Some lessons from using static analysis and software model checking for bug finding

Madanlal Musuvathi   Dawson Engler *

Computer Systems Laboratory
Stanford University
Stanford CA 94305
E-mail: {madan,engler}@cs.stanford.edu

General Terms

Reliability, Verification, Static Analysis, Model Checking.

Categories and Subject Descriptors

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

1. INTRODUCTION

This paper grew out of our experiences with software model checking after several years of using static analysis to find errors. We initially thought that the trade-off between the two was clear: static analysis was easy but would mainly find shallow bugs, while model checking would require more work but would be strictly better — it would find more errors, the errors would be deeper, and the approach would be more powerful. These expectations were often wrong.

This paper documents some of the lessons learned over the course of using software model checking for three years and three projects. The first two projects used both static analysis and model checking, while the third used only model checking but sharply re-enforced the trade-offs we had previously observed.

The first project, described in Section 2 and 3, checked FLASH cache coherence protocol implementation code [24]. We first used static analysis to find violations of FLASH-specific rules (e.g., that messages are sent in such a way as to prevent deadlock) [7] and then, in a follow-on work, applied model checking [26]. One interesting feature of the model checking work was to use static analysis to do automatic model extraction, which was more effective than a prior manual verification effort [31]. One startling result (for us) was that despite the depth of model checking, it found many fewer errors than relatively shallow static analysis (8 errors versus 34).

The second project, described in Section 4, checked the AODV network protocol [10]. We eliminated the need to extract a model by building CMC [29], a model checker that directly checks C implementations. We used CMC to check three AODV implementations, and then statically analyzed them. The results indicate a clear success for model checking. The model checker found 42 errors (roughly 1 per 300 lines of code). About half of these errors involve protocol properties that would be difficult to check statically. However, in the class of properties both methods could handle, static analysis found more errors than model checking. Also, static analysis involved considerably less effort: it took just a couple of hours, while our model checking effort took approximately three weeks.

The final project, described in Section 5, used CMC on the Linux TCP network stack implementation. This project was clearly motivated by our desire to test the applicability of model checking on large systems. This project exposed several unexpected scalability issues with model checkers. Initially, we thought that our only major hurdle would be the state explosion problem. Instead, we spent many months and multiple iterations in just getting our first working environment model for the TCP implementation. Even this required us to take an extreme approach of running the *entire* Linux Kernel along with the TCP implementation in CMC. Once we had a working model, another challenge was to decide on the right environment inputs to trigger various protocol behaviors. Section 5 elaborates these problems and our attempted solutions. Our model checking effort is still in progress and CMC has till date found 4 errors in the TCP implementation.

While this paper describes drawbacks of software model checking compared to static analysis, it should not be taken as a jeremiad against the approach. We reside very much in the "model checking camp" and intend to continue pursue research in the area. A main goal of the paper is to recount what surprised us when using model checking for

*This research was supported in part by DARPA contract MDA904-98-C-A933, by GSRC/MARCO Grant No:SA3276JB, and by a grant from the Stanford Networking Research Center. Dawson Engler is partially supported by an NSF Career Award.

SoftMC '03, July 18–19, 2002, Boulder, CO, USA.

reasonably large-scale checking of software implementations. While more seasoned minds might not have made the same misjudgments, our discussions with other researchers have shown that our naivete was not entirely unreasonable.

This paper is a set of case studies, rather than a broad study of static analysis and model checking. While this limits the universality of our conclusions, we believe the general trends we observe will hold, though the actual coefficients observed in practice will differ.

1.1 The model checking approach

All of our case studies use traditional explicit state space model checkers [13, 22]. We do no innovation in terms of the actual model checking engine, and so the challenges we face should roughly mirror those faced by others. We do believe our conclusions optimistically estimate the effort needed to model check code. A major drawback of most current model checking approaches is the need to manually write a specification of the checked system. Both of our approaches dispense with this step. The first automatically extracts a slice of functionality that is translated to the model checking language, similar to the automatic extraction work done by prior work, notably Bandera [9] and Feaver [21]. Our second approach eliminates extraction entirely by model checking the implementation code directly.

1.2 The static analysis approach

At a high level, our static analysis checking is based on compiler extensions ("checkers") that are dynamically linked into the compiler and applied down a control-flow graph representation of source code [14]. Conceptually these extensions examine one path at a time. I.e., they are *flow sensitive*, rather than using a more traditional dataflow framework that would conflate information at program joint points. Extensions can perform either intra- or inter-procedural analysis at the discretion of the checker writer. In practice, the approach has been effective, finding hundreds to thousands of errors in Linux, BSD, and various commercial systems.

While we make claims about "static analysis" in general, this paper focuses on our own static analysis approach, since it is the one we have personal experience. The approach has several idiosyncratic features compared to other static approaches that should be kept in mind.

First, our approach is unsound: code with errors can pass silently through a checker. We generally optimize checkers to find bugs, rather than demonstrating their absence. In particular, when checkers cannot determine a needed fact they typically do not emit a warning. In contrast, a sound approach would conservatively emit an error report whenever it cannot prove the error cannot occur. Unsoundness allows us to aggressively check properties beyond the practical reach of sound tools, which would overwhelm the user with false positives.

Second, we use relatively shallow data flow analysis rather than a deeper simulation based approach such as in PREfix [4]. While we perform a mild amount of path sensitive analysis to prune infeasible paths [20], we do not model the heap, do not track most variable values, and have limited aliasing information. The flip side of this shallowness is that we do not have to build an accurate, working models of the environment or of code we do not have. In contrast, while the PREfix tool can find deeper errors, it requires that models of missing functions be supplied. [1] If we relied more heavily on simulation, then the trade-offs might look different as well. In a sense simulation brings static analysis closer to classic model checking, and hence share some of its weaknesses as well as strengths.

Third, the approach tries as much as possible to avoid the need for annotations, in part by using statistical analysis to infer properties to check [15] (such as which functions must be paired, which functions can return null, etc.). The need for annotations would dramatically increase the effort necessary to use the tool, and to an extent diminish its advantages

There are many papers on the approach, so we elide a thorough description here. The paper [14] gives a reasonable, though dated overview. The paper [20] gives a more up-to-date view of the system. The tutorial in [6] has a series of checker examples and references that the interested reader can use for a more thorough introduction.

