Static analysis versus software model checking for bug finding

Dawson Engler and Madanlal Musuvathi
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.
{engler,madan}@cs.stanford.edu

1 Introduction

This paper describes 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 3, checked FLASH cache coherence protocol implementation code [20]. 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) [6] and then, in follow-on work, applied model checking [21]. A startling result (for us) was that despite model checking's power, it found far fewer errors than relatively shallow static analysis: eight bugs versus 34.

The second project, described in Section 4, checked three AODV network protocol [8] implementations. Here we first checked them with CMC [24], a model checker that directly checks C implementations. We then statically analyzed them. Model checking worked well, finding 42 errors (roughly 1 per 300 lines of code), about half of which involve protocol properties difficult to check statically. However, in the class of properties both methods could handle, static analysis found more errors than model checking. Also, it took much less effort: 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. The most startling result here was just how difficult it is to model check real code that was not designed for it. It turned out to be easier to run the *entire* Linux Kernel along with the TCP implementation in CMC rather than cut TCP out of Linux and make a working test harness. We found 4 bugs in the Linux TCP implementation.

The main goal of this paper is to compare the merits of the two approaches for finding bugs in system software. In the properties that could be checked by both methods, static analysis is clearly more successful: it took less time to do the analysis and found more errors. The static analysis simply requires that the code be compiled, while model checking a system requires a carefully crafted environment model. Also, static analysis can cover *all* paths in the code in a straightforward manner. On the other hand, a model checker executes only those paths that are explicitly triggered by the environment model. A common misconception is that model checking does not suffer from false errors, while these errors typically inundate a static analysis result. In our experience, we found this not to be true. False execution paths in the model checker can be triggered by erroneous environments, leading to false errors. These errors can be difficult to trace and debug. Meanwhile, false errors in static analysis typically arise out of infeasible paths, which can be eliminated by simple analysis or even unsubstantial manual inspection.

The advantage of model checking is in its ability to check for a richer set of properties. Properties that require reasoning about the system execution are not amenable to static checking. Many protocol specific properties such as routing loops and protocol deadlocks fall in this category. A model checker excels in exploring intricate behaviors of the system and finding errors in corner cases that have not been accounted for by the designers and the implementors of the system. However, the importance of checking these properties should significantly over-weigh the additional effort required to model check a system.

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 are very much in the "model checking camp" and intend to continue research in the area. One of the goals of this paper is to recount what surprised us when applying model checking to large real code bases. 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.

2 The Methodologies

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.

2.1 The model checking approach

All of our case studies use traditional explicit state model checkers [10, 18]. 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 estimates 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 [7] and Feaver [17]. Our second approach eliminates extraction entirely by directly model checking the implementation code. It is similar to Verisoft [15], which executes C programs and has been successfully used to check communication protocols [4] and Java PathFinder [2], which uses a modified Java virtual machine that can check concurrent Java programs.

2.2 The static analysis approach

The general area of using static analysis for bug finding has become extremely active. Some of the more well-known static tools include include PREfix [3], ESP [9], ESC [14], MOPS [5] and SLAM [1], which combines aspects of both static analysis and model checking.

The static tool approach discussed in this paper is based on compiler extensions ("checkers") that are dynamically linked into the compiler and applied flow-sensitively down a control-flow graph representation of source code [11]. Extensions can perform either intra- or inter-procedural analysis at the discretion of the checker writer. In practice, this 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 ("metacompilation" or MC), 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. In particular, these features generally reduce the work needed to find bugs as compared to other static analysis techniques.

First, our approach is unsound. Code with errors can pass silently through a checker. Our goal has been to find the maximum number of bugs with the minimum number of false positives. In particular, when checkers cannot determine a needed fact they do not emit a warning. In contrast, a sound approach must conservatively emit error reports whenever it cannot prove an error cannot occur. Thus, unsoundness lets us check effectively properties that done soundly would overwhelm the user with false positives.

