a *metal* variable `p` of type `int*`. `p` will only match other variables that are compatible with (i.e. assignable to without causing a compiler error) the type `int*`.

   If the type specified between the brackets is some variant of integer, real value, or string, then the keyword "literal" can be optionally specified before the opening bracket. This keyword means that this variable should only match constant values of the appropriate type. For example:

   ```
   decl literal { int } number;
   ```

   will only match actual integer constants.

8. `state decl ...`
   The keyword state can be appended to any valid declaration. These declarations can have the form specified in either of the previous two items. Attaching the keyword `state` indicates to *metal* that the variable should have an attached per-variable state. For an explanation of per-variable states see Section 5.

9. `pat <name> = { ... } | (optional) { ... };`
   The code above declares a pattern variable. The pattern name can be any arbitrary identifier. The pattern variable must be initialized to some code pattern. This code pattern can either be a single bracketed code pattern, or an arbitrarily long sequence of code patterns separated by the punctuation | to indicate that the pattern is the or of each individual bracketed pattern. The list of bracketed patterns must be terminated by a semi-colon no matter how many patterns are in the list. The following two examples illustrate pattern variable declarations:

   ```
   pat fn_foo = { foo () };
   pat int_comparison = { x < y } | { x > y }  |
      { x <= y } | { x >= y } | { x == y };
   ```

   The first pattern will match all calls to the function `foo`. The second will match any comparisons between two variables `x` and `y`, where we assume that `x` and `y` have been declared as *metal* variables with comparable types using a `decl` statement.

10. ```
    <state>:
    <transition> |
    <transition> |
    <transition>;
    ```
    The construct above defines a new state. The state name can be any arbitrary identifier. The state is followed by a sequence of one or more transition terminated by a semi-colon. The various forms that these transitions can take are described in the subsequent items. If there is more than one transition, each transition must be separated from the next by the | operator. As described in Section 5, there are two types of states: global and local states. Global states are specified by an arbitrary identifier, whereas local states are specified with the name of a *metal* variable declared with the `state decl` construct followed by a period, then an arbitrary identifier. Thus,

    ```
    intr_enabled:
    ```
    ```

is a global state whereas

```
v.null:
```

is a local state if the following declaration appears earlier in the SM:

```
state decl any_pointer v;
```

Note that the variable v can have any arbitrary type, not just type any_pointer.

11. <pattern> ==> <new state>, { <code> }
The above construct is the most common form of a state transition. The left hand side of the transition operator, ==>, is a pattern. A pattern can either be a pattern variable defined as shown in the previous item, or a bracketed fragment of arbitrary C code. The state on the right-hand side can be any of the states defined in the SM using the state definition construct specified in the previous item. The code can be any arbitrary bracketed C code that the extension writer wants to execute each time this particular state transition occurs. The following is a sample transition from the state intr_enabled to the state intr_disabled on a call to cli. The SM action for this transition prints a note saying that a transition occurred:

```
intr_enabled: { cli () } ==> intr_disabled,
   { mc_note (mc_stmt,``NOTE:transition from enabled to disabled''); }
```

Note that whitespace is ignored in all state definitions with the small exception that the C code actions must follow C's rules for whitespace (i.e. string constants cannot span multiple lines).

12. <pattern> ==> { <code> }
The construct above is another form of a transition. In this case, the state of the SM does not change. All that happens when the pattern is matched is the C code action is executed.

13. <pattern> ==> <state>
In this case, there is no action, all that happens on transition is the state of the SM changes.

14. <pattern> ==> true=<true state>, false=<false state>, { <code> } (optional)
The above construct is used when the pattern is any sort of conditional pattern and the analysis is flow sensitive. In this case, on the path where the condition evaluates to true, the SM will transition to the state <true state>. On the path where this condition evaluates to false, the state will transition to <false state>. This type of transition can also have an arbitrary C code action, although the action is not required. The true and false states can be either global or local states. The following example transitions to the v.null state on the true branch when the pointer variable v is compared to 0 and the v.not_null state on the false branch of the comparison to 0. The transition has no action:

```
{ (v == 0) } ==> true = v.null, false = v.not_null
```

12 TODO for A Crash Course on Metal

This section is woefully incomplete right now, but it will be expanded and improved. Hopefully the examples worked out in class will give some feel for what features are available and how to use them.

Other topics/features include:

1. Including headers for pattern matches
2. What exactly can be in a pattern? How can I tell what it will match?
3. Path variables
4. State variables
5. Special pattern holes (e.g. any_pointer)
6. Per-variable state
7. begin/end
8. Library functions for getting information out of trees
9. Global analysis (?)

References

[1] A. Chou, B. Chelf, D.R. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[2] A. Chou and D.R. Engler. Metal: A language and a system for building lightweight, system-specific software checkers, analyzers, and optimizers. Submitted for publication, available upon request: acc@cs.stanford.edu, November 2000.

[3] D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. To appear in Operating Systems Design and Implementation (OSDI) 2000, September 2000.