USENIX Association

# Proceedings of the First Symposium on Networked Systems Design and Implementation

San Francisco, CA, USA
March 29–31, 2004

# Model Checking Large Network Protocol Implementations

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

## Abstract

Network protocols must work. The effects of protocol specification or implementation errors range from reduced performance, to security breaches, to bringing down entire networks. However, network protocols are difficult to test due to the exponential size of the state space they define. Ideally, a protocol implementation must be validated against all possible events (packet arrivals, packet losses, timeouts, etc.) in all possible protocol states. Conventional means of testing can explore only a minute fraction of these possible combinations.

This paper focuses on how to effectively find errors in large network protocol implementations using model checking, a formal verification technique. Model checking involves a systematic exploration of the possible states of a system, and is well-suited to finding intricate errors lurking deep in exponential state spaces. Its primary limitation has been the effort needed to use it on software. The primary contribution of this paper are novel techniques that allow us to model check complex, real-world, well-tested protocol implementations with reasonable effort. We have implemented these techniques in CMC, a C model checker [30] and applied the result to the Linux TCP/IP implementation, finding four errors in the protocol implementation.

## 1 Introduction

Network protocols must work. The current state-of-practice for automatically ensuring they do are various forms of testing — using a network simulator, doing end-to-end tests on a live system, or as an interesting twist, analyzing their traces to find anomalies. The great strength of such testing approaches is that they are easily understood and give an effective, lightweight way to check that the common case of an implementation works. Unfortunately, protocols define an explosively large state space. Implementors must carefully handle all possible events (lost, reordered, duplicated packets) in all possible protocol and network states (with one packet in flight, with two, with a just-wrapped sequence number). It is only possible to test a minute fraction of the exponential number of such combinations. Thus, just at the moment implementors need validation the most, testing works the least well. As a result, even heavily-tested systems can have a residue of errors that take days or even weeks to arise, making them all but impossible to replicate.

When applicable, formal verification methods can find such deep errors [26, 32, 37]. One approach involves an *explicit state* model checker that starts from an initial state and recursively generates successive system states by executing the nondeterministic events of the system. States are stored in a hash table to ensure that each state is explored at most once. This process continues either until the whole state space is explored, or until the model checker runs out of resources. When it works, this style of state graph exploration can achieve the effect of impractically massive testing by avoiding the redundancy that would occur in conventional testing.

Most conventional model checkers, however, require that an abstract specification (commonly referred to as the "model") of the system be provided. This upfront cost has traditionally made model checking completely impractical for large systems. A sufficiently detailed model can be as large as the checked system. Thus, implementors often refuse to write them, those that are written have errors and, even if they do not, they "drift" as the implementation is modified but the model is not.

Recent work has developed model checkers that work with the implementation code directly with-

[*]Supported by GSRC/MARCO Grant No:SA3276JB

out requiring an abstract specification. In prior work, we developed an implementation-level model checker, CMC [30], and used it to check three different implementations of the AODV protocol (roughly 6K lines of code each). Model checking AODV involved extracting the AODV processing code from the implementation and running it along with an input generating test harness. Using this approach, CMC found errors every few hundred lines of code, as well as an error in the underlying AODV protocol specification [12].

This paper is about how to scale model checking up to protocols a factor of ten larger. After the success checking AODV we decided to check the hardest thing we could think of: the Linux kernel's widely-used (and thus extremely thoroughly tested and visually inspected) implementation of TCP. The particular implementation we used (version 2.4.19) is roughly 50K lines of code.

Scaling CMC to such a large system involved some unusual challenges. First and foremost was the "unit-testing" problem — model checking TCP requires running the kernel implementation as a closed system in the context of CMC. However, extracting large pieces of code from a host system not designed for unit testing is much, much harder than it may seem. Any procedure this code calls must be reimplemented in the model checker; real code has a startling number of such procedures (the narrowest interface we could cut along in TCP had over 150 procedures). Worse, such procedures too-often have unspecified interfaces, making it easy to get their functionality slightly wrong. Model checkers are excellent at finding slightly wrong code, and will happily detect the resulting bogus effects, requiring a laborious tracking back to the source of these false errors. In many cases, this backtracking took days.

