Model Checking System Software with CMC

Madanlal Musuvathi[*]
Stanford University
madan@cs.stanford.edu

Andy Chou
Stanford University

David L. Dill
Stanford University

Dawson Engler
Stanford University

Abstract

Complex systems have errors that involve mishandled corner cases in intricate sequences of events. Conventional testing techniques usually miss these errors. In recent years, formal verification techniques such as [5] have gained popularity in checking a property in all possible behaviors of a system. However, such techniques involve generating an abstract model of the system. Such an abstraction process is unreliable, difficult and miss a lot of implementation errors.

CMC is a framework for model checking a broad class of software written in the C programming language. CMC runs the software implementation directly without deriving an abstract model of the code. We used CMC to model check an existing implementation of AODV (Ad Hoc On Demand Distance Vector) routing protocol and found a total of 29 *bugs in two implementations [7],[6] of the protocol. One of them is a bug in the actual specification of the AODV protocol [3]. We also used CMC on the IP Fragmentation module in the Linux TCP/IPv4 stack and verified its correctness for up to* 4 *fragments per packet.*

1 Introduction

The reliability of system software is particularly important in areas such as communication protocols, security-sensitive applications, and embedded software. In these applications, failures may affect many machines, compromise the integrity or privacy of user data, subvert the ability of an administrator to control the system, or totally cripple the application. On the other hand, errors in these systems are difficult to find as they usually involve mishandled corner cases triggered by intricate sequences of events. Conventional testing techniques are not able to detect these errors for two key reasons. Firstly, traditional coverage techniques [1] do not guarantee coverage across all sequences of events. Secondly, the occurrence of systemic events such as packet loss, link failure or disk crashes are not easily controlled in a test environment. Thus, even a rigorously tested system contains a residue of errors that either cause the system to crash after long periods of execution, or present a potential security hole to a malicious user.

In recent years model checking [2] has gained popularity as an automatic verification technique that enumerates, either explicitly or implicitly, all possible states of a finite state system. Conventional model checkers [5] usually assume that the design is described at a high level that abstracts away many implementation details. Verifying actual code using such a tool requires reconstructing this abstract description from the code. This process requires a great deal of manual effort. Moreover, human errors in the manual abstraction result in missing bugs and causing false alarms during the verification process. A final, serious problem is that the models are usually written in a special-purpose description language that do not support the low level semantics of C, the preferred language for system implementations. As a result the models are often so abstract that there is a significant semantic gap between such a model and an implementation. For these reasons, it is a notable curiosity when software is model checked, rather than an everyday occurrence.

This paper describes CMC, a C Model Checker. CMC is a framework for model checking broad range of software implementations directly. Given an implementation of the system, a set of events that affect the system, and the event handlers in the implementation that handle each event, CMC explores the state space of the system by systematically executing all possible sequences of events. After each event execution, it calls the *pickle* function provided by the user to extract the state of the system and stores the state in a hashtable. By remembering the states visited, CMC explores each state of the system only once. The primary advantages of this approach follow from the ability to model check an implementation directly. No model of the code needs to be constructed, refined, or debugged. In addition, we can leverage existing dynamic techniques for debugging code, such as memory leak and stack overflow detectors. These tools have exhaustive coverage when run in the CMC framework.

To our knowledge, Verisoft[4] is the only other tool that is able to model check the implementation directly. However, Verisoft does not store states. As a result, states that have been explored before are redundantly checked, and systems can not be exhaustively verified if cycles exist in the state space. Interesting systems almost always have state spaces

[*]Partially supported by GSRC/MARCO Grant No: SA3276JB

with cycles because otherwise only a finite number of events (e.g. message sends) can be handled before stopping.

2 Design of CMC

CMC models the system being verified as one or more *processes*. These processes execute the native code of the implementation in the context of the model checker. Processes can be separate implementations or different instances of the same implementation (e.g. multiple nodes running the same routing protocol).

Each process has a private *internal state* and uses a shared *global state* to communicate with other processes. At any instant, the *system state* is the aggregate of the internal states of the processes and the shared global state. A *transition* of the system is defined as the execution of a deterministic, atomic procedure by a process that modifies its internal state and the global state. The set of transitions for a process are exposed to CMC via a standard interface.

Figure 1 shows a stripped-down implementation of a routing protocol. A process is a node running the protocol in a network. The internal state of each process is its IP address and its routing table. The routing table is simply a linked list containing the next hop for each destination IP address. Each process has two transitions, `recv_packet()` and `on_link_failure()`, which modify the route table in response to network events. The global state of the system is the state of the network, which is not shown in the figure. The system state is the aggregate of the internal states of all of the processes in the network and the global state.

