A System and Language for Building System-Specific, Static Analyses

Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler
Stanford University

ABSTRACT

This paper presents a novel approach to bug-finding analysis and an implementation of that approach. Our goal is to find as many serious bugs as possible. To do so, we designed a flexible, easy-to-use extension language for specifying analyses and an efficent algorithm for executing these extensions. The language, *metal*, allows the users of our system to specify a broad class of analyses in terms that resemble the intuitive description of the rules that they check. The system, *xgcc*, executes these analyses efficiently using a context-sensitive, interprocedural analysis.

Our prior work has shown that the approach described in this paper is effective: it has successfully found thousands of bugs in real systems code. This paper describes the underlying system used to achieve these results. We believe that our system is an effective framework for deploying new bug-finding analyses quickly and easily.

Keywords

Extensible compilation, error detection.

General Terms

Reliability, Security, Verification.

Categories and Subject Descriptors

Software [**Software Engineering**]: Coding Tools and Techniques

1. INTRODUCTION

This paper describes the implementation of an unusual approach to finding bugs that we call metacompilation (MC). The focus of our approach is pragmatism: we want to find as many serious bugs as possible. We do so using programmer-written compiler extensions (checkers). This paper presents a language, *metal*, for implementing these extensions, and an analysis engine, *xgcc*, that executes extensions using a context-sensitive, interprocedural analysis.

PLDI'02, June 17-19, 2002, Berlin, Germany.

The main barrier to finding bugs is simply knowing the correctness rules that code must obey. The more rules you can check, the more bugs you will find. Thus, we designed *metal* to be (1) easy to use and (2) flexible enough to express a broad range of rules within a unified framework. *Metal* must be easy to use since many rules are known only to programmers; if they cannot write extensions, we cannot check these rules. Thus, *metal* is designed for system implementers, not compiler writers. *Metal* must be flexible because we want to check arbitrary rules. We do not want a system that is limited to checking a specific set of properties (e.g., synchronization constraints; temporal rules) or a specific underlying assumption (e.g., "the analysis must be conservative").

Metal is easy to use because it provides the state machine (SM) as a fundamental abstraction. State machines are an easy abstraction because they are a familiar concept in systems programming. *Metal* is flexible because it allows the extension writer to enhance the SM abstraction in near-arbitrary ways with general-purpose code. *Metal*'s flexibility allows extensions to make the analysis rule-specific without modifying the language or the underlying system.

Our prior work has shown that *metal* works well. It requires little investment to get results: a day's work can produce an extension that finds tens or even hundreds of serious errors in actual code. Further, extensions are small — usually between 10 and 200 lines of code, depending mostly on the amount of error reporting that they do. *Metal*'s flexibility is demonstrated by the fact that we were able to write over fifty checkers that express significantly different types of analyses including: (1) finding violations of known correctness rules [1, 9] and (2) automatically inferring such rules from source code [10]. We describe *metal* in Sections 2 through 4.

We have three main requirements for *xgcc*; it must: (1) provide the analysis needed to find bugs, (2) not significantly restrict what *metal* extensions can do, and (3) scale to large programs. Our ideal division of labor is that extensions encode only the property to check, leaving the details of how to check the rule to *xgcc*. The second and third requirements are important since the more rules we check and the more code we analyze, the more bugs we will find. The main restriction that *xgcc* places on extensions is determinism; they can otherwise perform arbitrary computations internally. In this paper, we present the analysis algorithm, implemented in *xgcc*, that executes our extensions. We describe *xgcc* in Sections 5 and 6.

In Section 7, we discuss the approximations that our

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:    { err("using %s after free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:    { err("double free of %s!", mc_identifier(v)); }
9: ;
```

Figure 1: Free Checker