Second, we use relatively shallow analysis as compared to a simulation-based approach such as in PREfix [3]. [1] Except for a mild amount of path-sensitive analysis to prune infeasible paths [16], we do not: model the heap, track most variable values, or do sophisticated alias analysis. A heavier reliance on simulation would increase the work of using the tool, since these often require having to build accurate, working models of the environment and of missing code. In a sense simulation pushes static analysis closer to model checking, and hence shares some of its weaknesses as well as strengths.

Third, our approach tries to avoid the need for annotations, in part by using statistical analysis to infer checkable properties [12]. The need for annotations would dramatically increase the effort necessary to use the tool.

3 Case study: FLASH

This section describes our experience checking FLASH cache coherence protocol code using both static analysis and model checking [20]. The FLASH multiprocessor implements cache coherence in software. While this gives flexibility it places a serious burden on the programmer. The code runs 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 FLASH protocols with static analysis and four with model checking. Protocols ranged from 10K to 18K lines and have long control flow paths. The average path was 73 to 183 lines of code, with a maximum of roughly 400 lines. Intra-procedural paths that span 10-20 conditionals are not uncommon. For our purposes, this code is representative of the low-level code used on a variety of embedded systems: highly optimized, difficult to read, and 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 one protocol had even been model checked using a manually constructed model [25].

3.1 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 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). 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 or contexts.

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 cache requests, most rules involve correct message handling. The most common errors were not deallocating message buffers (9 errors) and 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 that it can eliminate. (The severity of the errors made the given rate perfectly acceptable.)

3.2 Model checking FLASH

While static analysis worked well on code-visible rules, it has difficulty with properties that were not visible in the source code, but rather implied by it, such as invariants over data structures or values produced by code operations. For example, that the sharing list for a dirty cache line is empty or that the count of sharing nodes equaled the number of caches a line was in. On the other hand, these sort of properties and FLASH structure in general are well-suited to model checking.

Unfortunately, the known hard problem with using model checking on real code is the need to write a specification (a "model") that describes what the software does. For example, it took a graduate student several months to build

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

Rule	LOC	Bugs	False
WAIT_FOR_DB_FULL must come before MISCBUS_READ_DB	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).	29	18	2
Message buffers must be: allocated before use, deallocated after, and not used after deallocation.	94	9	25
Message handlers can only send on pre-specified "lanes."	220	2	0
Total	355	33	28

Table 1: Representative FLASH rules 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.

hand-written, heavily-simplified model of a single FLASH protocol [25]. Our model checking approach finessed this problem by using static analysis to automatically extract models from source code. We started the project after noticing the close correspondence between a hand-written specification of FLASH [25]) with the implementation code itself. FLASH code made heavy use of stylized macros and naming conventions. These "latent specifications" [12] made it relatively easy to pick out the code relevant to various important operations (message sends, interactions with the I/O subsystem, etc) and then automatically translate them to a checkable model.

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.

Table 2 lists a representative subset of the rules we checked that static analysis would have had difficulty with. Surprisingly, there were relatively few errors in these properties as compared to the more shallow properties checked with static analysis.

3.3 Myth: model checking will find more bugs

The general perception within the bug-finding community 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, either in this case study or in the next one. For FLASH, 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 [6]; publishing a follow-on paper for a technique that required more work and found fewer was worrisome. In the end, only two of the eight bugs found with model checking had been missed by static analysis. Both 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 just requires you compile it. 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 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 cost must be paid for each new checked system and, given finite resources, it can preclude checking new code or

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.

limit checking to just code or properties whose environment can be specified with a minimum of fuss. A good example for FLASH is that time limitations caused us to skip checking the "sci" protocol, thereby missing five buffer management errors (three serious, two minor) found with static analysis.

3.4 Summary

Static analysis works well at checking properties that clearly map to source code constructs. Model checking can similarly leverage this feature to automatically extract models from source code.

As this case study shows, many abstract rules can be checked by small, simple static checkers. Further, the approach was effective enough to find errors in code that was (1) not written for verification and (2) had been heavily-tested for over five years.

After using both approaches, static analysis had two important advantages over model checking. First, in sharp contrast to the thorough, working code understanding demanded by model checking, static analysis allowed us to understand little of FLASH before we could check it, mainly how to compile it and a few sentences describing rules. Second, static analysis checks all paths in all code that you can compile. Model checking only checks code you can run; and of this code, only of paths you execute. This fact hurt its bug counts in the next case study as well.