2. CASE STUDY: FLASH

This section gives a short summary of using both static analysis and model checking to find bugs in FLASH cache coherence protocol implementation code. The next section focuses on the lessons learned from these efforts. Readers familiar with Chou et al. [7] can skip Section 2.1 and 2.2. Readers familiar with Lie et al. [26] can skip Section 2.1 and 2.3.

2.1 FLASH overview

The Stanford FLASH multiprocessor [24] is a scalable cache-coherent DSM machine that implements its communication protocols in software that runs on an embedded processor in its programmable node controller, MAGIC. While implementing such protocols in software facilitates great flexibility, it places a serious burden on the programmer. The code executes on each cache miss, so it must be egregiously optimized. At the same time a single bug in the controller can deadlock or livelock the entire machine.

We checked five protocols with static analysis and four with model checking. These protocols range between 10K to 18K lines of code and have long control flow paths. The average control flow path ranges from 73 to 183 lines of code, while the maximum is around 400 lines. Intra-procedural paths that span 10-20 conditionals is not uncommon. For our purposes, FLASH protocol implementation is a representative of low-level code that exists on a variety of embedded systems. It is highly optimized, difficult to read, and thus difficult to get correct. For the purpose of finding errors, FLASH was a hard test: by the time we checked it had already undergone over five years of testing under simulation, on a real machine, and via manual formal verification [31].

2.2 Checking FLASH with static analysis

While FLASH code was difficult to reason about, it had the nice property that many of the rules it had to obey mapped clearly to source code and thus were readily checked with static analysis. The following rule is a representative

[1] Our approach has found errors in code checked by PREfix, so the depth of checking is not entirely one-sided.

example. In the FLASH code, incoming message buffers are read using the macro MISCBUS_READ_DB. All reads must be preceded by a call to the macro WAIT_FOR_DB_FULL to synchronize the buffer contents. To increase parallelism, WAIT_FOR_DB_FULL is only called along paths that require access to the buffer contents, and it is called as late as possible along these paths. This rule can be checked statically by traversing all program paths until we either (1) hit a call to WAIT_FOR_DB_FULL (at which point we stop following that path) or (2) hit a call to MISCBUS_READ_DB (at which point we emit an error). A checker for this rule, written in *metal* can be found in Appendix A. In general the static checkers roughly follow a similar pattern: they match on specific source constructs and use a extensible state machine framework to ensure that the matched constructs occur (or do not occur) in specific orders.

Table 1 gives a representative listing of the FLASH rules we checked. Since the primary job of a FLASH node is to receive and respond to requests, most rules involve correct message handling. Most errors were caused by failure to deallocate message buffers (9 errors) and by mis-specifying the length of a message (18 errors). The other rules were not easier, but generally had less locations where they had to be obeyed. There were 33 errors in total and 28 false positives. We obtained these numbers three years ago. Using our current system would have reduced the false positive rate, since most were due to simple infeasible paths, which our current system would eliminate. (However, the severity of the errors made the given rate perfectly acceptable.)

2.3 Model checking FLASH

Our model checking approach used static analysis to reduce the work required by automatically extract models from source code. We started the project after noticing the close correspondence between a hand-written specification of FLASH (from [31]) with the implementation code itself. FLASH code made heavy use of stylized macros and naming conventions. These "latent specifications" [15] made it relatively easy to pick out the code relevant to various important operations (message sends, interactions with the I/O subsystem, etc).

Model checking with our system involves the following four steps. First, the user provides a *metal* extension that when run by our extensible compiler marks specific source constructs, such as all message buffer manipulations or sends. These extensions are essentially abstraction functions. Second, the system then automatically extracts a backward slice of the marked code, as well as its dependencies. Third, the system translates the sliced code to a Murφ model. Fourth, the Murφ model checker checks the generated model along with a hand-written environment model.

Model checking allowed us to validate properties out of the reach of static analysis. Table 2 lists a representative subset. Surprisingly, there were relatively few errors in these properties as compared to the more shallow properties checked with static analysis.

Invariants
The RealPtrs counter does not overflow (RealPtrs maintains the number of sharers)
Only a single master copy of each cache line exists (basic coherence)
A node can never put itself on the sharing list (sharing list is only for remote nodes)
No outstanding requests on cache lines that are already in Exclusive state
Nodes do not send network messages to themselves
Nodes never overflow their network queues
Nodes never overflow their software queues (queue used to suspend handlers)
The protocol never tries to invalidate an exclusive line
Protocol can only put data into the processor's cache in response to a request

Table 2: Description of a representative subset of invariants checked in four FLASH protocols using model checking. Checking these with static analysis would be difficult.

3. LESSONS FROM FLASH

Outside the model checking community, the general perception is that since model checking is "deeper" than static analysis then if you take the time to model check code, you will find more errors. We have not found this to be true. In the FLASH case, static analysis found roughly four times as many bugs as model checking, despite the fact that we spent more time on the model checking effort. Further, this differential was after we aggressively tried to increase bug counts. We were highly motivated to do so since we had already published a paper that found 34 bugs (Chou et al [7]); publishing a follow-on paper for a technique that found fewer was worrisome. In the end, six of the eight bugs found with model checking had been found by static analysis. Only two bugs were new — these were counter overflows that were deeper in the sense that it required a deep execution trace to find them. While they could potentially have been found with static analysis, doing so would have required a special-case checker.

The main underlying reason for the lower bug counts is simple: model checking requires running code, static analysis does not. This fact has important implications. We discuss several below.

First, model checking requires a working model of the environment. Environments are often messy and hard to specify. The formal model will simplify it. There were five main simplifications that caused the model checker to miss FLASH bugs found with static analysis:

1. We did not model cache line data, though we did model the state that cache lines were in, and the actual messages that were sent. This omission both simplified the model and shrank the state space. The main implication in terms of finding errors was that there was nothing in the model to ensure that the data buffers used to send and receive cache lines were allocated, deleted or synchronized correctly. As a result, model checking missed 13 errors: all nine buffer allocation errors and all four buffer race conditions.