CMC model checks software using explicit state enumeration. From each system state, it executes all transitions of all processes to determine the set of successor states. Applying this recursively from the initial state, CMC explores the entire set of reachable states of the system. CMC maintains a hash table of visited states and never explores a state more than once. We use hash compaction [10] to guarantee negligibly small probability of collisions. For each visited state, a set of invariants are checked to determine the correctness of the system.

CMC requires that processes provide two functions, *pickle* and *unpickle*. A pickle function gathers the internal state of a process, consisting of global variables and heap data, and produces a concise representation of the state. An unpickle function inverts the pickle function, taking the state and restoring the global variables and heap data. These two functions provide access to the internal state of the processes and make state space enumeration possible.

Given these two functions, CMC executes a transition as follows. It unpickles the current state of the process and executes the transition by running the implementation. This is the only time the implementation has control of the processor. When the transition function returns, CMC pickles the state of the process to determine the next system state. The current implementation of CMC requires that processes execute events "atomically" in the sense that they run to completion without blocking. This restriction also eliminates the state of the stack from a process' internal state.

In Figure 1, the `pickle()` function takes the IP address and route table entries for a process and copies them into a state buffer. The `unpickle()` function does the inverse, taking a state buffer and restoring the IP address and route table entries.

Real systems pose two key challenges. First, the state space might be infinite. However, the state space can be effectively pruned using techniques such as limiting the number of processes, constraining state variables to a subrange, and limiting the size of the heap. Even after applying these techniques, the state space might still be too large to complete search the entire space. We are currently investigating various heuristic search techniques to direct the search.

Second, internal state of a process may contain complex data structures, and implementing pickle and unpickle may be difficult. However, as these functions involve a traversal of the data structures, skeleton of such functions could be generated automatically using the type definitions. We are currently exploring this possibility. In our case studies (§3.1), we wrote the pickle and unpickle functions by hand. However, we used the accessor functions in the implementation itself, which greatly simplified the process.

CMC is particularly suited to systems where the behavior of interleaving executions of multiple processes gives rise to complex, emergent behavior. As long as the state space of the model is kept small enough, CMC can be very effective at detecting unexpected behaviors. On the other hand, using CMC requires a nontrivial amount of work, which makes it difficult to apply to large amounts of code. As described above, the system also imposes restrictions that may preclude its use on certain programs.

3 Model Checking Case Studies

In this section we discuss our experience with CMC on two case studies: AODV (Ad-hoc On-demand Distance Vector)[3] routing protocol, and the IP fragmentation module in the Linux kernel (Version 2.4.18).

3.1 Verifying AODV

AODV is a loop-free distance vector protocol for ad-hoc networks [3]. We applied CMC on two implementations of the protocol. The first implementation, *mad-hoc* [7], was released two years ago and has been under active development since. It runs as a user-space module and contains approximately 5500 lines of code. The second implementation, *Kernel AODV* [6], which descends from *mad-hoc* was released a

```
/* internal state of process */
IPAddress my_ip;
struct RouteEntry {
  IPAddress dest_ip;
  IPAddress next_hop;
  struct RouteEntry* next;
} *route_table;

/* transition functions */
recv_packet(buffer, len) {
  parse buffer;
  update route_table;
}
on_link_failure(neighbor) {
  for each entry in route_table
    if entry->next_hop == neighbor
      remove entry from route_table;
}

/* pickling code */
pickle(state){
  copy my_ip into state;
  for each entry in route_table
    copy dest_ip, next_hop into state;
}
unpickle(state){
  copy my_ip from state;
  for each entry in state
    insert entry into route_table
}
```

Figure 1. A skeleton routing protocol implementation.

bit less than a year ago. It contains 7500 lines of code and runs as a loadable kernel module in Linux and ARM based PDAs.

We modeled up to 4 AODV processes, each running an instance of the same implementation. The state of the process consisted of several global variables and the routing table, implemented as a link list. The pickle and unpickle functions traverse the entries in the routing table, using the accessor functions present in the implementation. The pickle and unpickle functions were straightforward to write and are 50 lines of code each. The AODV model, including the pickle and unpickle functions were shared between the two implementations, with minor modifications.

Table 1 summarizes the set of bugs found using CMC in both AODV implementations. The bugs range from simple memory errors to protocol invariant violations. We found a total of 29 unique bugs in the two implementations. The Kernel AODV implementation has 5 bugs (shown in parenthesis in the table) that are instances of the same bug in the mad-hoc implementation. The AODV specification bug is one of them, since both implement the same specification.

We describe the bugs below at a high level to give a feel for the breadth of coverage provided by CMC.

Memory errors. The first three error classes were various ways to mishandle dynamically allocated memory: not checking for allocation failure (10 errors), not freeing allocated memory (8 errors), or using memory after freeing it (2 errors). These were all detected by the built-in memory manager in CMC.