To avoid these problems, this paper presents an unusual approach; instead of extracting the TCP implementation, we run the *entire* Linux kernel in CMC. To trigger TCP related behaviors, the system contains an environment that interacts with the kernel through well-defined interfaces, such as the system call interface and the hardware abstraction layer. The semantics of these interfaces are well understood and thus, easy to implement correctly. The execution of the TCP code in the model checker exactly mirrors the execution in the kernel, thereby reducing false errors.

Running the entire kernel in CMC required us to heavily optimize the model checker. The system consists of *two* kernels communicating with each other as TCP peers, and the size of the system state is over two hundred kilobytes. This paper describes techniques that enable CMC to scale to such a large system and validates them by applying them to the Linux TCP implementation, where we find four errors. We believe that the approach we took and the techniques we used are useful (and perhaps necessary) to model check real code of any size.

This paper makes the following contributions:

1. It develops novel techniques to check code *in situ* rather than requiring it to be extracted from the system itself.
2. It develops ways for saving and restoring state to be automatic, yet efficient, and for superficial differences at the bit-level to be eliminated.
3. To the best of our knowledge, it is the first paper to check software as complex as TCP; the closest other efforts are an order of magnitude smaller.
4. It demonstrates that the approach can find real error in heavily inspected and tested, complex code.
5. It provides a generic framework for testing other protocols with much lower effort than any other model checker.

## 2 CMC Overview

CMC is an explicit model checker that directly executes a given protocol implementation. One way to understand the working of CMC is to consider it as a *backtracking* network simulator. Like any network simulator [25], CMC runs a protocol description along with a suitable environment that consists of a network model and a user model. Instead of using a simplified protocol description, CMC executes an actual implementation of the protocol (along the lines of [7]). As in a network simulator, CMC enables protocol behavior by triggering the events in the environment, such as network interrupts, timeouts, and user inputs.

However, the key difference between CMC and a network simulator is that CMC checkpoints the system state at specific points. This provides two important capabilities. First, CMC can backtrack to a previous state and explore a different sequence of events. In contrast, a network simulator is restricted to exploring a single event sequence determined by the initial random seed provided to the simulator. Second, CMC stores a signature of each checkpointed state in a hash table. By doing so, CMC avoids redundantly exploring a system state more than once. This is particularly helpful when exploring all event interleavings in the system.

CMC is implemented as a library that links with the C (or C++) implementation of the network protocol. CMC models a given system as one or more interacting *processes*. Each process can have one or more threads. A CMC process behaves in many respects like a normal operating system process; each process has its own copy of global variables, heap, and stacks for each of its threads. Processes do not share state, but communicate with each other through a region of shared memory. CMC along with all the process in the system run as a *single* operating system process. Internally, each process is implemented as one or more user space threads.

The notions of processes and threads provide a straightforward way to model network protocols. Each process models a node in the network and executes the implementation of the protocol. If the implementation is multithreaded (as is the case with the Linux TCP implementation), a CMC process can allocate a thread for each threads of execution in the implementation. The processes can communicate with each other using a network modeled using the shared memory region.

A user has to perform the following tasks before a protocol can be model checked. First, the protocol implementation should run as a closed system in a CMC process. Typically, this involves extracting the protocol specific parts from an implementation and providing *stubs* or support functions to close the system. This task is necessary for unit-testing the protocol implementation, or to execute the implementation *as is* in a network simulator [7].

Second, the user specifies the *nondeterminism* in the system. This allows CMC to explore multiple executions from a single state. In most cases, this involves calling special functions in the CMC library. For instance, CMC provides a `send()` function that nondeterministically drops any packet sent to the network. As a result, CMC explores two possible executions for every packet; one in which the packet is sent to the network, and the other in which the packet is dropped.

Third, the user provides some correctness properties about the protocol. The user specifies these properties as invariants implemented as boolean functions (in C), which CMC evaluates at each state during state space exploration. For instance, a user can provide a function that checks if a TCP implementation reduces the congestion window in response to a packet loss.