```
1:int contrived(int *p, int *w, int x) {
2:    int *q;
3:
4:    if(x)
5:    {
6:        kfree(w);
7:        q = p;
8:        p = 0;
9:    }
10:   if(!x)
11:       return *w;  // safe
12:   return *q;      // using 'q' after free!
13:}
14:int contrived_caller (int *w, int x, int *p) {
15:   kfree (p);
16:   contrived (p, w, x);
17:   return *w;      // using 'w' after free!
18:}
```

Figure 2: Free Checker Example Code

analyses make and their implications. Section 8 discusses several analysis techniques for handling false positives including a simple, path-sensitive analysis for eliminating nonexecutable paths. Section 9 continues the false positive discussion by presenting the ways in which *xgcc* ranks error reports. Finally, Section 10 discusses related work and Section 11 concludes.

2. OVERVIEW

Our extensions are written in *metal*, a language for expressing a broad class of customized, static, bug-finding analyses. The common thread among these analyses is that they all exploit the fact that many abstract program restrictions map clearly to source code actions [9]. 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 [10].

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 constraint. *Metal* organizes extensions around a state machine (SM) abstraction. State machines are a concise way to represent many program properties. Note that the SM abstraction provides sugar for common operations, it does not limit extensions to checking finite-state properties. When needed, extensions can be augmented with general-purpose code. *Metal* extensions are executed by the interprocedural analysis engine, *xgcc*.

Figure 1 shows the free checker that flags when freed pointers are dereferenced or double-freed. We use this checker and the code example in Figure 2 throughout the paper. The extension will find two errors in the example (lines 12 and 17).

2.1 Metal Extensions and State Machines

Metal extensions define a collection of one or more state machines. During execution of an extension, the current state of the extension is simply the combination of all the current states of the underlying state machines that the extension defines. Each of these state machines is logically separate: transitions in one SM do not affect any of the others. The number of state machines grows and shrinks during the course of the analysis.

Each individual SM's current state consists of one *global* state value and one or more *variable-specific* state values. Global state values capture a program-wide property (e.g.,"interrupts are disabled"). Variable-specific state values capture program properties associated with specific source objects (e.g., "pointer `p` is freed").

Each state *value* defined above is assigned to an *instance* of a state *variable*. Each extension defines one global state variable and, optionally, a variable-specific state variable. For simplicity, the discussion assumes that an extension has exactly one of each. A state variable has one or more instances, each of which is assigned a state value. The global state variable has exactly one instance that persists throughout the analysis. The variable-specific state variable has one instance for each program object with an attached state. The number of such instances grows and shrinks as the analysis decides to track new program objects and ignore previously tracked objects. An SM state consists of the value of the global instance and the value of one of the variable-specific instances. Thus, the number of SMs defined within each extension at a given point in the analysis is equal to the number of program objects with attached state.

In the free checker, the variable-specific state variable, `v`, is declared with the keywords `state decl`. The notation *v.freed* means that the state value *freed* is bound to `v`. Thus, only instances of `v` can be assigned the value *freed*. The global state variable is implicitly-defined. The state value *start* is bound to the global state variable because it has no explicit binding.

The alphabet of each SM is defined by the *metal patterns* used within the extension. Patterns are used to identify source code actions that are relevant to a particular rule. The free checker uses patterns to recognize deallocations (using the pattern "{`kfree(v)`}") and dereferences of deallocated variables (using the pattern "{`*v`}"). The variable `v` in these patterns will match pointers of any type.

Each state value defines a list of transitions. In the free checker, the *start* state defines a single transition rule and the *v.freed* state defines two. The transition for the *start* state (line 3) says that when the global instance has the value *start* and the current program point matches the pattern {`kfree(v)`}, a transition should execute that attaches the state *freed* to the abstract syntax tree (AST) matching `v` (i.e., the freed pointer). The transition from *start* to *v.freed* is a special type of transition that creates a new instance of `v` and, thus, a new state machine.

The *v.freed* state value has two transition rules: the first triggers when a freed variable is dereferenced, and the second triggers when a freed variable is freed again. Both transitions print an error message that describes the error and identifies the particular variable to which the erroneous action was applied. A transition that begins in a variable-specific state value is triggered by a specific instance of the state variable bound to that value. Thus, the two transitions in the *v.freed* state are triggered when one of the freed variables that the extension is tracking is either double-freed or dereferenced. These transitions update the value of the instance that triggered the transition to the special value *stop*. When an instance is assigned the value *stop*, the state machine tracking that instance is removed from the extension's collection of SMs. However, if the variable associated with the instance is freed again, the transition in the *start* state will execute and thus reinstantiate the deleted SM.

The initial state of an extension contains one state machine that expresses the fact that nothing is known about the program at the start of the analysis. Thus, the global state variable in the free checker initially has the value *start*, and `v` has the special value <> that reflects the fact that the extension does not know about any freed variables.

xgcc applies an extension to the control flow graph (CFG) for a single function in depth-first order, one execution path at a time, beginning at the entry points to the callgraph for the source base. At each program point, the extension looks for executable transitions in any of the current SMs. After iterating over all the SMs, the analysis moves on to the next program point. As described in Section 8, *xgcc* also enhances the extension with additional analysis to prune non-executable paths, follow simple value flow, and delete the state attached to an expression that is redefined.

2.2 Execution of the Free Checker

We tie all of these pieces together by following the execution of the free checker on the example in Figure 2.

1. Line 14: `contrived_caller` has no known callers and is, thus, an entry point to the callgraph for our example. We assume that none of the input parameters are aliased. The extension begins in the initial state.

2. Line 15: The `kfree` call will match the pattern in the *start* state and the transition on line 3 of the checker will execute, attaching the state *freed* to `p`.

3. Line 16: *xgcc* follows the call to `contrived`, tracking the variable `p` because it is passed as a parameter.

4. Line 4: The analysis splits down the true and false paths, following the true path first. When the analysis splits, a separate copy of the extension's state is applied to each path. The analysis tracks that `x` equals 0 and is not equal to 0 down each respective path.

5. Line 6: The call to `kfree` places `w` in the *freed* state. At this point, there are two instances of `v` with the value *freed*: `p` and `w`.

6. Line 7: The assignment causes *xgcc* to transparently create another instance of `v` for the variable `q`, also in the *freed* state.

7. Line 8: The assignment to variable `p` causes *xgcc* to transition `p` to the *stop* state, removing `p` from the extension's state.

8. Line 10: Rather than splitting at the conditional, *xgcc* uses the information that `x` is non-zero on this path to prune the true branch. If the true branch were followed, there would be a false error report at line 11 because `w` has attached state *freed* (line 6).

9. Line 12: The dereference pattern for *v.freed* matches `*q` and reports a use-after-free error. `q` is transitioned to the *stop* state. After analyzing the return, the analysis backtracks to follow the false branch from line 4.

10. Line 10: Rather than splitting at the conditional, *xgcc* uses the information that `x` is equal to 0 on this path to prune the false branch.

11. Line 11: The path ends. We have explored all paths through `contrived`.

12. Line 17: Control returns to the caller. The set of outgoing instances of `v` is the union of all instances active at the exit from any path through `contrived`. There are two such instances, `p` and `w`, active at lines 11 and 12, respectively. The extension flags an error at the subsequent dereference on line 17.

The next two sections describe *metal* in more detail.

3. METAL STATES AND TRANSITIONS

3.1 Metal States

Each state variable's domain consists of all the state values bound to that variable. This section elaborates the discussion of state variables and provides a more precise definition of the extension's state and each state machine within it. The definition of extension state that we describe here is translated to the data structures described in Section 5 that define an extension from *xgcc*'s perspective.

The extension must be allowed to extend the state space using general-purpose code. The advantage of this form of flexibility is that it allows our extensions to express properties where the state space is defined dynamically.

We allow extensions to grow the state space by extending the *domain* of each instance within general purpose code. For this reason, we enhance each variable-specific instance with a data value that is a C structure of arbitrary size that the extension can manipulate within the escapes to C code. Extensions may also update the value of the global instance directly within an escape to C code to allow more complex transitions.

An extension's state is defined as a set of *state tuples*, each of which corresponds to a single SM contained within that extension. A state tuple has one component that is filled by the value of the global instance. In the free example, this slot always contains the value $start$. The second component contains the value of a variable-specific instance (e.g., an instance of `v` in the free checker). For example, after analyzing line 15 in Figure 2, the free checker's state would include the tuple $(start, \texttt{v} : \texttt{p} \mapsto freed)$ because the state variable `v` has an instance attached to the program object `p` whose value is *freed*.

While the state tuples in this paper have only two components, the actual implementation of *metal* allows the extension to define tuples with additional components. The actual implementation of the algorithms in this paper handles the more general case.

3.2 Metal Transitions

A simple *metal* transition consists of a source state value, a pattern, and a destination state value. The transition on line 3 of the free checker follows this template. The extension

```
state decl { lock_t } l;

start:
    {trylock(l) != 0} ==> true=l.locked, false=l.stop
  | {trylock(l) == 0} ==> true=l.stop, false=l.locked
  | {lock(l);} ==> l.locked
  | {unlock(l);} ==>
   { err("%s is not locked", mc_identifier (l)); }
  ;