2. We did not model the FLASH I/O subsystem, primarily because it was so intricate. This caused the

Rule	Intuition	LOC	Bugs	FP
"WAIT_FOR_DB_FULL must come before MISCBUS_READ_DB"	The synchronizing wait call is needed to ensure the all the data has arrived.	12	4	1
"The has_data parameter for message sends must match the specified message length (be one of LEN_NODATA, LEN_WORD, or LEN_CACHELINE)"	Message lengths and has_data are decoupled to simplify hardware design.	29	18	2
"Message buffers must be: allocated before use, deallocated after, and not used after deallocation."	Identical to memory semantics, except there are limited numbers of buffers.	94	9	25
"Message handlers can only send on pre-specified 'lanes"'	Deadlock prevention.	220	2	0
Total		355	33	28

Table 1: Representative FLASH rules, a simplistic intuitions for each, the number of lines of code for a MC rule checker (LOC), the number of bugs the checker found (Bugs) as well as the number of false positives (FP). We have elided other less useful checkers; in total, they found one more bug at a cost of about 30 false positives.

model checker to miss some of the message-length errors found by the static checker.

3. We did not model uncached reads or writes. The node controllers support reads and writes that explicitly bypass the cache, going directly to memory. These were used by rare paths in the operating system. Because these paths were rare it appears that testing left a relatively larger number of errors on them as compared to more common paths. These errors were found with static analysis but missed by the model checker because of this model simplification.

4. We did not model message "lanes." To prevent deadlock, the real FLASH machine divides the network into a number of virtual networks ("lanes"). Each different message type has an associated lane it should use. For simplicity, our model assumed no such restrictions. As a result, we missed the two deadlock errors found with static analysis.

5. FLASH code has many dual code paths — one used to support simulation, the other used when running on the actual FLASH hardware. Errors in the simulation code were not detected since we only checked code that would actually run on the hardware.

Taking a broader view, the main source of false negatives is not incomplete models, but the need to create a model at all. This must be done for each new system to check and, given finite resources, the cost of doing so can preclude checking new code or limit checking to just code or properties whose environment can be specified with a minimum of fuss. In the case of FLASH, time limitations caused us to skip checking the "sci" protocol, thereby missing five buffer management errors (three serious, two minor).

Second, as with dynamic checking tools, model checking can only find errors on executed code paths. It turns out that in practice it is actually quite difficult to exercise large amounts of code. In the case of the networking code we describe in the next two sections, we execute around 50% of all *statements* (not paths!), despite aggressive attempts to raise this count higher. We do not have a good notion of how many paths we miss in FLASH, but an uncontroversial guess would put it in the range of substantial to enormous.

Unchecked code has the unpleasant feature that it is silent — you receive no warning when code is not exercised with the model checker. While we use path coverage tools (i.e., static analysis) to partially detect this problem, these tend to work at a very coarse-level — whether the code was ever executed, rather than whether it was explored on all possible paths or with all "interesting values." Additionally, detection is diagnostic rather than constructive. Knowing that code is missed turns out to be a large step from knowing how to drive the model checker to hit it. In contrast, static analysis can traverse all program paths, finding errors on any of them. This is a crucial ability.

Static analysis: Push a button, check millions of lines of code. The first order limit on how many bugs you find is the number of properties you check times the number of times each property must be obeyed by code. In practice, this calculation degenerates to how much code you check. In this respect, static analysis has a clear advantage: it can check any code that you can compile. It does not require executing code, thus does not require an accurate model nor clever abstraction tricks. It can do all path coverage automatically. It allows you to check the environment itself, rather than abstracting it away. (Section 4 gives some quantitative measurements of how much this matters.) Operationally, one of the most important effects is that if code calls unresolved functions, you do not need to build a model or even understand what functions do to make progress — simply skip them. As a result, in our experience, it can easily be one to two orders of magnitude (in time, effort, cleverness) to model check code.

A second feature is that since it is so easy to run over all paths in large code bases, static analysis can gather large amounts of statistical information used to automatically infer which properties to check [15]. More checked properties equals more bugs. In contrast, model checking tends to be limited to a much smaller number of dynamic code paths and hence a more limited window for inference.

3.1 Observed model checking advantages

Neither static analysis nor model checking are at the stage where one dominates the other. We believe static analysis will generally win in terms of finding as many bugs as possible. In this sense it is better, since less bugs gets users closer to the desired goal of the absence of bugs ("total correctness"). However, model checking has two strengths that seem difficult to match.

First, since model checking executes the code, it can check properties not easily visible to static inspection, in particular invariants over the data structures and values produced by code. Examples include the invariants that a routing table does not have loops, that a tree is balanced, or (from FLASH) that only a single master copy of each cache line exists. In contrast, static analysis tends to work best at checking properties directly mirrored in the source code itself, such as ensuring that function calls happen in specific orders. In some sense, static analysis checks source code well, but checks the implications of the source code relatively poorly. On the other hand, model checking checks implications relatively better, but because of the problems with abstraction and coverage, can be less effective checking the actual code itself.

Second, model checking checks for actual errors, rather than having to reason about all the different ways the error could be caused. If it catches a particular error type it will do so no matter the cause of the error. For example, a model checker that runs the code directly will detect all null pointer dereferences, deadlocks, or any operation that causes a runtime exception since the code will crash or lock up. Importantly, it will detect them without having to understand and anticipate all the ways that these errors could arise. In contrast, static analysis cannot do such end-to-end-checks, but must instead look for specific ways of causing a given error. Errors caused by actions that the checker does not know about or cannot analyze will not be flagged. In our case, we check many properties that are undecidable, and so minimize false positives by looking for errors only in specific analyzable contexts. For example, most of our analysis can detect errors that occur on a particular code path, but are much weaker at finding errors that require heap analysis. The robustness of model checking (or its depth) would be the main ability we would like to take from it for our static analysis work.

4. CASE STUDY: AODV

This section describes our experiences finding bugs in the AODV routing protocol implementation using both model checking and static analysis. We first describe CMC, the custom model checker we built, give an overview of AODV, and then compare the bugs found (and not found) by both approaches.

4.1 CMC Overview

The approach we used on the FLASH protocol had two problems. First, it required the user to select and automatically mark stand-alone subparts of a system. Doing so required an intimate understanding the checked system, making it difficult to scale the approach to large systems. Second, Murφ, like most modeling languages lacks many C constructs such as pointers, dynamic allocation, and bit operations. These omissions make translation difficult. Our attempts in building a "generic" C to Murφ translator failed. Avoiding the problem by abstracting the C features not supported in Murφ would severely restrict the class of bugs that the model checker can find.

We countered these problems by building CMC, a model checker that checks programs written in C [29]. CMC was motivated by the observation that there is no fundamental reason model checkers must use a weak input language. As it executes the implementation code directly, it removes the need to provide an abstract model, tremendously reducing the effort require to model check a system. As the implementation captures all the behaviors of the system, CMC is no longer restricted to behaviors that can be represented in conventional modeling languages.