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

While automatically slicing out a model for FLASH was far superior to hand constructing one, the approach had two problems. First, it required that the user have a intimate knowledge of the system, so that they could effectively select and automatically mark stand-alone subparts of it. Second, Murφ, like most modeling languages lacks many C constructs such as pointers, dynamic allocation, and bit operations. These omissions make general translation difficult.

We countered these problems by building CMC, a model checker that checks programs written in C [24]. 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 required 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 an explicit state model checker that works more or less like Murφ though it lacks many of Murφ's more advanced optimizations. As CMC checks a full implementation rather than an abstraction of it, it must handle much larger states and state spaces. It counters the state explosion problem by using aggressive approximate reduction techniques such as hashcompaction [27] and various heuristics [24] to slice unnecessary detail from the state.

4.2 AODV Overview

AODV (Ad-hoc On-demand Distance Vector) protocol [8] is a loop-free routing protocol for ad-hoc networks. It is designed to handle mobile nodes and a "best effort" network that can lose, duplicate and corrupt packets.

AODV guarantees that the network is always free of routing loops. If an error in the specification or implementation causes a routing loop to appear in the network, the protocol has no mechanism to detect or recover from them, allowing the loop to persist forever, completely breaking the protocol. Thus, it is crucial to comprehensively test both the AODV protocol specification and any AODV implementation for loop freeness as thoroughly as possible.

AODV is relatively easy to model check. Its environmental model is greatly simplified by the fact that the only input it deals with are user requests 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

We used CMC to check three publicly-available AODV implementations: *mad-hoc* (Version 1.0) [22], *Kernel AODV* (Version 1.5) [19], and *AODV-UU* (Version 0.5) [13]. While it is not clear how well these implementations are tested, they have been used in different testbeds and network simulation environments [23]. On average, the implementations contain 6000 lines of code.

Assertion Type	Examples
Generic	Segmentation violations, memory leaks, dangling pointers.
Routing Loop	The routing tables of all nodes do not form a routing loop.
Routing Table	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.
Message Field	All reserved fields are set to 0. The hop count in the packet can not be infinity.

Table 3: Properties checked in AODV.

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

Table 4: Lines of implementation code vs. CMC modeling code.

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 4 describes the model and environment for these implementations. All three models reuse the same network model.

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 3 describes the assertions CMC checked in the AODV implementations. CMC automatically checks certain generic assertions such as segmentation violations. Additionally, the protocol model checks that routing tables are loop free at all instants and that each generated message and route inserted into the table obey various assertions. Table 4 gives 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 Static AODV checking: more paths + more code = more bugs

We also did a cursory check of the AODV implementations using a set of static analysis checkers that looked 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. Static analysis found a total of 34 bugs.

Table 6 compares the bugs found by static analysis and CMC. It classifies the bugs found into two broad classes depending on the properties violated: generic and protocol specific. For generic errors, our results matched those in the FLASH case study: static analysis found many more bugs than model checking. Except for one, static analysis found all the bugs that CMC could find. As in our previous case study (§3.3), the fundamental reason for this difference is that static analysis can check all paths in all code that you can compile. In contrast, model checking can only check 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 cut out during environment modeling. For instance, static analysis found two cases of mishandled `malloc` failures in multicast routing code. All our CMC models omitted this code.

Additionally, CMC missed errors because of subtle mistakes in its environment model. For example, the mad-hoc implementation uses the `send_datagram` function to send a packet and has a memory leak when the function fails. However, our environment erroneously modeled the `send_datagram` as always succeeding. CMC thus missed this memory leak. Such environmental errors caused CMC to miss 6 errors in total. Static analysis found one more more error in dead code that can never be executed by any CMC model.[2]

4.5 Where model checking won: more checks = more bugs

In the class of protocol-specific errors, CMC found 21 errors while static analysis found none. While this was partly because we did not check protocol-specific properties, many of the errors would be difficult to find statically. We categorize the errors that were found my model checking but missed by static analysis into three classes and describe them below.