l.locked:
    {lock(l);} || {trylock(l)} ==>
       { err("dbl. lock of %s", mc_identifier (l)); }
  | {unlock(l);} ==> l.unlocked
  | $end_of_path$ ==>
       { err("path ends with lock held"); }
  ;
```

Figure 3: Lock checker

determines which transitions to execute by iterating through both global and variable-specific instances and determining whether the value of each instance defines a transition that can execute. A transition can execute if its pattern matches at the current point in the analysis. An instance cannot trigger a transition at the statement where that instance was created; this restriction prevents a variable that is freed for the first time from triggering a double-free error at the same program point. Simple transitions can be enhanced with path-specific destination states and C code actions.

Path-specific transitions. Path-specific transitions allow the extension to track the value of simple boolean predicates (e.g., `l` is locked, `p` is null) or model functions that can have two possible outcomes. If a transition occurs at a branch condition in the source code, the extension can specify a different destination state depending on whether the analysis follows the true branch or the false branch from the condition. Figure 3 shows the lock checker, which warns when locks are (1) released without being acquired, (2) double acquired, or (3) not released at all. The routine `trylock`, used for nonblocking lock acquisition, returns 1 if it acquires the lock and 0 otherwise. Thus, in the first transition, we attach the state *locked* to the lock on the true path, and the state *stop* to the lock on the false path. The special pattern `$end_of_path$` in the last transition evaluates to true when either an instance of `l` in the *locked* state permanently leaves scope or when the program terminates.

C Code actions. Transitions can include C code *actions* that execute whenever the transition executes. Actions are another way that an extension can extend the basic SM abstraction. C code actions allow the extension to perform arbitrary computations whenever a transition executes. We describe two types of actions that we have found useful: those that perform complex error reporting and those that enhance the analysis machinery.

To make error messages useful, checkers must report not only *what* the error was, but also *why* the error occurred. Thus, all of our checkers track the calculations that found each error. These calculations depend on the particular characteristics of the extension. The code to track why an error was flagged accounts for the bulk of each extension.

In [10], we describe several checkers that use statistical analysis to infer checking rules. For example, to infer whether routines `a` and `b` must be paired: (1) assume that they must, (2) count the number of times they occur together and (3) count the number of times they do not (rule violations). The reported violations are then sorted using a statistical significance test. We implemented this functionality by using the C code actions to count the correct pairings and violations during the analysis. (Section 9 uses the same technique to rank rule violations.)

By default, a *metal* extension has a finite, statically determined domain for each state variable. The extension can extend this model by using C code actions to manipulate the extension's state directly using *xgcc*'s internal interface. For example, we could extend the lock checker described above to handle recursive locks by using the data values in each instance of `l` to track the current depth of the lock. Whenever a lock operation or an unlock operation occurs, the resulting transition could either increment or decrement the lock depth within the C code action. If this depth ever went below 0 or exceeded a small constant, the extension would report an incorrect lock pairing.

Composition is another mechanism extensions can use to enhance the SM model. Extensions can be composed such that each extension uses the results of the previous one in its own analysis. Extensions implement this composition by using *xgcc*'s internal interface to annotate the ASTs with arbitrary data values. Subsequent extensions can retrieve and use these values. One common use of composition is the *path-kill* extension [10], which flags all calls to `panic` so that subsequent analyses will not report errors on paths dominated by these calls. When a subsequent extension sees a flagged function call, it stops traversing the current path.

4. METAL PATTERNS

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 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 5 in the free 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. To match all pointer dereferences in the free checker, 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 pat-

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.

tern, 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 8 in the free checker refers to the AST for the freed pointer matched on line 7.

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 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.

5. INTRAPROCEDURAL ANALYSIS

This section describes our intraprocedural algorithm that applies *metal* extensions to a source base. The goal of this algorithm is to execute checkers efficiently without compromising *metal*'s flexibility.

Extensions are applied to each AST in a single path in execution order. Execution order means that the tree for each individual statement is visited in the order that the corresponding instructions would execute. For example, a function call's arguments are visited before the call; an assignment's right-hand side is visited first, then the left-hand side, then the assignment. We refer to AST nodes as program points. At each program point, the extension decides whether to execute any transitions and which transitions to execute.

We implement this traversal with a simple depth-first search (DFS) of the CFG starting at the entry block. Thus, the algorithm follows a single control path, traversing each block along this path until the end of the function, then backtracks to the last branch point. The DFS portion of the analysis is straightforward; the important feature of the analysis is the use of block-level state caching for speed. The algorithm records the extension state in each basic block before traversing that block. At a subsequent traversal of the same block, the traversal is aborted and the analysis backtracks to the last branch point if the extension state is contained within this cache.

We first describe how to execute an extension at a single program point. We then describe caching at the block level. Finally, we outline the pseudocode for the DFS algorithm.

5.1 Applying an extension to a program point

Figure 4 shows a simplified version of the DFS algorithm. We describe the data structures below.

Each variable-specific instance (`var_state`) consists of an integer holding a state value, a tree for the program object to which the state is attached, and an extension-defined data value of arbitrary size. The tree in the `var` field can be any tree in the code (e.g., an l-value, a general expression, a statement).

An extension's state is represented by an `sm_instance` structure, which has three main components: (1) the extension's single global state, `gstate`, (2) a list of all variable-specific instances, `active_vars`, and (3) a pointer to the extension code, `sm_fn`. Modifications to both `gstate` and `active_vars` are private to each path: mutations revert when the extension backtracks.

The extension code performs the following functions: (1) it determines which transitions to execute and (2) it executes these transitions. Together, these two steps specify the transfer functions for the analysis. When a transition does execute, it can have one of the following effects on the `sm_instance` structure: (1) it can alter `gstate`, (2) it can add or remove elements from `active_vars`, (3) it can alter the state and/or data value of a member of `active_vars`, or (4) it can leave the `sm_instance` unchanged.

To make the analysis algorithm efficient, we exploit the fact that if the extension is *deterministic*, applying the extension to the same program point in the same state will always produce the same result. Thus, we only need to apply the extension to each program point once in each state. More precisely, the determinism condition that we require says that given a single state tuple and a program point, if we set the extension's state to that tuple and apply the `sm_fn` function to the program point, it will always produce the same transformations to the `sm_instance` structure. In addition, we require that each state tuple is a logically separate state machine. We revisit the latter condition below.

5.2 Caching

From *xgcc*'s perspective, the state of an extension is viewed as a set of state tuples represented as pairs, `(gstate, v)`, where `gstate` is the extension's global instance and `v` is either a state variable instance from `active_vars` or the distinguished placeholder "<>." The placeholder ensures that when the analysis begins, the extension state