CMC is a Murφ-like tool for programs written in C or C++. It explores the state space of a given system explicitly by storing states. The state of a system is captured by its entire context: the global variables, heap, stack and the machine registers. Each process in the model checked system can be emulated by one or more threads. CMC schedules these threads to explore the state space of the system. During model checking, CMC checks for a variety of safety properties represented as boolean functions written in C.

As CMC deals with actual implementations and not their abstract models, it has to deal with much larger states and significantly larger state spaces. However, CMC is designed to be a bug-finding tool rather than a tool to check for absolute correctness. By using approximate reduction techniques, CMC is able to alleviate the state space explosion problem. CMC uses hashcompaction [36] to significantly reduce the state space requirements at the cost of a small risk of missing errors. Also, CMC employs various heuristics [29] to automatically remove unnecessary variables from the state.

4.2 AODV Overview

AODV (Ad-hoc On-demand Distance Vector) protocol [10] is a loop-free routing protocol for ad-hoc networks. It is designed to work in an environment of mobile nodes, withstanding a variety of network behaviors such as node mobility, link failures and packet losses. AODV guarantees that the network is free of routing loops at all instants. However, it is plausible that errors in the protocol specification or its implementation can introduce loops in the network. If any such routing loop appears in the network, the protocol has no mechanisms to detect or recover from them. Thus, the loops persist forever, completely breaking the functioning of the protocol. As a consequence, it is important that both the AODV protocol specification and any AODV implementation be tested for loop freeness as thoroughly as possible.

AODV has a key property that greatly simplifies the environment model. The only input it requires is a user request for a route to a destination. This can be easily modeled as a nondeterministic input that is enabled in all states. Apart from this, an AODV node responds to two events, a timer interrupt and a packet received from other AODV nodes in the network. Both are straightforward to model.

4.3 Model Checking AODV with CMC

CMC checked three publicly available implementations of AODV: *mad-hoc* (Version 1.0) [27], *Kernel AODV* (Version 1.5) [23], and *AODV-UU* (Version 0.5) [16]. While it is not clear how well these implementations are tested, they have been used in different testbeds and network simulation environments [28]. On average, the implementations contain 6000 lines of code.

Protocol	Checked Code	Correctness Specification	Environment		State Canonicalization
			network	stubs	
mad-hoc	3336	301	400	100	165
Kernel AODV	4508	301	400	266	179
AODV-UU	5286	332	400	128	185

Table 3: Lines of implementation code vs. CMC modeling code.

Types of Checks	Examples
Generic Assertions	Segmentation violations, memory leaks, dangling pointers.
Routing Loop Invariant	The routing tables of all nodes do not form a routing loop.
Assertions on Routing Table Entries	At most one routing table entry per destination. No route to self in the AODV-UU implementation. The hop count of the route to self is 0, if present. The hop count is either infinity or less than the number of nodes in the network.
Assertions on Message Fields	All reserved fields are set to 0. The hop count in the packet can not be infinity.

Table 4: Properties checked in AODV.

For each implementation, the model consists of a core set of unmodified files. This model executes along with an environment which consists of a network model and simplified implementations (or "stubs") for the implementation functions not included in the model. Table 3 describes the model and environment for these implementations. All three models reuse the same network model. While performing this case study, CMC did not have the functionality to automatically extract and canonicalize system states. The AODV models additionally include code to perform these functions.

As CMC was being developed during this case study, it is difficult to gauge the time spent in building these models as opposed to building the model checker itself. As a rough estimation, it took us two weeks to build the first, mad-hoc model. Building subsequent models was easier, and it took us one more week to build both these models.

Table 4 describes the assertions CMC checked in the AODV implementations. CMC automatically checks certain generic assertions such as segmentation violations. Additionally, the model contains an invariant to check if the routing tables are loop free at all instants. Also, the model checks each message generated and routes inserted into the routing table for specific assertions. Table 3 reflects the lines of code required to add these correctness properties.

CMC found a total of 42 errors. Of these, 35 are unique errors in the implementations and one is an error in the underlying AODV specification. Table 5 summarizes the set of bugs found. The Kernel AODV implementation has 5 bugs (shown in parentheses in the table) that are instances of the same bug in mad-hoc. The AODV specification bug causes a routing loop in all three implementations.

4.4 Comparison with Static Analysis

We then used MC to check the AODV implementations for errors. MC checked for generic errors such as memory leaks and invalid pointer accesses. The entire process of checking the three implementations and analyzing the output for errors took two hours. MC found a total of 34 bugs. We expect that if we invested more effort in writing AODV-specific rules or in running additional checkers that we would have found more errors.

Table 6 compares the bugs found by MC and CMC. It classifies the bugs found into two broad classes depending on the properties violated: generic and protocol specific. In the class of generic errors, our results are similar to the FLASH case study. MC found many more bugs than CMC. Except for one, MC found all the bugs that CMC could find.

The underlying reason is that MC checks all paths in all code that we can compile. It does not require us to abstract away parts of the system, nor does it require coming up with inputs to drive the system to execute a given code path. In contrast, CMC can only execute code triggered by the specific environment model. Of the 13 errors not found by CMC, 6 are in parts of the code that are either not included in the model or stubbed out during environment modeling. For instance, MC found two cases of mishandled `malloc` failures in multicast routing code. All our CMC models omitted this code.

CMC missed more errors due to subtle mistakes in the environment model. For example, the mad-hoc implementation uses a `send_datagram()` function to transmit a packet to the network. A memory leak in the implementation is triggered only when this function fails. In our environment however, we erroneously modeled the `send_datagram()` function to always succeed. Thus, CMC never detected this memory leak. CMC missed a total of 6 errors due to such errors in the environment. MC found 1 more error in dead code that can never be executed by any CMC model.[2]

The one error that MC missed requires reasoning about the length of a linked list. One function in the mad-hoc implementation assumes that the input argument points to a linked list of a particular length. However, when this linked list is allocated in another function, a `malloc` failure can cause the list to be smaller than expected, leading to a null

[2]MC also found a null pointer violation in one of our models! We obviously do not count this error.

	mad-hoc	Kernel AODV	AODV-UU
Mishandling `malloc` failures	4	6	2
Memory Leaks	5	3	0
Use after free	1	1	0
Invalid Routing Table Entry	0	0	1
Unexpected Message	2	0	0
Generating Invalid Packets	3	2 (2)	2
Program Assertion Failures	1	1 (1)	1
Routing Loops	2	3 (2)	2 (1)
Total	18	16 (5)	8 (1)
LOC per bug	185	285	661

Table 5: Number of bugs of each type in the three implementations of AODV. The figures in parenthesis show the number of bugs that are instances of the same bug in the mad-hoc implementation.