Given an appropriate system that includes the protocol implementation, CMC systematically enumerates the possible states of the system. The system state includes the state of all processes and the state of the shared memory region. The state of process consists of the contents of all the global variables, the heap, and the stacks (and register contents) of all the threads in the process.

CMC starts from the initial state of the system, and recursively generates all its successors. From each state, CMC executes a *transition* to generate a successor state. A transition roughly corresponds to the handling of a protocol event, such a packet reception or a timeout, by some thread of a process in the system. Due to nondeterminism in the system, there can be more than one transition possible from a state, each potentially leading to a different successor state.

The state space is prohibitively large for most nontrivial systems. As a result, it is impossible to enumerate *all* possible states of a given system with limited resources. While not able to provide absolute guarantees about a given protocol implementation, CMC focuses on exploring as many different protocol behaviors as possible before running out of resources (§5).

## 3 The Model Checking Framework

The first step in model checking a protocol is to run the protocol implementation as a process in CMC. In our case, this requires that the Linux kernel implementation run as a closed system in user space.

### 3.1 Why the Conventional Approach Fails

In our first attempt, we followed the conventional wisdom: extract the TCP related code from the kernel along the narrowest possible interface; and run it with a *kernel library* that provides stubs for all the kernel services the TCP code requires. Choosing a narrow interface keeps the library simple. This approach also has the advantage of minimizing the state size of the model, as the kernel library can be optimized by removing any redundant states.

Starting from the core set of TCP modules, we conservatively added a few tightly coupled modules (such as IP) to the model. To close the system, we then manually provided stub implementations for all the interface functions in the system boundary.

However, providing correct implementations for these interface functions proved to be an extremely difficult task. The TCP code interacts with the rest of the kernel along complex and undocumented interfaces. Our initial version of the kernel library involved around 150 interface functions. Providing stub implementations and understanding the subtle interactions between various interface functions required considerable understanding of the different kernel modules. More often than not, our stubs were buggy.

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 module. 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. However, 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.)

During our initial runs, we progressively fixed bugs in our implementations as we found them. After spending months, we gave up. 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.

## 3.2 Running the Entire Linux Kernel

The hard learned lesson from our previous approach is that instead of choosing a *narrow* interface, the model should involve *well-defined* interfaces. For the Linux kernel, there are only two such interfaces: the system call interface that defines the interface between the kernel and the user processes; and the "hardware abstraction layer" that defines the interface between the kernel and the hardware architecture. Though Linux does not explicitly define a hardware abstraction interface, such an interface is implicitly defined for most kernels to simplify the task of porting the kernel to different architectures.

Defining the TCP model along these two interfaces requires that the *entire* kernel is run in user space as a CMC process. While this might look like a daunting task, we heavily reused the user space implementation of the Linux kernel from [38]. This still requires CMC to deal with the entire kernel state which is orders of magnitude larger than the TCP relevant state alone. Section 4 describes techniques by which CMC in effect automatically extracts the TCP relevant state from the kernel state.

Using the system call interface and the hardware abstraction interface has another advantage. These interfaces change very rarely during future revisions. Thus, the effort required in building a TCP model can be reused across multiple versions of the kernel.

## 3.3 The TCP Model

Once the TCP implementation can run in user space, the next step involves constructing the actual CMC model: allocating one or more CMC processes to run the TCP implementation and designing an environment to appropriately trigger the implementation.

In the kernel, the TCP code executes in three contexts: in user context when a user process makes a system call, in the context of a network interrupt handler when a TCP packet is received, and in the context of a timer interrupt handler when a TCP timer fires. To mimic this behavior a CMC process running the Linux TCP implementation contains the following three threads:

- An *application* thread that makes socket related system calls to the kernel. Of the two application threads one behaves as a standard TCP server, while the other behaves as a standard TCP client.
- A *network* thread that emulates a packet interrupt and executes the code to handle packet reception.
- A *timer* thread that emulates a timer interrupt and fires one of the pending timer routines.