```
// instance of a state variable
struct var_state {
  AST var;   // AST for var
  int s;     // state of var
  ANY data;  // extension-specific data
};
// a summary edge.
struct edge {
    struct point {
        int gstate;         // global state
        var_state v;        // state var instance
    } start, end;
};
struct block {
  block succs[];  // successors; includes backedges
  AST trees[];    // block's trees in execution order
  set edge blk_add;
  set edge blk_transition;
* set edge sfx_add;
* set edge sfx_transition;
};
struct sm_instance {
  int gstate;                 // global state
  set var_state active_vars; // instances
  sm_fn(sm_instance, AST);   // SM function
};
// Build set of all vars not in block summary
set cache_misses(sm, b) {
  s = {};
  foreach v in sm.active_vars
     if((s1, s2) in b.blk_transition
             where s1 = (sm.gstate, v))
         s U= v;
  return sm.active_vars - s;
}
// DFS traversal
void traverse_cfg(sm, backtrace, caller, b) {
  push (backtrace, b);
  sm.active_vars = cache_misses(sm, b);
  // prune path if visited block in current state before
  if(sm.active_vars = {})
*     relax (backtrace);
      return;
  sm' = copy(sm);
  // apply extension function to each AST node in block
  foreach tree t in b.trees {
    sm->sm_fn (sm, t);
*   if (t is function call) {
*      // t is last tree in b; b has exactly one succ
*      follow_call(sm,backtrace,caller,t,b->succs);
*      return;
    }
  }
  // compute add and transition edges
  foreach v in sm.active_vars {
     e  = (sm.gstate, v);
     // if v was active at block entry: create a
     // transition edge.
     if v' in sm'.active_vars where v.tree = v'.tree
        b.blk_transition U= ((sm'.gstate, v'), e);
     // otherwise v was created by b: create an add edge.
     else
        v' = (v.tree, unknown, nil);
        b.blk_add U= ((sm'.gstate, v'), e);
  }
  if is_exit_block(b)
*    relax(backtrace);
  else
     // apply successor blocks to copy of current sm
     foreach s in b->succs
        traverse_cfg(copy(sm),copy(backtrace),caller,s);
}
```

Figure 4: Depth-first CFG traversal. Lines marked with a * are only relevant to the interprocedural case.

contains exactly one state tuple. For example, the initial state of the free checker would be represented by the set $\{(start, <>)\}$, and, after the first free at line 15, would change to $\{(start, <>), (start, \mathtt{v}:\mathtt{p} \mapsto freed)\}$.

As we described in Section 3, an extension's state is represented as a set of state tuples. Each basic block, b, contains a *block summary* that records the union of all extension states that reach that block and also records how the SM corresponding to each tuple is transitioned during the analysis of that block. Basic blocks are *xgcc*'s internal representation of the CFG for a function. The transitions caused by the basic block are visible to *xgcc* through modifications to the current `sm_instance` structure. We divide the potential ways an `sm_instance` can change while traversing a single block into two categories: (1) transitions that change the value of either the global instance or a variable-specific instance and (2) additions that create a new variable-specific instance. The summary for a block, b, represents these effects using two types of directed edges:

1. Transition edges: $(s, v : t \mapsto v_s) \rightarrow (s', v : t \mapsto v'_s)$. The initial state tuple specifies that at the entry to b, the global instance had the value s and there was an instance of state variable v with value v_s attached to the program object t. The final state tuple specifies that, during the analysis of the block, the SM corresponding to the initial state tuple transitioned to the state where the global instance has value s' and the variable-specific instance for t has value v'_s. Each state tuple that reaches a block generates exactly one transition edge, where the transition can be the identity.

   Figure 5 shows the CFG for the example in Figure 2. The first row in each block in the figure shows the block summary. An example transition edge from the block summary in block 7 is: $(start, \mathtt{v}:\mathtt{p} \mapsto freed) \rightarrow (start, \mathtt{v}:\mathtt{p} \mapsto stop)$. This says that the free checker enters block 7 in the global state *start* with an instance for `p` in the *freed* state and `p` is transitioned to the *stop* state (killed) during the analysis of block 7.

2. Add edges: $(s, v : t \mapsto unknown) \rightarrow (s', v : t \mapsto v'_s)$. The add edge says that when the global instance has initial value s, a new instance of v that attaches state v'_s to t is created while traversing the block. The start tuple for an add edge contains the special value $v : t \mapsto unknown$ because the edge only applies when we know nothing about t at the entry to b.

   An example add edge for block 2 in Figure 5 would be: $(start, \mathtt{v}:\mathtt{p} \mapsto unknown) \rightarrow (start, \mathtt{v}:\mathtt{p} \mapsto freed)$. At block 2's entry, the global instance has the value *start*. At its exit, the global instance still has the value *start*, but the variable `p` now has attached state *freed*. The need for the special value in the start tuple is clear if we consider that if we knew that `p` was freed at the entry to block 2, we would report a double-free error instead of transitioning `p` to the *freed* state.

The block summary is the union of all add and transition edges produced by that block. Before applying the extension to a block, the analysis converts the current extension to a set, s, of state tuples. It then removes any tuple, e, from s that is equivalent to the initial state tuple for some transition edge. After this process, if s is empty, the traversal of the current path is aborted. After a block is traversed, the transition edges for each $e \in s$ and the add edges are both added to the summary.

Note that while the intraprocedural algorithm does not use either the add edges or the destination tuple of the transition edges, they are crucial for the interprocedural caching described in the next section.

Our algorithm computes a fixed point that is similar to the meet-over-paths solution in a traditional dataflow analysis [16]. The analysis stops when the block summary (i.e., cache) at each block contains all state tuples that can reach that block along any control path (i.e., the maximal fixed-point solution).

The algorithm in this section adds an additional restriction to *metal* extensions beyond determinism. The transitions that a variable-specific instance attached to program object v undergoes at a program point cannot be affected by the presence, absence, or state of any other instance attached to object v'. This independence condition allows us to combine all state tuples that reach a block into a single set in the block summary because the state tuples represent independent state machines that could, logically, execute separately. Without independence, the number of times that we analyze each program point would grow exponentially with the number of variable-specific instances. With independence, this number scales linearly with the number of these instances. Note that transitions on a variable-specific instance can be coupled to the value of the global instance.

5.3 DFS With Caching Pseudocode

An extension, `sm`, is applied to a procedure, f, by calling the routine `traverse_cfg` in Figure 4 with four arguments: `sm`, which is initialized to the start state, an empty stack, the caller (relevant in the interprocedural case), and the entry block to f's CFG. In the start state, `gstate` is initialized to the first state in the extension text (*start* for the free checker) and the `active_vars` set contains one element in the special <> state so that the extension's state consists of exactly one state tuple. This element persists throughout the analysis, but it is ignored whenever `active_vars` is nonempty. Thus, we omit it from the block summaries in Figure 5 that contain at least one other element.

The routine `traverse_cfg` implements the depth-first search with caching. This routine is mostly a standard recursive DFS except that at the entry to each new basic block, b, it calls the function `cache_misses` to determine if the current extension state is a subset of the block summary as discussed above. `cache_misses` returns an updated `active_vars` set such that the `sm_instance` with the new set will only contain state tuples that were not in the block summary. If all of the tuples in the current `sm_instance` are in the block summary, the DFS backtracks to the last branch point. If not, `traverse_cfg` applies the extension code to every tree in b in execution order and then traverses b's successors. Successors are applied to a copy of the current extension state.

6. INTERPROCEDURAL ANALYSIS

This section describes our context-sensitive, interprocedural analysis. At a high level, it works as follows:

1. The first preprocessing pass compiles each file in isolation, emitting ASTs to a temporary file. These emitted files include all type declarations, variable declarations, and code within the source file and are typically four or five times larger than the text representation.

2. The second analysis pass reads these temporary files, reassembles their ASTs, and constructs the CFG and call graph. Functions with no callers are considered roots. When computing roots, recursive call chains are broken arbitrarily.

3. The system applies each extension to the CFG with a DFS traversal starting at each callgraph root. On each function call, the system retrieves the CFG for the callee and restarts the traversal there. The extension state is *refined* at the call boundary and *restored* at the return. The rules for refine and restore follow C scoping rules unless the extension specifies otherwise.

   By default, if the function's CFG is not available, the system silently continues to the next CFG node.

To make the DFS algorithm efficient, we add a summary cache to each function computed by combining the block summaries. This cache is checked at each function call. Similar to the intraprocedural caching, if a hit occurs, the call is not followed. Unlike the intraprocedural case, however, we cannot simply abort the current path when there is a cache hit at a function boundary. Because there are many callsites for each function, we may not have analyzed the code after the call in the current state. Thus, on a cache hit, we use both add and transition edges to update the `sm_instance` and the traversal resumes after the function call. The function summary memoizes the results of the state transformation defined by each function.

Our algorithm does not require that the extension has a finite state space, or that the state space is even known when the analysis begins. The algorithm that we describe here is inspired by the dynamic programming algorithm in [18], but the algorithm in [18] requires that the state space of the analysis is finite. The resulting practical difference is that our algorithm executes *metal* extensions top-down. Thus, rather than analyzing each function starting from all possible states, we only analyze each function starting in the states that can reach that function along an interprocedurally valid path (i.e., an interprocedural path that respects call and return sites).

6.1 Refine and Restore

State refinement occurs when a function call is encountered and that function call is followed. The state is restored when the analysis returns from the callee and resumes analyzing the caller. The extension's global instance passes across the function call boundary unchanged.

When the call is followed, any object that passes from the caller's scope to the callee's scope should retain its state. This operation often requires moving the state from an object in the caller's scope to the corresponding object in the callee's scope. When the call returns, the restore operation may need to move the state back from an object in the callee's scope to the appropriate object in the caller's scope and, potentially, restore the original state in the caller. In addition, any variable-specific instances that left scope when the call was followed should reappear when the call returns.

We refine and restore the extension state at a function call according to the list of rules in Table 2. Each rule lists the actual parameter, the formal parameter, the object whose state needs to be transferred, and how this state is

Actual	Formal	State in	Refine rule	Restore rule
x_a	x_f	x_a	state (x_f) = state (x_a)	state (x_a) = state (x_f) (by reference) or state (x_a) unchanged (by value)
$\&x_a$	x_f	x_a	state $(*x_f)$ = state (x_a)	state (x_a) = state $(*x_f)$
x_a	x_f	$x_a.field$	state $(x_f.field)$ = state $(x_a.field)$	state $(x_a.field)$ = state $(x_f.field)$ (reference) or state $(x_a.field)$ unchanged (value)
x_a	x_f	x_a->$field$	state $(x_f$->$field)$ = state $(x_a$->$field)$	state $(x_a$->$field)$ = state $(x_f$->$field)$
x_a	x_f	$*x_a$	state $(*x_f)$ = state $(*x_a)$	state $(*x_a)$ = state $(*x_f)$

Table 2: Refine and restore semantics for retargeting the analysis across a function call. The final four rules actually apply at all levels of indirection (e.g., `p` is the argument, `**p` has state). Note that the extension writer may specify whether or not the actual parameter should be treated as pass by value or pass by reference.

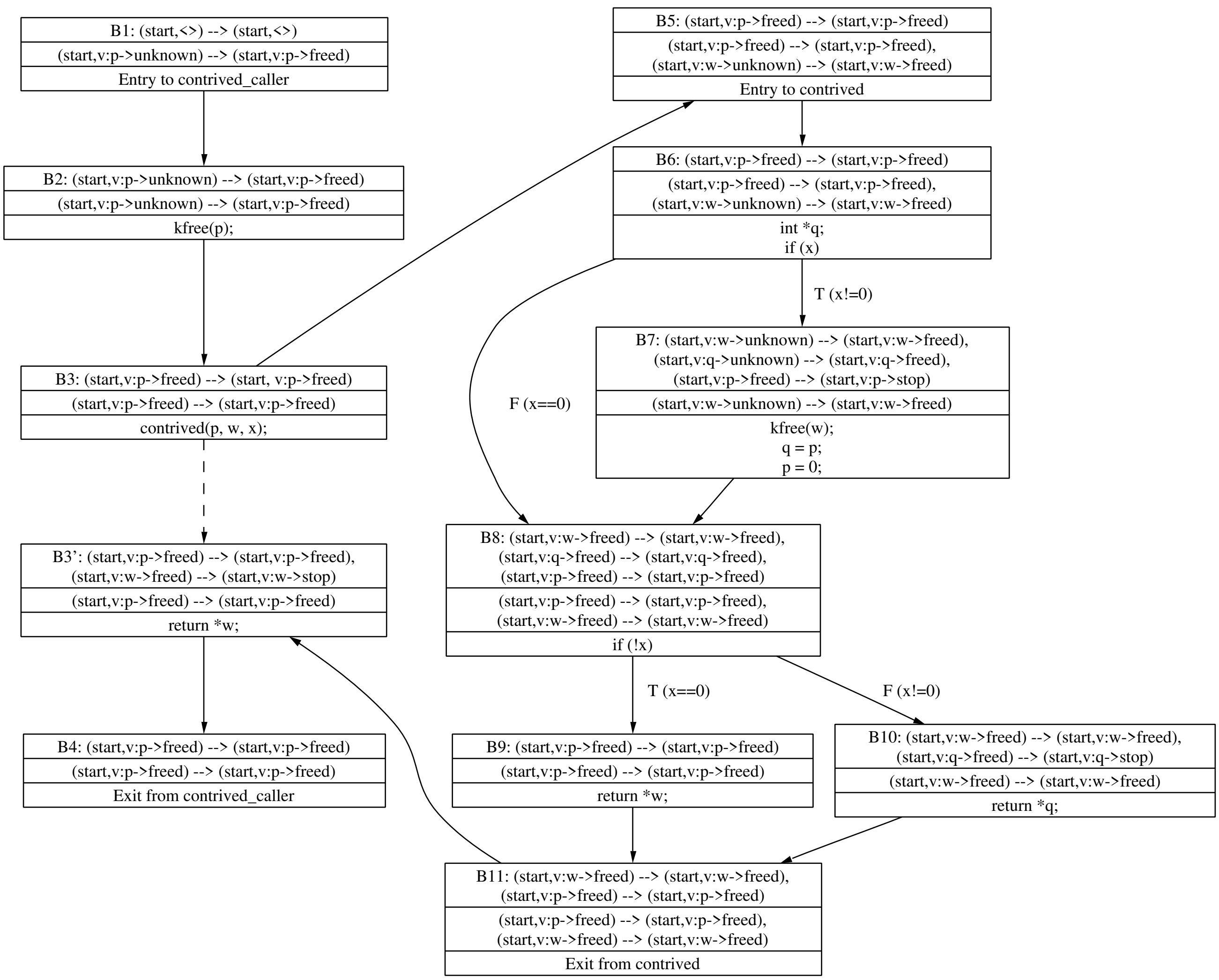

Figure 5: Supergraph for the example code shown in Figure 2. The top field in each basic block shows the block summary, the middle field shows the suffix summary, and the bottom field shows the source code in the block. Each block's number is listed in the first field. Note that none of the suffix summaries record any information about `q` because `q` is a local variable so the analysis would never use these edges. Edges that start and end in a tuple containing the placeholder <> are omitted from the cache unless this tuple is the only element in the cache. Also, the suffix summary intentionally omits edges that end in a tuple with the value *stop*. Suffix edges are only relaxed along traversed paths, i.e. those not suppressed by the algorithm described in Section 8. The analysis does not follow calls to `kfree` because the extension matches these calls. Thus, they are not considered callsites in the supergraph construction.

refined to the callee and then restored to the caller. The rules in the table only cover the case where the state passes through a function argument.

Global variables with attached state are not affected by the refine and restore operations. File-scope variables will leave scope if the call is to a different file. One important nuance with file-scope variables is that they may reenter scope before the callee returns if the analysis reaches a function further down the call chain that is in the same file as the original caller. For this reason, file-scope variables are passed across the function boundary but they are temporarily inactivated (and, thus, ignored by the analysis) until the analysis returns to the file in which they were declared. All state attached to variables and expressions that are local to the caller is saved at the call boundary, deleted from the `sm_instance` before the call is followed, then restored to the `sm_instance` when the call returns.

6.2 Dynamic programming summaries

This subsection describes how we use block summaries to build additional summaries at the function level and at the suffix level. A function summary stores add and transition edges that summarize how an entire function updates the extension state. Function summaries are used to reproduce the effects of analyzing a function when a cache hit occurs at a function call boundary. Each block, b, also has a *suffix summary* that consists of add and transition edges starting at b and ending at the exit point, e_p, to the enclosing function, p. A function summary can be viewed as a suffix summary beginning at the entry block s_p. e_p's suffix summary equals its block summary.

The second row for each block in Figure 5 shows the suffix summary for that block. For example, the summary in block 10 says that if the analysis reaches that block in the state $(start, \texttt{v}:\texttt{w} \mapsto freed)$, then the analysis will also reach the exit block in the state $(start, \texttt{v}:\texttt{w} \mapsto freed)$. Thus, the transition edge

$(start, \texttt{v}:\texttt{w} \mapsto freed) \rightarrow (start, \texttt{v}:\texttt{w} \mapsto freed)$

is part of block 10's suffix summary. Notice that none of the edges in the suffix summaries end in a tuple containing the *stop* state. These edges are unnecessary to the analysis.

Suffix summaries are necessary because distinct SMs can transition to the same state. Thus, an extension can begin analyzing a function call in a new state so that there is no cache hit at the function boundary, but still have a cache hit within the called function. A common example occurs when some source variable v is killed at a program point p that it reaches in two different states. To accurately reflect the effects of the function call to the caller, the analysis must recreate the effects of fully analyzing the called function. Suffix summaries provide exactly this information.

The `relax` function, which computes the suffix summaries, is called whenever the analysis hits the end of an intraprocedural path or the analysis aborts a path because of a cache hit. Figure 6 gives a sketch of the edge computation algorithm in the `relax` function. The code walks backwards through the list of blocks on the current path, stored in the backtrace, combining the edges in each block summary with the suffix edges of the subsequent block in the backtrace. Each block stores the set of suffix edges in the fields `sfx_add` and `sfx_transition`. Initially, all block and suffix summaries are empty.

More specifically, the code first checks if the current