		Bugs Found		
		by CMC & MC	by CMC alone	by MC alone
Generic Properties:	Mishandling `malloc` failures	11	1	8
	Memory Leaks	8	-	5
	Use after free	2	-	-
Protocol Specific:	Invalid Routing Table Entry	-	1	-
	Unexpected Message	-	2	-
	Generating Invalid Packets	-	7	-
	Program Assertion Failures	-	3	-
	Routing Loops	-	7	-
	Total	21	21	13

Table 6: Comparing MC and CMC. Note that for the MC results we only ran a set of generic memory and pointer checkers rather than writing AODV-specific checkers. Generating the MC results took less than two hours, rather than the weeks required for the AODV results.

pointer violation. Present static analyzers have difficulty detecting such invariants of heap objects.

In the class of protocol-specific errors, CMC found 21 errors while MC found none. Partly this was because it did not look for them. However, the bulk of these errors would be difficult to catch using static analysis. Properties such as routing loops involve invariants of objects across multiple processes. Detecting such loops statically would require reasoning about the entire execution of the protocol, a difficult task.

Many properties are local to a process, but still difficult to detect statically without generating many false positives. For instance, AODV-UU requires that the routing table of a node does not contain a route to itself. This would imply that the node has to route packets to a neighbor to reach itself. This implementation inserts a route to the *src-id* field in a received route response message. However, there are no checks to ensure that the *src-id* received is not the same as the current node identifier. While a static analyzer could look for such unchecked routing table insertions, it cannot determine which of these are harmless. On a first look it appears that *src-id* can never be the node identifier as no node sends a route response to itself. However, CMC found a specific instance of the protocol ("gratuitous route response") in which such a packet can be generated.

The results from this case study support the use of model checking techniques to find errors in systems. These results also suggest that in order for model checkers to be effective *and* to justify the additional effort, they should emphasize on properties that cannot be checked statically.

5. CASE STUDY: TCP

After our success with AODV, we decided to model check the Linux TCP implementation. The key motivation behind this case study is to evaluate the effectiveness of model checking large and well tested systems. The TCP protocol is mature and the particular implementation we used (from the stable 2.4.19 release of the Linux kernel) is widely used in the Internet today. This implementation contains approximately $50,000$ lines of code.

During the course of our case study, we realized that modeling the environment for a complex system is just *hard.* Section 5.1 describes the problems we faced in modeling the environment for the Linux TCP implementation in the context of our initial failed attempt. After spending months, we realized that the only meaningful way to build an environment is to run the *entire* Linux Kernel along with the TCP implementation in CMC. Section 5.2 describes this approach.

A related problem is that a given environment model might restrict the system behaviors explored by the model checker. As seen in our previous case studies, these restrictions in the environment lead to missed errors. Section 5.3 describes two coverage metrics we used to evaluate a particular environment model, and Section 5.4 describes our effort in using these metrics to refine the environment model. Finally, Section 5.5 presents our model checking results.

5.1 Difficulties in Environment Modeling

All model checkers require that an appropriate model of the environment be provided along with the system. While modeling the environment can be trivial for a relatively small system, doing so for a large system like TCP is not straightforward. Building an environment model requires the following steps.

1. **Defining the System Boundary:** The system to be model checked is typically embedded with other modules in a larger execution context. For instance, the TCP implementation is executes in the Linux kernel. The first step involves defining a boundary between the system and its environment. The modules within this boundary comprise the system.

2. **Closing the System:** The system is then closed by providing stubs for all the interface functions in the system boundary. Additionally, a user has to provide models of entities such as a network that interact with the system.

3. **Providing Environment Triggers:** Once the environment is modeled, a user determines the set of inputs the environment provides to the system model. For each input, the user has to provide an appropriate guard condition that determines when the particular input is enabled.

5.1.1 Failed Approach: The TCP Library Model

In our first attempt, we followed an approach similar to the one that worked for AODV: include a core set of modules in the system and abstract the remaining modules in the environment. However, this approach failed to produce a working model.

Defining the System Boundary: Following conventional wisdom, we attempted to make the checked system as small as possible. Including additional modules into the system increases the system state, and can potentially increase the state space. Starting from the core set of TCP modules, we conservatively added a few tightly coupled modules (such as IP) to simplify the system boundary. Even then, the system boundary consisted of as many as 150 interface functions.

Closing the System: To close the system, we manually provided stub implementations for all the interface functions in the system boundary. Despite our efforts to implement these stubs correctly, it was just impossible to get them right.

Faulty stubs typically result in false behaviors that CMC will (falsely) flag as errors in the checked code. These false positives can be very hard to debug and fix. For instance, after days of debugging we found that a memory leak of a socket structure was caused by incorrect stub implementation in the timer model. The TCP implementation uses a function `mod_timer()` to modify the expiration time of a previously queued timer. This function's returns value depends on whether the timer is pending when the function is called. Our initial stub implementation did not capture this behavior. This incorrect stub confused the reference counting mechanism of the socket structures leading to a memory leak. (As TCP timers are members of the socket structure, a queued timer amounts to an extra reference to the parent socket.)

Providing Environment Inputs: We also had trouble determining the right guard conditions for certain inputs in the environment. For instance, a user process can never get access to a partially connected TCP socket using the standard system call interface. Modeling this required us to carefully disable certain socket functions at different TCP states.

5.1.2 Hard Learned Lessons

All of the problems mentioned above arise because of our insufficient knowledge of the different interfaces within the kernel. It is quite possible that after sufficient iterations of fixing errors in the environment model, we would have converged on a model that implemented all the interfaces accurately. However, subsequent iterations involved bugs that were more subtle and took longer to debug.

Thus it is essential that the system and the environment be split across well-defined and documented boundaries. This greatly simplifies the environment modeling. Also, as these boundaries are less likely to change in future revisions, the same environment model can be reused as the system implementation evolves. While doing so might require the model checker to handle larger states and state spaces, we believe the benefits of a clean environment model outweigh the increase in state sizes.