By executing code, a model checker can check for properties that are not easily visible to static inspection (and thus static analysis). Many protocol specific properties fall in this class. Properties such as deadlocks and routing loops involve invariants of objects across multiple processes. Detecting such loops statically would require reasoning about

[2]Static analysis also found a null pointer violation in one of our environment models! We 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		
		CMC & MC	CMC alone	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 static analysis (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 about two hours, rather than the weeks required for AODV.

the entire execution of the protocol, a difficult task. Also, present static analyzers have difficulty analyzing properties of heap objects. The error in Figure 1 is a good example. This error requires reasoning about the length of a linked list, similar to many heap invariants that static analyzers have difficulty with. Here, the code attempts to allocate `rerrhdr_msg.dst_cnt` temporary message buffers. It correctly checks for `malloc` failure and breaks out of the loop. However, it then calls `rec_rerr` which contains a loop that assumes that `rerrhdr_msg.dst_cnt` list entries were indeed allocated. Since the list has fewer entries than expected, the code will attempt to use a null pointer and get a segmentation fault.

A second advantage model checking has is that it 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 such as CMC that runs 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. and so minimize false positives by looking for errors only in specific analyzable contexts.

A good example is the error CMC found in the AODV specification, shown in Figure 2. This error arises because the specification elides a check on the sequence number of a received packet. Here, the node receives a packet with a stale route with an old sequence number. The code (and the specification) erroneously updates the sequence number of the current route without checking if the route in the packet is valid. This results in a routing loop. Once the cause of the routing loop is known, it is possible (and easy) to statically ensure that all sequence number updates to the routing table from any received packets involve a validity check. However, there are only a few places where such specialized checks can be applied, making it hard to recoup the cost of writing the checker. Moreover, exhaustively enumerating all different causes for a routing loop is not possible. On the other hand, a model checker can check for actual errors without the need for reasoning about their causes.

In a more general sense, model checking's end-to-end checks mean it can give guarantees much closer to total correctness than static analysis can. No one would be at all surprised if code that passed all realistic static checks immediately crashed when actually run. On the other hand, given a good environment model and input sequences, it is much more likely that model checked code actually works when used. Because model checking can verify the code was actually totally *correct* on the executions that it tested, then if state reduction techniques allow these executions to cover much of the initial portion of the search space, it will be difficult for an implementation to efficiently get into new, untested areas. At risk of being too optimistic this suggests that even with a residual of bugs in the model checked implementation it will be so hard to trigger them that they are

```
// aodv_deamon.c:aodv_recv_message
   for(rerri=0; rerri<rerrhdr_msg.dst_cnt;rerri++) {
      // Step 1: break with < dst_cnt elements in rerrhdr_msg list.
      if (!(tp = malloc(...)))
         break;
      tp->next = rerrhdr_msg.unr_dst;   // enqueue onto list.
      rerrhdr_msg.unr_dst = tp;
      ...
   }
   // Step 2: rec_rerr assumes dst_cnt elements in rerrhdr_msg
  rec_rerr(&info_msg, &rerrhdr_msg);
  ...
int rec_rerr(struct info *tmp_info, struct rerrhdr *rh) {
   ...
   // Step 3: iterates rh->dst_cnt times even if not that many elements
   for(i = 0, t = rh->unr_dst; i < rh->dst_cnt; i++, tp = tp->next) {
     // ERROR: tp can be null!
     tmp_rtentry = getentry(tp->unr_dst_ip);
```

Figure 1: The one memory error missed by static analysis: requires reasoning about values, and is a good example of where model checking can beat static analysis.

```
// madhoc:rerr.c:rec_rerr
//    recv_rt: route we just received from network.
//    cur_rt: current route entry for same IP address.
cur_rt = getentry(recv_rt->dst_ip);
if(cur_rt != NULL && ...) {
      // Bug: updates sequence number without checking that
      // received packet newer than route table entry!
      cur_rt->dst_seq = recv_rt->dst_seq;
```

Figure 2: The AODV specification error. A common pattern: this bug could have been caught with static analysis, but there were so few places to check that it would be difficult to recover the overhead of building checker.

effectively not there.