Both implementations carefully checked the pointer returned by *malloc* was not null. However, functions that call *malloc* indirectly can also return null pointers when the allocation fails. The code only erratically checked such cases. Since CMC directly executes the implementation, such errors were manifested in segmentation faults.

Most of the memory leaks were similarly caused by mishandled allocation failures. Commonly, code would attempt to do two memory allocations and, if the first allocation succeeded but the second failed, would return with an error, leaking the first pointer.

Unexpected messages. CMC detected two places where unexpected messages would cause mad-hoc to crash with a segmentation violation. This occured when the system lost its state, either due to a reboot, or due to a timeout between a request response cycle. When the system received the response in its altered state, it resulted in a segmentation violation. This a serious security violation as an attacker can maliciously send a bogus response.

Invalid messages. There were 5 cases of invalid packets being created, 3 cases of using uninitialized variables (these could not be detected by gcc -Wall), and 2 cases where invalid routes were used to send routing updates, violating the AODV specification; CMC also detected 2 instances of integer overflow which resulted in program assertion failures. Both implementations use an 8 bit integer to store the hop counts and use 255 to represent a hopcount of infinity. In the two error cases, an infinite hopcount was erroneously incremented to 0. This also accounted for a program assertion failure, as it resulted in an invalid routing table.

Routing loops. As AODV is a loop-free routing protocol, any routing loop produced during the execution of the implementation is a bug. We found three instances of routing loops, one of which is a bug in the AODV specification [3]. The two routing loop errors resulting due to implementation error are discussed here. In one, the implementation performs a sequence number comparison before a subsequent increment, while the AODV specification requires the comparison to be done after the increment: In the second case, the implementation fails to increment a sequence number while processing specific protocol message, viz. the RERR message of AODV.

The specification bug. This bug involved the handling of “route-error” (RERR) messages. In AODV, every route has a sequence number that determines the “freshness” of the route. AODV guarantees loop-freeness by appropriately manipulating these sequence numbers. When a node receives an RERR from its next hop, it sets the sequence number of its route to the sequence number in an RERR message. Under normal conditions this is the right thing to do. However, when the underlying link layer can reorder messages, the RERR message might have an outdated sequence num-

	mad-hoc AODV	Kernel AODV
Mishandling *malloc* failures	4	6
Memory Leaks	5	3
Use after free	1	1
Unexpected Message	2	0
Generating Invalid Packets	3	0 (2)
Program Assertion Failures	1	0 (1)
Routing Loops	2	1 (2)
Total	18	11 (16)

Table 1. Summary of bugs found in the two implementations of AODV

ber resulting in the node setting its sequence number to an older version. This can ultimately result in a routing loop. This bug was mentioned to the authors of the protocol with a suggested fix. Both the bug and the fix were accepted by the protocol authors[8].

The specification bug was found by running 4 AODV nodes using a depth-first search of the state space. CMC came up with an error trace of length 93. By using a best-first search we were able to find traces as short as 27. Performing a breadth-first search of the state space would give the shortest trace. However, breadth-first search on AODV ran out of resources without finding the bug. A careful hand crafted simulation of the bug required at least 25 transitions. An error of this complexity would be very difficult to catch using conventional testing means.

3.2 Verifying IP Fragmentation

We verified the IPv4 Fragmentation[9] module in the Linux Kernel (Version $2.4.18$). This module assembles all fragments of an IP packet before sending it to the higher layer. This module, along with the *skbuff* library it uses, contains 1850 lines of code. In order for these kernel modules to run in user space, we provided stub functions, most of which were automatically generated. There are 21 stub functions with a total of 150 lines of code.

We verified the IP Fragmentation module with all sequences of 4 or less fragments of an IP packet. We did not find any bugs in this module. We did find a known bug in a previous version of this module, though. This case study is a proof of concept that its possible to model check kernel code using CMC.

4 Future Work

CMC is still in its preliminary stage and the results we have achieved till now are encouraging. We believe that CMC will be applicable to a wide variety of implementations. Currently we are looking into verifying properties in TCP stack implementations, filesystems, kernel schedulers, and security-sensitive applications such as root programs.

We are also looking into techniques to automate some of the steps needed to use CMC with an existing implementation. For example, compiler support might be used to help the user write the pickle and unpickle functions. Finally, we are evaluating different heuristic search techniques to guide our model checking.

5 Acknowledgments

We like to thank Satyaki Das and David Park for the numerous discussions on this topic. We also thank David Park for providing the implementation of hash compaction. We also would like to thank the blind reviewers who provided valuable suggestions to the first draft of this paper.

References

[1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

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

[3] 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.

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

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

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

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

[8] Charles E. Perkins, Elizabeth M. Royer, and Samir R. Das. Private Email Communication.

[9] J. Postel. Internet Protocol. RFC 791, USC/Information Sciences Institute, September 1981.

[10] 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.