5.2 The Linux TCP Model

Expanding from the TCP module, the model can only be bounded by the following two well-defined interfaces: the system call interface that defines the interaction between user processes and the kernel, and the "hardware abstraction layer" that defines the interaction between the kernel and the architecture. Bounding the TCP module at these two interfaces includes the entire kernel in our model.

To run the entire kernel in CMC, we had to "port" the kernel to CMC by providing a suitable hardware abstraction layer. Our approach is very similar to User Mode Linux (UML) [37].

Once the system boundary is clearly defined, defining the inputs and their guard conditions became straightforward. Two user processes, one behaving as a TCP server and the other as a TCP client triggered the TCP model by making standard socket calls. The TCP model accessed the CMC network through an appropriate network device driver. The environment nondeterministically fired the clock interrupt to enable timers.

Apart from the generic errors that CMC automatically checks, we added a few TCP specific assertions to our model. For instance, the model checks if a packet generated by the implementation has a valid checksum. To check the implementation for protocol compliance, the model simultaneously runs a TCP reference model along with the implementation. The reference model performs the basic state machine transitions described in [34]. The model provides the same inputs to both the implementation and the reference model and checks if their states are consistent.CMC reports any inconsistency as a protocol violation error.

5.3 Measuring the Search Effectiveness

Even after a working environment model is built, one problem still remains: the environment at hand might not trigger all the behaviors in the system. The TCP protocol is complex and includes many functionalities that are enabled by various configuration variables and socket options. While enabling more inputs can potentially trigger more behaviors in the system, doing so can drastically increase the state space. This clearly indicates a need for search effectiveness metrics by which different environment models can be evaluated. As CMC almost never completes the state space search, such an effectiveness measure can also indicate how well a particular system is tested.

We used two ways to measure the effectiveness of the search. The first measure is the line coverage achieved during model checking. While this measure need not correspond to how well the system has been tested, it is helpful in detecting the parts that are *not* tested.

The second measure, which we call "protocol coverage," corresponds to the behaviors of the protocol tested by the model checker. We calculate protocol coverage as the line coverage achieved in the TCP reference model mentioned above. This roughly represents the degree to which the protocol transitions have been explored.

5.4 Iterative Environment Refinement

We used the metric discussed in the previous section to iteratively refine the model to explore more system behaviors. In many instances, low coverage helped in pointing out errors in our environment model.

Table 7 describes the coverage achieved during the model refinement process. For a particular model, we measured coverage cumulatively using three search techniques: breadth-first, depth-first, and random. In random search, each generated state is given a random priority. Table 7 also reports the branching factor of the state space as a measure of its size. For the first three models the branching factor is calculated from the number of states in the queue at depth 10 during a breadth first search. For the fourth model, CMC ran out of resources at depth 8, and the branching factor is calculated at this depth.

The first model consists of a TCP client communicating with a TCP server. Once the connection is established, the client and server exchange data in both directions before closing the connection. This standard model discovered two protocol compliance bugs in the TCP implementation.

Starting from this model, we iteratively refined the model by manually inspecting the line coverage and protocol coverage to determine the behavior we wanted to include next. In the second model, the server nondeterministically decides to actively initiate a connection. This enables additional transitions in the protocol that handle simultaneous connection of two peers. In the third model, both the client and the server nondeterministically decide to close a connection during data transfer. This improved the protocol coverage and resulted in the discovery of two more errors.

Still, the environment did not allow enough "bad" behavior. As an attempt to generate "random" packets, we nondeterministically toggled certain key control flags in the TCP packet. These corrupted packets triggered a lot of error recovery code in the implementation. But they also resulted in an enormous increase in the state space.

Tweaking the environment the right way to achieve a more effective search still remains an interesting but unsolved problem. It is not clear how much of this refinement process can be automated.

5.5 TCP Model Checking Results

We have detected four errors in the Linux TCP implementation. All are instances where the implementation fails to meet the TCP specification. These errors are fairly complex and require an intricate sequence of events to trigger the error.

The following is a brief description of two of the bugs CMC found. While a detailed understanding of the TCP protocol [34] is required to completely understand these bugs, the purpose of the description below is to provide a general flavor of errors CMC is able to find.

The first bug involves the processing of RST (reset) packets. A RST packet is used to indicate an abnormal close of a connection. In response to a RST packet, a TCP implementation is required to free any resources used by the connection and gracefully inform the application. However, in the *SYN_RCVD* state, the Linux TCP implementation fails to process certain RST packets. Specifically, a RST packet without the ACK bit set will be ignored. This results in unnecessary lockup of kernel resources.

The second bug involves an inappropriate handling of the ACK field in a packet. An ACK acknowledges the reliable transfer of a data segment to the sender. Also, specific ACKs indicate an opportunity for the sender to increase its congestion window and send more data. However, using wrong ACKs for this purpose can result in a decrease in the data transfer performance. CMC found an instance where the implementation used a duplicate ACK packet to increase its congestion window. The specification requires that such duplicate packets be ignored.

5.6 Lessons from Model Checking TCP

This section summarizes some key lessons we learned during our TCP model checking effort.

1. No model is as good as the implementation itself. Any modification, translation, approximation done is a potential for producing false positives, danger of checking far less system behaviors, and of course missing critical errors.

2. Any manual work required in the model checking process becomes immensely difficult as the scale of the system increases. In order to scale, model checker should require as little user input, annotations and guidance as possible.

3. If an unit-test framework is not available, then define the system boundary only along well-known, public interfaces.

4. Try to cover as much as possible: the more code you trigger, the more bugs you find, and more useful model checking is.

Model checking currently requires spending a lot of effort on designing the environment. Table 8 gives crude measurements of the environment modeling effort for TCP and AODV. Most of this effort can be eliminated if the system is designed for unit testing, as much of the work needed to

	Description	Line Coverage	Protocol Coverage	Branching Factor	Additional Bugs
1	Standard server and client	47.4 %	64.7 %	2.91	2
2	Model 1 + simultaneous connect	51.0 %	66.7 %	3.67	0
3	Model 2 + partial close	52.7 %	79.5 %	3.89	2
4	Model 3 + message corruption	50.6 %	84.3 %	7.01	0
	Combined Coverage	55.4 %	92.1 %		

Table 7: Coverage achieved during model refinement. The branching factor is a measure of the state space size.