These threads once triggered can either execute to completion or can block. These threads block by *yielding* control to the kernel scheduler. In the real kernel, the scheduler would then execute the scheduling algorithm to determine the thread that

|  | **Size of a System State (in KB)** | **Average Change in a Transition (in KB)** |
|---|---|---|
| Global Variables | 78.28 | 11.37 |
| Heap (average) | 25.06 | 2.13 |
| Stack | 24.00 | 4.00 |
| Total per Process | 127.34 | 17.50 |
| Network State | 1.00 | 1.00 |
| System State (Sum of two process states and a network state) | 255.68 | 18.50 |

Table 1: The state size for the TCP model described in Section 3.3. The second column shows the amount that changes in a transition, on average. For the global variables and the stack, CMC can only detect changes at page size (4KB) granularities. For the heap, CMC detects changes at individual object granularities.

is scheduled next. In our model, the scheduler is modified to immediately transfer control to CMC. This enables CMC to *nondeterministically* choose the thread that is executed next, and explore multiple thread schedules from a given state.

Similarly, the kernel timer routine is modified to allow CMC to choose the timer that fires next when multiple timers are enabled. Optionally, CMC can fire timers out of order irrespective of their expiration times. While this can lead to some behaviors not possible in a real implementation, it has the benefit of making the implementation behavior independent of the actual values of the timers.

The two kernels communicate through a network, modeled as a list of messages in the shared memory. The network model can lose, duplicate, reorder and corrupt messages. Each kernel communicates with the network through an appropriate network device driver.

## 4 Handling Large States

Protocol implementations can have large states. For the TCP model discussed in Section 3.3, each state is around 250 kilobytes, which is shown in Table 1. Note that a process in the TCP model runs the entire Linux kernel. Thus, the process state consists of all the global variables in the kernel (78KB), and any memory dynamically allocated during the kernel boot-up and the subsequent processing of TCP events (25KB). Additionally, the process runs three kernel threads (§3.3), each of which run in a separate 8KB stack. [1] The system consists of two such processes along with a model of the network, resulting in a state that is 250KB in size.

### 4.1 Managing Memory Resources

During the state space search, CMC maintains two data structures: a hash table of states seen during the search, and a queue of states seen but whose successors are yet to be generated. It is not necessary to store the entire state in a hash table [36]. CMC uses a hash compaction algorithm [36] to store only a small signature (typically, 8 bytes) for each state in the hash table. By doing so, the memory requirements of the hash table depend only on the number of states explored during model checking.

The queue, however, has to maintain the states in their entirety, as all of the information in the state is necessary to generate the successor states. However, the queue has good locality of reference, so much of it can be swapped to disk during model checking. Moreover, the states in the queue can be efficiently compressed; as states can simply be regenerated by remembering the sequence of events that generated them from the initial system state.

In practice, the large amount of memory available in modern machines makes managing memory much easier. For instance, a gigabyte hash table is more than sufficient to explore 100 million states when using a 8 byte compacted signature for each state. The remaining memory can be allocated for the queue. As will be shown below, CMC is limited more by the time required for the state space exploration than by the memory available.

[1] This stack size is hard-coded in the Linux kernel implementation.

| | Time in microseconds | | | | |
|---|---|---|---|---|---|
| | **State Restore** | **Transition Execution** | **Hash Computation** | **State Store** | **Total** |
| Processing Entire States | 656 | 305 | 17816 | 608 | 19385 |
| With Incremental States | 298 | 363 | 2927 | 64 | 3652 |
| With Incremental Heap Canonicalization | 305 | 365 | 1453 | 65 | 2188 |

Table 2: Time taken for a single CMC transition, averaged over the first million transitions. The experiment involves CMC checking the TCP model in a server running a 800MHz Intel Pentium III processor with 256KB cache and 2GB of memory.

## 4.2 Time Taken for a Transition

The basic step of CMC consists of a transition, which generates a single successor from a particular state. A transition consists of the following steps.

1. **State Restoring:** First, CMC restores the system to the desired start state of the transition.

2. **Executing the Transition:** After restoring the state, CMC transfers control to one of the enabled threads in the system. The thread executes the TCP code to process a specific input event such as a packet reception or a timeout.