A final model checking advantage was that there were true bugs that both methods would catch, but because they "could not happen" would be labeled as false positives when found with static analysis. In contrast, because model checking produces an execution trace to the bug, they would be correctly labeled. The best example was a case where an AODV node receives a "route response" packet in reply to a "route request" message it has sent. The code first looked up the route for the reply packet's IP address and then used this route table entry without checking for null. While a route table lookup can return null in general, this particular lookup "cannot be null," since a node only sends route requests for valid route table entries. If this unchecked dereference was flagged with static analysis it would be labeled a false positive. However, if the node (1) sends this message, (2) reboots, and (3) receives the response the entry can be null. Because model checking gave the exact sequence of unlikely events the error was relatively clear.

4.6 Summary

The high bit for AODV: model checking hit more properties, but static hit more code; when they checked the same property static won. The latter was surprising since it implies that most bugs were shallow, requiring little analysis, perhaps because code difficult for the analysis to understand is similarly hard for programmers to understand. As with FLASH, the difference in time was significant: hours for static versus weeks for model checking.

One view of the trade-off between the approaches is that static analysis checks code well, but checks the implications of code relatively poorly. On the other hand, model checking checks implications relatively better, but because of its its problems with abstraction and coverage, can be less effective checking the actual code itself.

These results suggest that while model checking can get good results on real systems code, in order to justify their significant additional effort they must target properties not checkable statically.

5 Case study: TCP

This section describes our efforts in model checking the Linux TCP implementation. We decided to check TCP after the relative success of AODV since it was the hardest code we could think of in terms of finding bugs. There were several sources of difficulty. First, the version we checked (from Linux 2.4.19) is roughly ten times larger than AODV code (50K lines versus 6K). Second, it is mature, often frequently audited code. Third, since almost all Linux sites constantly use TCP, it is one of the the heaviest-tested pieces of open source code around.

We had expected TCP would require only modest more effort than AODV. As Section 5.1 describes below, this expectation was wildly naive. Further, as Section 5.2 shows, it is a very different matter to get code to run at all and getting it to run so that you can comprehensively test it.

5.1 The environment problem: lots of time, lots of false positives

The system to be model checked is typically present in a larger execution context. For instance, the Linux TCP implementation is present in the Linux kernel, and closely interacts with other kernel modules. Before model checking, it is necessary to extract the relevant portions of the system to be model checked and create an appropriate *environment* that allows the extracted system to run stand alone. This environment should contain *stubs* for all external functions that the system depends on. With an implementation-level model checker like CMC, this process is very similar to building a harness for unit testing. However, building a comprehensive environment model for a large and complex system can be difficult. This difficulty is known to an extent in the model checking literature, but is typically underplayed.

Extracting code amounts to deciding for each external function that the system calls, whether the function should be included in the checked code base, or to instead create a stub for the function and include the stub in the environment model. The advantages of including the function in the checked code are (1) the model checker executes the function and thus can potentially find errors in it (2) there is no need to create the stub or to maintain the stub as the code evolves. However, including an external function in the system has two downsides: (1) this function can potentially increase the state space and (2) it can call additional functions for which stubs need to be provided.

Conventional wisdom dictates that one cut along the narrowest possible interface. The idea is that this requires emulating the fewest possible number of functions, while minimizing the state space. However, while on the surface this makes sense, it was an utter failure for TCP. And, as we discuss below, we expect it to cause similar problems for any complex system.

5.1.1 Failure: building a kernel library

Our first attempt to model check TCP did the obvious thing: we cut out the TCP code proper along with a few tightly coupled modules such as IP and then tried to make a "kernel library" that emulated all the functions this code called. Unfortunately, TCP's comprehensive interaction with the rest of the kernel meant that despite repeated attempts and much agonizing we could only reduce this interface down to 150 functions, each of which we had to write a stub for.