Model	Closing the System		Environment Triggers		Correctness Specifications	
	lines of code	time	lines of code	time	lines of code	time
AODV	568	1 week	400	1 week	311	1 day
TCP library	3406	1.5 months	806	2 days	50	1 day
TCP with UML Linux	6583	3 weeks	1415	2 weeks	718	3 days

Table 8: Effort required for modeling the environment, in terms of the lines of code required and the time taken. The time reported here is approximate and is only intended to show the relative effort required. Also for the TCP with UML Linux model, we closed the system by modifying the UML code. We counted the number of lines in all the files in which at least one modification was made.

unit test code is identical to that needed to build an environment model. Unfortunately, this is not the case with most systems we encounter.

One way to mitigate the environment modeling effort is to provide model checking compatible standard libraries. As many software systems use the same libraries, the effort involved can be amortized over many model checking projects. Providing tools to automate the environment modeling process is an interesting area of future research.

6. GENERAL LESSONS

6.1 Myth: model checking = no false positives

A common claim, at least among static analysis researchers is that model checkers do not suffer from false positives. They most certainly do. In several of our projects, the majority of the errors found during development were false positives, primarily due to under-constrained or misspecified environments and "harness" code. In a real system, there will be huge numbers of interfaces with typically non-obvious or at least rich semantics. At some point a line has to be drawn and these interfaces faked so that the model checker can work on a subset of the system. However, it is easy to get such functions slightly wrong. Since the point of model checking is to find corner cases, it will persistently root out misunderstandings in environmental interfaces. These mistakes can take several days to track down, since they often just lead to the model-checked code crashing. The answer to the question "is it a bug in the code or in our model?" comes down on the latter irritatingly often. It is hard to overestimate the difficulty in correctly modeling the parts of the system you attempt to cut out.

6.2 Ease-of-inspection really matters

A surprise for us from our static analysis work was just how important ease-of-inspection is. Errors that are too hard to inspect might as well not be flagged since the user will ignore them (and, for good measure, may ignore other errors on general principle). For example, the commercial PREfix tool explicitly avoided finding race conditions and deadlocks simply because the errors were too difficult to inspect [33]. Our initial commercial efforts have similarly scaled back on analysis sophistication to focus on errors that were easy to reason about. Given two bugs, one easy to examine and one hard, then in the absence of additional discriminatory information (severity, likelihood) the first is better.

6.3 Myth: more analysis is always better

We, like many others in the field, initially believed that more analysis was always better then less, whether it came in the form of model checking, simulation, or deeper static analysis. This view was simplistic: adding more analysis does not always improve results and can even make them worse. The ideal error is easy to diagnosis, is a true error, and is easy to fix. Generally speaking, the more analysis required to find an error the worse it is on all three of these metrics:

1. Typically, the more analysis used to find an error, the harder the error is to reason about. During inspection, the user must mentally emulate each analysis step (how aliases were determined, whether an interprocedural call path is feasible, etc) to determine how plausible they are and how they can be countered. The more steps the more work this emulation becomes.

2. As the number of analysis steps increases, so does the chance that one of them went wrong. If there is no analysis, then there can be no approximation mistakes. The more analysis there is, the more widespread the effects of a mistake.

3. Hard errors to find are often hard errors to fix.

As an example, our initial static checkers were almost syntactic [14]. As a result, the errors they found were almost certainly errors and were trivial to inspect. As we added more interprocedural support, errors became more difficult to inspect. In fact, we often deliberately reverted to much weaker analysis to find errors than our system supports, simply because specializing to these error classes cherry picks easy-to-diagnose bugs. The most common case is that we often design checkers explicitly to use intraprocedural analysis despite the fact that our system supports transparent interprocedural analysis: Local bugs are much easier to diagnose than interprocedural ones. Even if we do use strong analysis, we almost always rank error reports based on the number of analysis steps required. For example, bugs involving aliasing or spanning procedure calls are demoted below those that do not.

6.4 Myth: all bugs matter

We initially thought that all bugs matter and all bugs will be fixed. This is not true. If you find a small number of bugs, people will fix them all. If you find thousands, they will not. We have observed this both with open source projects and with commercial systems — many of the bugs we have detected are still open. Prior to our work, the PREfix group observed a similar dynamic: giving someone a stack of 1,000 defects is an effective way to elicit a blank stare and then the question "that's great, but which ones matter?"

Its not enough to find a lot of bugs. As tools become more effective, this will become more obvious. What users really want is to find the 5-10 bugs that "really matter" — e.g., the ones that will hurt a large number of customers, absorb the bulk of debugging time, etc. A general, not-unreasonable belief is that bugs will follow a 90-10 distribution. Thus, out of 1000 errors, 100 will account for most of the pain and 900 will be a waste of resources to fix. In fact, fixing these 900 errors may worsen system quality by introducing additional errors or draining resources from other efforts (testing, code reviews). Unfortunately, while current tools can easily segregate errors into different types that can be inspected by priority (security holes before storage leaks before null pointer dereferences) they lack effective methods for identifying the "most important" errors. Identifying these would be a good area of future research.

7. RELATED WORK

This paper discusses our experiences in using two approaches to find errors in systems: static analysis and model checking. While the benefits and difficulties of these two approaches are individually well understood, to our knowledge this is the first paper to compare the two approaches and evaluate them with respect to the effort involved and the results produced on different sets of properties.

The area of using static analysis for bug finding has become extremely active. Some of the more well-known tools include include PREfix (mentioned above), ESP [11], ESC [17, 25], the Warlock race detector [35], and Wagner's security work [39, 38]. Others have gone towards more language-based approach, such as Vault [12] and Foster et al [18]. Or CCured [30], a hybrid static-dynamic tool for detecting memory errors that uses a type inference algorithm to eliminate the need for many dynamic checks. Finally, the SLAM project combines aspects of both static analysis and model checking [1, 2].

Many verification tools statically extract an abstract model for a given system. Bandera [9] is a sophisticated model extractor for Java programs. It uses a given temporal property as a slicing criteria to extract relevant parts of the system. Also, Bandera accepts user provided annotations to abstract data values to specific subranges. FeaVer [21] uses a set of user defined mappings to extract abstract models from C code. These models are checked using the SPIN model checker.

Two prior verification tools have used the idea of directly model checking the implementation. Verisoft [19] executes C programs and has been successfully used to check communication protocols [5]. Java PathFinder [3] consists of a modified Java virtual machine that can check concurrent Java programs. The difficulties of environment modeling have been discussed before both in the context of Verisoft [8] and Java PathFinder [32].