3. **Computing the Hash:** When the thread yields control back to CMC, the implementation state, as modified by the thread, represents the successor state of the transition. To store this state in the hash table, CMC computes a signature and a hash value (that determines the location of the signature in the hash table) for the state. Additionally, CMC might perform state transformations (§5.1) before this hash computation.

4. **State Storing:** If the successor state is not present in the hash table, CMC queues a copy of the successor state for further exploration.

Thus, each transition requires at least three traversals of the state contents. When the state is hundreds of kilobytes, naively performing these traversals leads to a poor memory cache performance, considerably slowing the model checker. Table 2 shows the time taken for a transition when CMC processes the entire state contents during the transition. In this case, each transition takes around 20 milliseconds. At this rate, CMC can run for weeks without running out of memory. (A gigabyte hash table can easily store 100 million states, and for the TCP model, only one in five transitions generate a new state.)

From Table 2, we can see that the hash computation is the most expensive step in a transition; CMC spends more than 90% of its time computing the signature and the hash value of the state. While saving and restoring states involve simple memory copies, hash computation requires performing a few arithmetic operations for *every* byte of the state.

## 4.3 Incremental State Processing

It is possible to reduce the hash computation overhead by using a simple, but crucial observation. Even though the state is large, only small portions of it change in a transition, as shown in Table 1. There are a couple of reasons for this behavior. As the environment triggers only TCP specific events, parts of the kernel not relevant to TCP processing do not change during model checking. Additionally, a transition involves a specific event and thus, only involves data structures related to the processing of that event.

By processing only the incremental differences between states, CMC can considerably reduce the time taken for a transition. The basic idea is to subdivide an entire state into a set of smaller *objects*. CMC computes the hash value and the signature for each object separately and caches these values with the object. CMC identifies the objects that are modified in a transition, and only needs to process these objects during hash computations. The cached values can be used for objects not modified in a transition.

To determine the objects modified in a transition, CMC uses the virtual memory protection allowed by the underlying operating system. This scheme is similar to that used by distributed shared memory systems (such as Treadmarks [23]). The entire state is subdivided into a set of virtual memory pages. Before a transition, CMC restores the system to the desired start state and write protects all the pages in the state. During the transition, a first write to a protected page generates an access violation signal. CMC traps this signal, marks the page as dirty, and creates a copy of the page. This copy represents the state of the page in the start state of the transition. Once a page is dirty, CMC ensures that subsequent writes to the page do not involve an expensive signal delivery by removing the write protection on the page.

Table 2 shows the performance improvement in this incremental scheme. The cost of hash computation reduces by more than a factor of 5, and the transition takes less than 4 milliseconds on average. There is also an improvement in the state store and restore phase, as CMC only needs to copy pages that differ when switching between states. As expected, the time for actually running the implementation code increases due to the overhead of the access violation signals.

Section 5.1 describes another mechanism to further reduce the hash computation overhead.

## 5 Handling State Space Explosion

All model checkers have to handle the *state explosion* problem, which refers to the unmanageable size of state spaces even for moderately sized systems. As CMC deals directly with protocol implementations rather than their abstract models, CMC confronts much larger state spaces than conventional model checkers.

CMC tackles the state space explosion problem by following a best-efforts approach. CMC does not guarantee a complete search of the state space, but attempts to explore as many protocol behaviors as possible before running out of resources. No current approach can practically verify protocols of any complexity, so we instead focus on exercising the protocol as throughly as possible.

There are two ways in which CMC confronts the state explosion problem. First, CMC (like all model checkers) uses state transformations to recognize states that are superficially different at the bit level but are actually the same (or similar enough) at the semantic level. These transformations enable CMC to check only one out of a large (and exponential) set of equivalent states. Second, CMC uses heuristics to selective explore interesting portions of the state space.

This section describes two techniques that are an improvement of those previously described in [30]. Scaling CMC to TCP required that these techniques be automated and made more efficient.

### 5.1 Incremental Heap Canonicalization

One problem in model checking software programs is to handle heap canonicalization [22]. When objects are dynamically allocated in the heap, different allocation orders can result in heap states that are equivalent but differ in the memory locations of the objects. For instance, consider two states in which the TCP implementation receives the same two data packets in order and out of order respectively. The socket buffers will be allocated at different memory locations in the two states but queued in the sequence number order in both states. Thus, the two states differ in the bit level but are semantically the same. CMC should identify such states and avoid exploring them redundantly.