```
// Propagate addition and transition edges up path.
relax(backtrace) {
 b = pop(backtrace);
 // Initialize suffix edges.
 if(is_exit_block(b)) {
    b.sfx_add        U= b.blk_add;
    b.sfx_transition U= b.blk_transition;
 }
 foreach prev in backtrace {
   // All add edges propagate backwards
   foreach e in b.sfx_add
     // Relabel gstate component of add state tuple
     foreach s in prev.blk_transition where
                 s.end.gstate = e.start.gstate {
       e' = e;
       e'.start.gstate = s.start.gstate;
       prev.sfx_add U= e';
     }
   // Transition edges can descend from both edge types
   foreach e in b.sfx_transition {
     foreach s in prev.blk_transition where
                 s.end = e.start
       prev.sfx_transition U= (s.start, e.end);
     foreach s in prev.blk_add where s.end = e.start
       prev.sfx_add U= (s.start, e.end);
   }
   b = prev;
 }
}
```

Figure 6: Pseudocode for the summary computation

block, b, is an exit block. If so, it adds b's block summary to its suffix summary. It then propagates both add and transition edges in b's suffix summary backwards to the previous block's (`prev`'s) suffix summary. This backwards propagation uses the block summary to extend the length of all of the suffix edges in b by one block. It does so by creating new edges from the start point of a block summary edge and the endpoint of a suffix summary edge and adding these extended edges to `prev`'s suffix summary.

For a suffix add edge, e_a, in b, the algorithm looks for an edge in `prev`'s block summary whose end point matches the start of e_a. Recall that if e_a adds an instance attached to the program object p, the start tuple of e_a will contain the special value $\texttt{v}:\texttt{p} \mapsto unknown$. Each block summary records how that block updates the global instance with an edge whose endpoints are state tuples that only include the global instance and the placeholder <>. For the purposes of relaxation, these special transition edges will match the initial state of an add edge if the values of the global instance match. For a suffix transition edge, e_t, the algorithm looks for an add edge or transition edge in `prev`'s block summary whose end tuple is equivalent to e_t's start tuple. The algorithm stops when it either finishes walking over the backtrace or when no new edges are propagated (i.e., the previous block's summary does not grow).

The input to our algorithm is the supergraph for the source base, which is defined in [18]. The supergraph is constructed from the CFG for every function in the source base with the following modifications. First, the algorithm adds two nodes to each routine p: an entry node, s_p, and an exit node, e_p. Second, it splits calls to p into two nodes: a callsite node, c_p, and a return-site node, r_p. Finally, it adds two directed edges: one from c_p to s_p, the other from e_p back to r_p. The supergraph ensures that the only intraprocedural successor of c_p is r_p.

6.3 The Top-Down Algorithm in Detail

The top-down algorithm traverses the supergraph depth-first starting at all function roots. As shown in Figure 4, when a function call is encountered, `follow_call` is called to restart the traversal at the entry to the callee. The routine takes the `sm_instance`, the caller's backtrace, the caller's AST, the callee's AST, and the return-site node, and performs the following operations:

1. Refines the extension state to the callee's scope as described in Section 6.1.
2. Calls `traverse_cfg` with the refined `sm_instance`, an empty backtrace, the callee's AST, and the callee's entry block.
3. Uses the callee's function summary to compute a set, s, of transition and add edges that apply to the current extension state.
4. Restores the edges in s to the `caller`'s context.
5. Creates new `sm_instance` structures for each disjoint exit state. The `sm_instance` can only assign one state value to each instance (in both `gstate` and `active_vars`), and `active_vars` can only contain one instance attached to a particular program object. Thus, s is partitioned into disjoint sets, each of which contains edges whose global instance has the same value and whose variable-specific instances are all attached to different program objects. These partitions are used to construct the new `sm_instance` structures.
6. Uses the new `sm_instance` structures to analyze the remainder of the caller by calling `traverse_cfg` on each new `sm_instance`, the backtrace saved at the callsite, the caller's AST, and the return block at the callsite.

When a state variable instance is transitioned to the sink state, *stop*, in the callee, the instance should be deleted from the extension state when the analysis returns to the caller. Any edges that end with a tuple containing an instance in the *stop* state are omitted from the function summary. Thus, steps 4-6 in the list above will not add the stopped variable to the outgoing extension state.

The analysis will terminate if each SM within an extension reaches a final state after a finite number of transitions from every state. The complexity of this algorithm is similar to that in [18]. Note that our algorithm has an implementation disadvantage over the algorithms in [5, 18] because we may analyze any given function at several different points in the analysis as we reach a call to that function in different states. Thus, we cannot free the storage associated with a function until we are sure that it will not be analyzed again. For large programs, it may be necessary to create compact path summaries that only retain those portions of the AST that are relevant to the analysis. This has not, however, prevented our analysis from running effectively on the Linux kernel. We leave this computation to future work.

7. UNSOUNDNESS

The strength of our extensions is that they can express many rules in a concise way; they are not designed to express sound analyses. It is easy for extensions to make approximations or use analyses that are not conservative. Because the extensions are not intended to be sound, building a sound analysis engine is a misdirected effort; the analysis should instead focus on executing the extensions effectively.

Metal extensions often introduce unsoundness by making approximations or by using analysis techniques that produce good results but are not necessarily correct. For example, using statistical analysis to infer which routines must be paired (such as `lock` and `unlock`) is an effective technique, but cannot guarantee that these inferences are correct.

The interprocedural analysis algorithm in *xgcc* is unsound because it does not analyze recursive loops conservatively, and it does not analyze value flow conservatively. When a function cache hit occurs during a recursive loop, the function summary may not be complete. The conservative solution is to assume that the extension could be in any possible state after the cache hit. Instead, our algorithm assumes that the existing function summary is sufficient.

Our approach is vulnerable to both false negatives and false positives. False negatives occur when a checker fails to warn about an error in the program. For certain classes of errors, such as security holes, false negatives may be a serious problem. However, even here, the tradeoff between soundness and unsoundness at a practical level is not clear-cut. Our focus on expressiveness means that we can easily check many security properties. As a result, to the best of our knowledge, we are able to find more security holes than sound analyses [1, 9, 10].

False positives present a different problem: if a checker's warnings are often wrong, then a user will ignore all of its warnings. The next two sections discuss how we counter false positives with a variety of lightweight suppression techniques and a post-processing ranking step that tries to order the rule violations that we report such that the most important, most likely violations appear first.

In an ideal world, we could write effective, sound analyses to check every program rule that we could think of. Unfortunately, it is well known that it is infeasible to prove programs correct, so it is unlikely that we will ever approach this goal. Thus, the ideal approach is one that is sound when it can be and unsound where the sound approach fails. Our approach explores the benefits and uses of unsoundness.

Program rules fall into equivalence classes where a violation of one rule is no less or more important than a violation of another. Common classes include the set of all exploitable security holes or nondeterministic bugs. In such cases, finding 1000 bugs of a given class is more important than all 10 violations of a single rule in that class. It is the observable behavior of the program that actually matters, not its behavior with respect to any of these properties. The observable behavior will not be correct until all of the bugs in the system are fixed. We are essentially making an end-to-end argument [20]: it makes little sense to expend significant resources reducing the error rate of one part of a system below the residual error rate of the other parts. An unsound analysis that finds more bugs improves the end-to-end behavior of the system more than a sound analysis that finds fewer bugs.

8. FALSE POSITIVE SUPPRESSION

Static analyses can make approximations that lead to incorrect error reports (false positives). This section describes our main techniques for false positive suppression.

Killing variables and expressions. Whenever a variable is defined, *xgcc* iterates through the list of program objects with attached state and determines if the defined variable is used within any of these objects. If so, the object is transitioned to the *stop* state, thereby deleting the corresponding state variable instance. In Figure 2, *xgcc* automatically transitions the variable `p` from the *freed* state to the *stop* state at the assignment, "`p = 0`," at line 8. The assignment case is obvious; the slightly more subtle case is that an expression (e.g., `a[i]`) with attached state is transitioned to the *stop* state when a component of that expression (e.g., `i`) is redefined. This analysis runs transparently unless a checker requests otherwise, and it is the single most important technique for suppressing false positives in checkers that attach state to specific program objects.

Synonyms. If a variable tracked by an extension is assigned to another variable, both variables become synonyms: state changes in one are mirrored in the other. For example, since `p` and `q` are equal in the following code fragment, a successful check that `p` is not `null` also implies that `q` is not `null` at the dereference:

```
p = q = kmalloc(...);
if(!p)
   return 0;
*q;  /* safe dereference: q = p = not null */
```

We implemented synonyms with a 50 line addition to our system. In addition to reducing false positives, synonyms also increase coverage by increasing the number of variables with an attached state. In Figure 2, the assignment on line 7 allows the analysis to catch the error on line 12.

False path pruning.[1] Nonexecutable "false paths" caused by data dependencies are another source of false-positives. *xgcc*'s simple path-sensitive analysis uses basic value tracking combined with a congruence closure algorithm to prune infeasible paths. In Figure 2, because the conditions on lines 4 and 10 are contradictory, there are only two executable paths through the function `contrived`, not four. *xgcc*'s algorithm will prune the two infeasible paths. The algorithm executes the following steps:

1. We track all variable assignments and comparisons, either to constants (e.g., `x = 10`, `x < 100`) or to other variables (e.g., `y = x`, `x < y`). For each assignment to a variable, we assign a new name to that variable so that different definitions of the variable are not confused. If we see the statement `(x < y)`, we record that `x < y` holds along the true branch and `x >= y` holds along the false branch.

2. When we see an expression (e.g., `y = x + 1`), we try to evaluate the expression based on what we already know. If we know that `x` is 10, then we will assign `y` the value 11. If we know nothing about `x`, we store the entire expression.

3. If we see a loop, we set the value of all variables defined in the loop to "unknown" after the loop body. This step eliminates the need to unroll loops.

4. We infer which variables must have the same value through the `=`, `==`, and `!=` operators and place them into a single equivalence class. Using a congruence closure algorithm [8], we then derive as many equalities and non-equalities as possible from the list of tracked assignments. If an equivalence class contains a constant, we know the exact value of everything in that equivalence class. If not, using the tracked inequalities we can derive relationships between equivalence classes. For example, if `x < y` holds, then everything in `x`'s equivalence class is smaller than everything in `y`'s equivalence class.

5. When the extension reaches a branch in the CFG, we first check if the branch condition is a comparison between an expression and a constant and we know the value of the expression. If so, we evaluate the condition and prune the false path. If not, we look through the list of relations between congruence classes. If there is a relationship that either contradicts or confirms the branch condition, we prune the true or false path. Otherwise, we assume both paths are possible.

6. If a path is pruned, we remove all block summary entries that were inserted while analyzing the pruned path so that the summaries at each block do not contain any non-reachable state tuples.

Our algorithm is scalable because it does not track values or evaluate branches too precisely. The justification for this choice is that most paths are executable and most data dependencies are simple. Complex data dependencies are difficult for programmers to understand, so they avoid them as bad practice.

Targeted suppression of false positives. One common cause of false positives is a conflict between an idiomatic code sequence and an analysis approximation. *Metal* makes it easy for an extension to suppress these system-specific idioms. In some cases, this conflict is an indication that the approximation is too coarse and a more thorough analysis is appropriate; in other cases, it is best to suppress the problematic sequence directly.

A conservative version of the free checker that flags all uses of freed variables as errors is a good example. The false positives for this checker came from two sources: (1) passing a freed pointer to a debugging function that prints the pointer, and (2) in BSD, passing the addresses of freed variables to functions that redefine them. We added eight lines of code to the checker to suppress both classes of false positives.

History. Initially, we worried that after the errors we reported were fixed, we would only detect false positives in newer versions that would require heavyweight techniques to eliminate. A simple alternative is to just remember false positives from past versions and suppress them in future versions. We match error reports across versions by comparing file name, function name, variable names involved in the analysis, and the actual error itself as stated by the checker. These fields are relatively invariant under edits (unlike, for example, line numbers) and seem to work well in practice.

9. RANKING

Given ten errors, you can inspect all of them. Given 1000 errors, you cannot. An effective bug-finding approach will report 100s or 1000s of errors in a real system. The ideal

[1]Note that the algorithm described here was implemented in a previous version of *xgcc*. We have not yet ported it to the current version with interprocedural analysis.

error ranking will rank all true error reports before false error reports, and it will order the true error reports according to the severity of each bug. We try to approximate the ideal ranking by first stratifying errors based on their severity, then sorting within each class based on both the probability of the error being a false positive and the difficulty of inspection. The user can then start with the most important class, inspect within that class until the false positive rate is too high or inspection requires too much effort, and skip to the next class of errors.[2]

From our experience with Linux and BSD, implementers almost always fix errors that are difficult to diagnose with testing first. These include use-after-free errors, missing lock releases, and security holes. We rank these errors over those that are easier to diagnose with testing, such as memory allocation failures.

We also group all errors that are computed from a common analysis fact into the same class. For example, all use-after-free errors that involve the same freeing function are placed in the same class. Such grouping makes it easy to suppress them all if the analysis is wrong.

Generic ranking. By default, our system sorts error messages using the following criteria:

1. Distance. Errors that span hundreds of lines are more difficult to diagnose than those that span a few. We rank based on the distance between the statement that contains the error and the statement where the extension started checking the property that led to the error.

2. Number of conditionals. The more conditionals an error spans, the harder it is to diagnose and the more likely it is to be a false path. Each conditional is arbitrarily weighted as ten lines of distance.

3. Degree of indirection. We rank errors that use synonyms below those that do not; the former are more difficult to inspect. We then sort synonyms based on the length of the assignment chain.

4. Local versus interprocedural. Local errors can take seconds to diagnose, whereas interprocedural errors can take minutes. We rank all local errors over global ones and then order global errors based on the length of the shortest call chain that causes the error.

The latter two criteria partition error messages into different classes, which are then sorted using the first two criteria.

In dealing with Linux and OpenBSD implementers, we have observed a curious phenomenon: given errors of equal importance, the more analysis required to find an error, the lower the error should be ranked. As the number of analysis steps increases, the likelihood that an analysis approximation made a mistake and the manual inspection effort both increase. Thus, these error reports are more likely to be false positives and more difficult to diagnose.

Checker-specific and system-specific ranking. The domain knowledge that allows an extension to check a rule also helps it to rank errors more effectively by gathering checker-specific or system-specific information. We mostly use checker-specific ranking to (1) rank errors by severity and (2) perform targeted demotion of errors.

[2]From informal discussions with the PREfix implementers, this strategy and many of the ranking rules in this section have similarities to those that they use.

Many extensions are composed with a simple extension that annotates paths that can be triggered by the user (using the string `SECURITY`) and paths that are likely to be error paths (using the string `ERROR`). Errors on the first type of path pose security risks, since they can be triggered by the user. Errors on the second are empirically more likely to be real errors, in part because error paths are less tested. The extension can also add these two annotations and the additional annotation `MINOR` manually. Errors annotated with `SECURITY` are ranked highest, those annotated with `ERROR` are ranked next, and those annotated with `MINOR` are ranked last.

Statistical ranking. Our most novel ranking method uses statistical analysis. We have observed that an analysis mistake often leads to a local explosion of error reports. The most reliable rules are followed many times and violated rarely. We can use statistical analysis to sort errors based on these numbers.

An earlier version of the free checker used a flow-insensitive, interprocedural analysis to compute a list of all functions that freed their arguments or passed an argument to a function that did. It would then run a local pass that used this list to find errors. The checker had an enormous number of false positives, most due to a single limitation of our analysis: a small number of functions only freed one argument based on the value of another argument, but our analysis decided that these functions always freed their argument. Thus, rather than having an error rate of one error per few hundred callsites, these functions had rates closer to fifty errors per hundred callsites. When we sorted errors based on these rates, all of the real errors went to the top and the errors caused by functions the analysis could not handle were pushed to the bottom.

We rank errors based on the reliability of the rules that caused them using the z-statistic for proportions. The z-statistic evaluates the hypothesis that an outcome that occurs e times out of n is consistent with an expected probability, p_0, for that outcome. We compute the z-statistic as

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

Our null hypothesis is that a rule is obeyed or violated at random. In this case, we expect half of all checks to be successful and half of all checks to fail, hence $p_0 = 0.5$. If a rule is obeyed at random, that rule is probably incorrect. Conversely, if a rule is almost always followed, that rule is probably correct.