We abandoned this effort after months of trying to get the stubs to work correctly. We mainly failed because TCP, like other large complex subsystems, has a large, complex, messy interface its host system. Writing a harness that perfectly replicates the *exact* semantics of this (often poorly documented) interface is difficult. In practice these stubs have a myriad of subtle corner-case mistakes. Since model checkers are tuned to find inconsistencies in corner cases, they will generate a steady stream of bugs, all false. To make matters worse, these false positives tend to be much harder to diagnose than those caused by static analysis. The latter typically require seconds or, rarely, several minutes to diagnose. In contrast, TCP's complexity meant that we could spend *days* trying to determine if an error report was true or false. One such example: an apparent TCP storage leak of a socket structure was actually caused by an incorrect stub implementation of the Linux timer model. The TCP implementation uses a function `mod_timer()` to modify the expiration time of a previously queued timer. This function's return value depends on whether the timer is pending when the function is called. Our initial stub always returned the same value. This mistake confused the reference counting mechanism of the socket structures in an obscure way, causing a memory leak, which took a lot of manual examination to unravel.

It is conceivable that with more work we could have eventually replicated all 150 functions in the interface to perfection (at least until their semantics changed). However, we did not seem to be reaching a fixed point — in fact, each additional false positive seemed harder to diagnose. In the end, it was easier to do the next approach.

5.1.2 Surprising success: run Linux in CMC

While it seems intuitive that one should cut across the smallest interface point, it is also intuitive that one cut along well-defined and documented interfaces. It turns out that for TCP (and we expect for complex code in general) the latter is a better bet. It greatly simplifies the environment modeling. Additionally, it makes it less likely that these interfaces will change in future revisions, allowing the same environment model to be (re)used as the system implementation evolves. While the approach may force the model checker to deal with larger states and state spaces, the benefits of a clean environment model seem to outweigh the potential disadvantage.

It turns out that for TCP there are only two very well-defined interfaces: (1) the system call interface that defines the interaction between user processes and the kernel and (2) the "hardware abstraction layer" that defines the interaction between the kernel and the architecture. Cutting the code at this level means that we wind up pulling the entire kernel into the model! While initially this sounded daunting, in practice it turned out to not be that difficult to "port" the kernel to CMC by providing a suitable hardware abstraction layer. This ease was in part because we could reuse a lot of work from the User Mode Linux (UML) [28]) project, which had to solve many of the same problems in its aim to run a working kernel as a user process.

In order to check the TCP implementation for protocol compliance, we wrote a TCP reference model based on the TCP RFC. CMC runs this alongside the implementation, providing it with the same inputs as to the implementation, and reports if their states are inconsistent as a protocol violation error.

5.2 The coverage problem: no execute, no bug

As with dynamic checking tools, model checking can only find errors on executed code paths. In practice it is actually quite difficult to exercise large amounts of code. This section measures how comprehensively we could check TCP.

We used two metrics to measure coverage. The first is line coverage of the implementation achieved during model checking. While the crudeness of this measure means it may not correspond to how well the system has been checked, it does effectively detect the parts that have *n*ot been tested. The second is "protocol coverage," which corresponds to the abstract protocol behaviors 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 abstract protocol transitions have been explored.

We used the two metrics to detect where we should make model checking more comprehensive. Low coverage often helped in pointing out errors in our environment model. Table 7 gives the coverage achieved with each step in the model refinement process. We measured coverage cumulatively using three search techniques: breadth-first, depth-first, and random. In random search, each generated state is given a

Description	Line Coverage	Protocol Coverage	Branching Factor	Bugs
Standard server and client	47.4 %	64.7 %	2.91	2
+ simultaneous connect	51.0 %	66.7 %	3.67	0
+ partial close	52.7 %	79.5 %	3.89	2
+ 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.

random priority. Table 7 also reports the branching factor of the state space as a measure of its size — lower branching factors are good, since they mean the state increases exponentially less each step in. 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 single TCP client communicating with a single 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. The second model adds multiple simultaneous connections, which are initiated nondeterministically. The third model lets either end of the connection nondeterministically decide to close it during data transfer. This improved coverage and resulted in the discovery of two more errors. Finally, much of the remaining untested functionality was code to handle bad packets, so we corrupted packets by nondeterministically toggling selected key control flags in the TCP packet. While these corrupted packets triggered a lot of recovery code 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.

In the end we 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.

6 Discussion

This section discusses some of the lessons we have learned after using static analysis on many large, real systems.

