

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

Hallem, Chelf, Xie and Engler (Stanford University)  
Summary by Abheek Anand

## Motivation

There have been several recent techniques for detecting program bugs statically. *PREfix* performs symbolic evaluation of interprocedural execution paths, while looking for errors such as uninitialized memory, buffer overflows and memory leaks. While it is very comprehensive, it allows only for fixed types of analyses, and thus finds a fixed subset of the bugs. *ESC/Java* uses programmer-written annotations to run theorem-provers to verify program correctness. Recent work includes automatically inferring these annotations (Houdini) at the expense of exponentially higher performance overhead. However, techniques based on annotations require strenuous, invasive code modifications, which most programmers are unwilling to do. Moreover, the amount of annotation is proportional to the size of the codebase being tested, making these analyses limited in their scalability.

This paper describes a new technique for finding bugs in systems code. *Metal* is a language used for specifying program properties, and an analysis engine ‘*xgcc*’ is then used to test these properties against a given codebase.

## Metal

*Metal* is designed to be easy to use, and flexible to express a variety of properties. It provides a state-machine (SM) as a fundamental abstraction to model these properties. These SM’s (1) recognize source-code relevant to a given rule by using pattern matching, and (2) check that these actions satisfy the given constraints. For example, the following piece of metal code describes an SM to check for correct uses of the ‘kfree’ function (checking for double freeing and using a pointer after it has been freed).

```
state decl any_pointer v;  
start: { kfree(v) } → v.freed;  
  
v.freed: { *v } → v.stop,  
  { err(“using %s after free”, mc_identifier(v)) ;}  
| kfree(v) → v.stop,  
  { err(“double free of %s”, mc_identifier(v)) ;}
```

The SM matches any pointer declared with the variable *v*, and attaches a new instance of the SM with each such variable. The number of instances associated with a pattern grows and shrinks as the program executes, since the SM keeps track of only those instances that are currently in use. Each SM is associated with a global state (start), and local states (v.freed, v.stop) associated with each instance of the SM. Extensions in *Metal* can be extended using arbitrary C-code to do more complex operations. The expression ‘*v*’ in the above example is called a hole variable in the paper. Hole variables are typed expressions which are matched against the appropriate pattern in the actual source code.

## XGCC

Recall that *xgcc* is the engine used to run these SM’s over the actual source code. It applies extensions to the source code AST in a top-down manner, and in execution order. Each execution path is explored once, and interprocedural calls carry state from the caller to the callee and then back. Several optimization techniques are used to make this scalable.

1. Assuming that each extension is deterministic, the state at each block is cached for each incoming state. Two forms of caching are used:
  - Intraprocedural caching: Each block keeps a cache of outgoing state for each incoming state (called a block summary), and a cache hit means we do not need to further analyse the block.
  - Interprocedural caching: Each function call maintains a similar suffix summary, and on a cache hit the function is not explored any further, and the final state is picked off the cache. However, it

is still necessary to continue along the execution path since different functions might have multiple call-sites.

2. Remembering State: The analysis engine remembers decisions it makes while taking branches. For example, each branch at a statement like `if (x==0)` remembers the value of `x` (zero/non-zero) along the path. This is useful for later suppression of infeasible paths, and optimizations related to value-forwarding.
3. The algorithm computes a Meet-Over-All-Paths fixed point for computing the final state, similar to traditional data-flow analysis.

The intra-procedural analysis computes the final state across a single function. The inter-procedural analysis carries state from caller to callee ('Refine'), and then returns the state back to the caller at the function return ('Restore'). They define rules to do these operations correctly given the semantics of the procedure call. Similar rules are defined for other types of scopes, like file-scoped variables.

The overall analysis algorithm uses a dynamic-programming approach. The state-space is not assumed to be finite, and thus the algorithm can work successfully in a top-down manner (unlike in other similar systems). This has the obvious advantage that each function is analysed only for the input states that can possibly reach that function, as opposed to a bottom-up approach which would require the analysis to work on all possible states.

### Unsoundness

The system takes an end-to-end argument to finding bugs. Namely, it is more important to find more bugs in the overall system than finding a lot of bugs in a particular section of the code, since the residual error rate over the entire program path is what characterizes the 'goodness' of the analysis. Because of this tradeoff in the analysis, the engine is vulnerable to both false-negatives and false-positives. They describe some of the reasons for this happening, and some further optimizations which they claim reduce the false-positive rate.

Finally, they go on to describe a statistical ranking system, which given a set of errors, ranks them based on a heuristic according to their individual severity.

### Results

The 'Stanford Checker', as this system came to be known as in the systems community when it was first introduced, has successfully discovered thousands of bugs in real systems code like the Linux kernel and OpenBSD. The ranking system gives programmers an effective method for correcting a large set of bugs according to their level of severity. The approach is clearly scalable (it has been tested on the Linux kernel, with 2MLOC).

There have been several other papers which have described further extensions to this work. One of them discusses using statistical methods for actually determining correctness properties automatically. They also describe several other *Metal* extensions to detect specific security bugs, and other generic systems errors like deadlocks.

It would have been nice had the tool been available for public use. However the group claims its still in development and haven't released it yet. Several extensions could make this more useful. For example, it is not clear to me whether they do any form of pointer analysis at all, which would be useful both in increasing the coverage of the bugs found, and in increasing the efficiency of the engine.