We count the number of times the rule was followed (or examples) as e and the number of rule violations (or counterexamples) as c. The total number of events, n, is the sum of e and c.

The larger the computed value of the z-statistic, the higher the significance level at which we can reject the null hypothesis. High values indicate a higher probability that the counterexamples found are indeed violations of a valid rule, and are, therefore, most likely errors.

For the free checker above, each freeing function defines its own rule. That rule is violated when an error is reported on a pointer passed to that function (c). The rule is followed when a pointer passed to that function is never touched again (e).

Ranking code. If a particular block of code causes an explosion of errors, the analysis probably cannot handle some aspect of that code. We first applied this observation to an intraprocedural lock checker that flagged when calls to a locking function did not have a matching unlock. The major source of false positives for this extension was wrapper functions that either always acquired or always released locks. In this case, the locking rule is context-dependent; in some contexts the rule is correct, in some contexts it is not.

When each function is analyzed, we set e to the number of times the function correctly acquired and released locks and c to the number of mismatched pairs. The highest ranked functions had a large number of successful acquire/release pairs with only a few errors. These functions are exactly the ones that most likely contain errors.

Interprocedural analysis would solve this particular problem, but all analysis has limits. For example, if we extend the locking analysis to include the Linux semaphore routines `up` and `down`, there will be a high rate of false positives since semaphores are sometimes used as counters, which need not be paired, and sometimes as locks, which must be paired. Ranking easily distinguishes these two different uses, whereas adding interprocedural analysis will not.

Discussion. Statistical ranking can also be used to infer the severity or likelihood of real errors. The most serious or most likely errors tend to violate rules that are almost always followed. Thus, ranking is useful even for an approach that does not report any false positives. Sound approaches should find ranking especially useful because conservative assumptions often lead to large numbers of false positives. In our experience, false positives are not randomly distributed but often come from a small set of analysis mistakes that are automatically identified with ranking.

Ranking can be a simple technique, but, from the error-inspector's point of view, it makes an exhilarating difference.

10. RELATED WORK

In this section, we discuss other systems for finding bugs in C programs. We divide these systems into those that require programmer annotations and those that do not. Most of the systems discussed here are sound whereas our system is not. We focus on checking a broad class of properties that are either difficult or impossible to specify soundly. Because *metal* is flexible, we believe that our system can check a wider variety of properties with a wider variance in precision than any other systems with similar goals. The discussion below focuses on other differences between our system and other static bug-finding tools.

10.1 Bug-Finding Without Annotations

ESP [5] is the project most similar in spirit to our own. Properties are specified in ESP using a state machine language similar to *metal*. These properties are then verified using a sound, interprocedural dataflow analysis based on the RHS algorithm [18]. ESP includes the "abstract simulation" algorithm, which is an interprocedural false-path pruning algorithm. Our false-path pruning algorithm uses a congruence closure algorithm which, in the intraprocedural case, is more powerful than the algorithm actually used in ESP. The ESP approach is more likely to scale in the interprocedural case than ours.

The SLAM project [2] aims to verify temporal safety properties by using a combination of predicate abstraction [15], model checking [4], and predicate discovery. Our approach and the SLAM approach have different goals: SLAM is a verification tool intended for small, bug-prone pieces of larger systems. It is effective within these scalability limits. Our approach is intended for large systems.

Intrinsa's PREfix [3] is an industrial-strength tool for C that performs symbolic evaluation of interprocedural execution paths while looking for errors such as uses of uninitialized memory, buffer overflows, NULL-pointer dereferences, and memory leaks. PREfix works on large software systems. It does a deeper, more expensive analysis than our system by building a memory model along each execution path in the program. However, it only finds a fixed set of error types using a fixed set of analyses. We allow programmers to extend both.

10.2 Bug-Finding With Annotations

There are many annotation-based checking projects. One of the most developed is Extended Static Checking [7] (ESC) and ESC/Java [19], which are annotation-based tools that use a theorem prover to find errors. The annotations used by ESC allow for varying levels of detail, which lets the annotator balance annotation effort, completeness, and verification time. For basic programming errors, the reported annotation overhead for ESC ran as high as one annotation per three lines of code for small programs. Recent work on inferring these annotations attempts to reduce this burden [12]. Because of this effort, they run on small code bases, and find relatively few bugs compared to our approach.

LCLint [11] statically and unsoundly checks C programs with the aid of programmer annotations. LCLint requires additional annotations to improve precision, which leads to a significant annotation burden to produce useful results.

Cqual [14] is an annotation-based approach that adds flow-sensitive qualifiers to standard C types. The analysis is interprocedural and sound. However, it requires annotations both to express program properties and to suppress false positives caused by conservative aliasing assumptions. These annotations are a significant practical drawback.

In general, bug-finding techniques that rely on annotations require strenuous, invasive code modifications. This annotation overhead can be prohibitive for large systems. One of the most rigorous measurements of this overhead comes from Flanagan and Freund who measured an annotation overhead of one annotation per 50 lines of code at a cost of one programmer hour per thousand lines of code [13]. For a system the size of Linux (2MLOC), this would require two spells of 40 days and 40 nights of continuous annotating for a single property! In contrast, once the fixed cost of writing a *metal* extension is paid (often a day or so) there is little incremental cost to applying it to a large amount of code.

10.3 Language-based approaches

We view language-based approaches to preventing bugs as largely complementary to our work. The Vault [6] language lets users specify typestate properties within the language; the role analysis concept proposed in [17] can specify even more complex properties by providing a language mechanism for specifying legal aliasing relationships. Programs written correctly using these languages would be protected from some of the bugs that we find.

Tool-based analysis, however, does have some significant practical advantages. First, our statistical extensions can automatically infer some of the temporal properties that a languages like Vault requires programmers to manually specify [10]. Tools can also transparently check properties without requiring the use of a specific language for code construction or rewrites. Language adoption has historically been an erratic process. Tools work immediately.

11. CONCLUSION

This paper describes a language for specifying program properties, *metal*, and an analysis engine for checking these properties statically, *xgcc*, that has found thousands of bugs in real source code. The approach we present centers around a single goal: to find as many bugs in real systems as possible. *Metal* and *xgcc* are designed to support this goal. One implication of our work is that finding bugs is easy given the right approach. We present one possible approach that centers around extensibility. An extensible specification language can express a broad class of properties within a single framework. Expressiveness in this language must be matched with an efficient algorithm that does not impose too many restrictions on the analyses it executes. Thus, extensibility is a system-wide property. We believe that *metal* and *xgcc* are a reasonable step towards building such a system.

12. ACKNOWLEDGMENTS

Andy Chou built numerous pieces of the system that we describe. Wilson Hsieh, David Chen, and David Gupta provided helpful comments. We especially thank Godmar Back, Manu Sridharan, David Heine, and Robert Hallem for many close reads, and John Mitchell for succinct, effective structuring comments. This work was supported by NSF award 0086160 and by DARPA contract MDA904-98-C-A933.

13. REFERENCES

[1] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[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] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] 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.

[6] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.

[7] D.L. Detlefs. An overview of the extended static checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996.

[8] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[9] 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.

[10] 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.

[11] D. Evans, J. Guttag, J. Horning, and Y.M. Tan. Lclint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[12] C. Flanagan, K. Rustan, and M. Leino. Houdini, an annotation assistant for esc/java. In *Symposium of Formal Methods Europe*, pages 500–517, March 2001.

[13] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[14] 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.

[15] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, 1997.

[16] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[17] Victor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Conference Record of the Twenty-Ninth ACM Symposium of Principles of Programming Languages*, January 2002.

[18] 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.

[19] K. Rustan, M. Leino, G. Nelson, and J.B. Saxe. Esc/Java user's manual. Technical note 2000-002, Compaq Systems Research Center, October 2001.

[20] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.