8. CONCLUSION

This paper has described tradeoffs between both static analysis and model checking, as well as some of the surprises we encountered while applying model checking to large software systems.

9. ACKNOWLEDGMENTS

This paper recounts research done with others. In particular, we thank David Lie and Andy Chou for their discussions of learned in model checking the FLASH code (of which they did the majority of the work). David Park did significant work helping develop CMC and on model-checking the TCP implementation.

We thank Priyank Garg, Rushabh Doshi, Russell Greene, Rafael Hernandez and Adam Simpkins for discussions on model checking and willingness to endure using CMC in its prototype form. We thank Anthony Hui and Ted Kremenek for valuable proof-reading assistance and Willem Visser for thoughtful comments on a previous version of this paper.

10. REFERENCES

[1] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software (LNCS 1885, Springer)*, August/September 2000.

[2] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[4] 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.

[5] Sathish Chandra, Patrice Godefroid, and Christopher Palm. Software model checking in practice: An industrial case study. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2002.

[6] Benjamin Chelf, Seth Hallem, and Dawson Engler. How to write system-specific, static checkers in metal.

In *Invited Paper. Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2002.

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

[8] Christopher Colby, Patrice Godefroid, and Lalita Jategaonkar Jagadeesan. Automatically closing open reactive programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357, 1998.

[9] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000*, 2000.

[10] C.Perkins, E. Royer, and S. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. IETF Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-10.txt, January 2002.

[11] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation*, 2002.

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

[13] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[14] 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.

[15] 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.

[16] Erik Nordstrom *et al.* AODV-UU Implementation. http://user.it.uu.se/ henrikl/aodv/.

[17] C. Flanagan, M. R. K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *2002 Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[18] J.S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999.

[19] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[20] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[21] G. Holzmann and M. Smith. Software model checking: Extracting verification models from source code. In *Invited Paper. Proc. PSTV/FORTE99 Publ. Kluwer*, 1999.

[22] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[23] Luke Klein-Berndt and et.al. Kernel AODV Implementation. http://w3.antd.nist.gov/wctg/aodv_kernel/.

[24] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

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

[26] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[27] F. Lilieblad and et.al. Mad-hoc AODV Implementation. http://mad-hoc.flyinglinux.net/.

[28] H. Lundgren, D. Lundberg, J. Nielsen, E. Nordstrom, and C. Tschudin. A large-scale testbed for reproducible ad hoc protocol evaluations. In *IEEE Wireless Communications and Networking Conference*, March 2002.

[29] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[30] G.C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[31] S. Park and D.L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the 8th ACM Symposium on Parallel Algorithsm and Architectures*, pages 288–296, June 1996.

[32] J. Penix, W. Visser, E. Engstron, A. Larson, and N. Weininger. Verification of time partitioning in the deos real-time scheduling kernel. In *22nd International Conference on Software Engineering (ICSE)*, 2000.

[33] J. Pincus. Personal communication. PREfix did not target data races because of the user-interface complexities in reporting and diagnosis., March 2003.

[34] J. Postel. Transmission control protocol. RFC 793, USC/Information Sciences Institute, September 1981.

[35] N. Sterling. WARLOCK - a static data race analysis tool. In *USENIX Winter*, pages 97–106, 1993.

[36] U. Stern and D. L. Dill. A New Scheme for Memory-Efficient Probabilistic Verification. In *IFIP TC6/WG6.1 Joint International Conference on*

Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, 1996.

[37] The User-mode Linux Kernel. http://user-mode-linux.sourceforge.net/.

[38] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[39] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *The 2000 Network and Distributed Systems Security Conference. San Diego, CA*, February 2000.

APPENDIX

A. CHECKER EXAMPLE

Figure 1 provides an example of a *metal* checker. This checker checks for the rule "`WAIT_FOR_DB_FULL` must come before `MISCBUS_READ_DB`." Checkers are built of variable declarations, states, patterns that can match in the state, and arbitrary actions written in C code that perform operations when their transition occurs (allowing them to check more than finite state properties). The checker declares two variables, `addr` and `buf` as wildcard variables that will pattern match any C expression (`any_expr`). The remainder of the checker defines a simple state machine with a single state, `start`. SMs start execution in the first state they define (in this case `start`). From its start state, the SM uses two patterns to search for all uses of the macros `WAIT_FOR_DB_FULL` and `MISCBUS_READ_DB`. When either matches, the scalar expression passed as their arguments will be placed in `addr` and, for `MISCBUS_READ_DB`, `buf`. The matching rule will then cause the SM to transition to the (optional) state (the token after the `==>` operator) and then execute the (optional) action. If a rule's state is omitted, the SM remains in the current state. The `start` state has two rules. If the first rule's pattern for `WAIT_FOR_DB_FULL` matches, then the handler has correctly waited for its data buffer to fill, and any subsequent read on this execution path will be valid. Thus, the checker transitions to the `stop` state, which causes it to stop running on the current path. If the second rule's pattern matches, then the execution path being checked did not wait for its buffer to fill and it had a buffer race condition error. This rule's associated action will then print out an error message. Since the rule does not give a transition state, the checker will remain in the `start` state to catch further violations along the path.

```
sm wait_for_db {
 // Declare two variables 'addr' and 'buf' that can
 // match any expression. */
 decl any_expr addr, buf;

 // The checker begins in the first state (here 'start').
 // This state searches for two patterns conjoined
 // with the '|' operator. */
 start:
  // The handler is allowed to read the data buffer
  // after calling 'WAIT_FOR_DB_FULL' --- once the
  // pattern below matches, we transition to the
  // 'stop' state, which stops checking on this
  // path.
    { WAIT_FOR_DB_FULL(addr); } ==> stop

  // If we hit a read of the data buffer in this
  // state, the handler did not do a WAIT_FOR_DB_FULL
  // first so emit an error and continue checking. */
  | { MISCBUS_READ_DB(addr, buf); } ==>
      { err("Buffer not synchronized"); }
  ;
}
```

Figure 1: A simplified *metal* checker to find violations of the rule "`WAIT_FOR_DB_FULL` must come before `MISCBUS_READ_DB`." It searches FLASH code looking for any data buffer read (using `MISCBUS_READ_DB`) not preceded by a synchronizing wait call (using `WAIT_FOR_DB_FULL`).