Several features of bug finding analysis were unexpected. First, when we started we thought (like many others) that it would be difficult to find bugs in a large working system and that detecting just a few would be a success. (In fact, we thought it was likely we would have to use historical data to demonstrate the approach rather than being able to find bugs in a "live" system.) This view was a dramatic overestimation of the work needed to find interesting bugs. If code has to obey a checked property more than a couple hundred times you will almost certainly find violations of it; if not the analysis is almost certainly broken. In practice, anytime we push a million lines of code through the checker and it does not find anything we immediately assume there is a bug in our system.

Second, the analysis needed to check a given property is often easier to write than the code needed to articulate how the property was violated. The latter requires tracking each analysis step done to detect the error and searching for the shortest number of such steps needed to cause the error. Perhaps unsurprisingly, saying *why* something happened often takes more work than saying what happened.

Finally, the analysis needed to check properties in actual code is often much simpler than one would need in the fully general case. In part this seems to come the fact that code dealing with the human-level properties we check ("`lock()` must be followed by a call to `unlock()`") cannot be arbitrarily complex but must be understood by a programmer. Code hard for analysis to understand is often hard for people to understand as well.

We look a several other issues in more detail below.

6.1 Ease-of-inspection really matters

A surprise for us from our static analysis work was just how important ease-of-inspection is: in many cases a hard-to-inspect error might as well have not been found since users will simply 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 [26]. 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.

6.2 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 [11]. As a result, the errors they found were almost certainly errors and were trivial to inspect. As we added more interprocedural support and simple aliasing 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.

6.3 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 Conclusion

This paper has described trade-offs between both static analysis and model checking, as well as some of the surprises we encountered while applying model checking to large software systems. Neither static analysis nor model checking are at the stage where one dominates the other. Model checking gets more properties, but static analysis hit more code; when they checked the same property static analysis won.

The main advantages static analysis has over model checking: (1) gets all paths in all code that can compile, rather than just executed paths in code you can run, (2) only requires a shallow understanding of code, (3) applies in hours rather than weeks, (4) easily checks millions of lines of code, rather than tens of thousands, (5) can find thousands of errors rather than tens. The first question you ask with static analysis is "how big is the code?" Nicely, bigger is actually better, since it lets you amortize the fixed cost of setting up checking. Model checking's first question is "what does the code do?" This is both because many program classes cannot be model checked and because doing so requires an intimate understanding of the code. Finally, given enough code we are surprised when static analysis gets no results, but less surprised if model checking does not (or if the attempt abandoned). Most of these are direct implications of the fact that model checking runs code and static analysis does not.

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 advantages that seem hard for static analysis to match: (1) it can check the implications of code, rather than just surface-visible properties, (2) it can do end-to-end checks (the routing table has no loops) rather than having to anticipate and craft checks for all ways that an error type can arise, (3) it gives much stronger correctness results — we would be surprised if code crashed after being model checked, whereas we are not surprised at all if it crashes after being statically checked.

A significant model checking drawback is the need to create a working, correct environment model. We had not realized just how difficult this would be for large code bases. In all cases it added weeks to months of effort compared to static analysis. Also, practicality forced omissions in model behavior, both deliberate and accidental. In both FLASH and AODV, unmodeled code (such as omitting the I/O system or multicast support) led to many false negatives. Finally, because the model must perfectly replicate real behavior, we had to fight with many (often quite-tricky-to-diagnose) false positives during development. In TCP this problem eventually forced us to resort to running the entire Linux kernel inside our model checker rather than creating a set of fake stubs to emulate TCP's interface to it. This was not anticipated.

All three model checking case studies reinforced the following four lessons:

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.

8 Acknowledgments

This paper recounts research done with others. In particular, we thank David Lie and Andy Chou for their discussions of lessons learned model checking the FLASH code (of which they did the bulk of the work) and David Park for his significant help developing CMC and model-checking TCP. We especially thank David Dill for his valuable discussions over the years. We thank Willem Visser for thoughtful comments on a previous version of this paper.

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.

References

[1] 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.

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

[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] 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.

[5] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235 – 244. ACM Press, 2002.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

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

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

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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