Equivalent heap states can be identified by transforming a heap state to a canonical representation shared by all equivalent heaps. Informally, the objects in the heap along with their pointers form a heap graph. Equivalent heaps have the same underlying graph structure. A canonicalization algorithm works by relocating each object to its canonical location and modifying the contents of all pointers to the object to reflect its new location.

The previously known algorithm [22] does not scale to large heaps. This algorithm requires processing large portions of the heap, which can reduce the speed of state space exploration (§4.2). Specifically, it performs a depth first traversal of the heap graph, and the canonical location of each object depends on its depth first ordering. Even small changes to the heap, such as an object allocation or a deletion, can change the depth first ordering for a large number of objects. This forces CMC to traverse and compute the hash for large portions of the heap, as shown in Table 3. During model checking, the heap, on average, consists of 103 objects that total 25KB. Note that the heap also includes non-TCP related data structures allocated by the Linux kernel. A TCP transition, on an average modifies 5 of these objects. However, this change requires CMC to recompute the hash value of almost half the objects in the heap.

Our contribution is an improved, *incremental* heap canonicalization algorithm. We briefly describe this algorithm; interested readers can refer [28] for more details. The incremental algorithm generates the

|  | **Number of Objects** | **Total Length in KB** |
|---|---|---|
| Objects in the entire heap | 102.7 | 25.06 |
| Objects modified in a transition | 5.1 | 2.14 |
| Objects accessed during heap canon. | 45.8 | 11.91 |
| Objects accessed during incremental heap canon. | 5.2 | 2.17 |

Table 3: Objects accessed during heap canonicalization. Every transition on an average modifies 5 heap objects. The incremental canonicalization algorithm requires traversing only these objects in most cases.

canonical location of a heap object from the *shortest path* of the object to some global variable in the heap graph. When a transition makes small changes to the heap structure, the shortest path of most objects is likely to remain the same [19, 31]. After a transition, CMC recomputes the hash only for those objects whose shortest paths have changed in a transition. This algorithm works well in practice, as shown in Table 3; in most cases CMC traverses only the objects that are modified in a transition. This improves the performance of CMC by 40% as indicated in Table 2.

## 5.2 Exploring Interesting Protocol Behaviors

Since CMC cannot exhaustively explore the state space of real protocols it takes a different approach and instead tries to explore the "most interesting" portions of them. It does so by attempting to focus on states that are the most different from previously explored states. The intuition for this is that the more different a state is from previous states the more likely it is to have new behaviors and, as a result, bugs. We describe two techniques CMC uses below.

The first uses the abstract protocol state to find states that will generate new behaviors. Conceptually, a protocol implementation state can be viewed at two levels. There is the concrete state, which is the heap, stack, global variables and registers — all the memory values that define the current computation. There is also the abstract, protocol state encoded in the concrete state — for example, whether TCP is in the LISTEN or CLOSED state. Many different concrete states may in fact define the same abstract state. We want to focus on concrete states that correspond to new abstract protocol states since they will generate new behaviors.

For TCP, the state of the reference model (§6) corresponds to the the abstract state of the protocol. We can thus use it to preferentially explore system states whose corresponding reference model state has not been seen before. We further tune this process by doing a series of *symmetry reductions* [10] on the abstract state, which allow us to determine when two superficially different abstract states will in fact generate the same behavior. The simplest example is sequence number standardization where, a state with all sequence numbers at 256 can be considered equivalent to a state with all sequences numbers at 512 (which can be reached, for instance, from the first state after successful data transfer of 256 bytes). While performing such reductions on the concrete state is difficult the small size of the abstract state ($10 - 20$ bytes) makes it relatively easy.

The second technique also explores states that appear to be different from previous ones. It uses the heuristic that on balance, a change in a variable that never or rarely changed before is more interesting than in one that changes often. CMC tracks the values of all variables in the states generated. If a particular variable (as determined by its byte location in the state) takes on more than a threshold number of distinct values, CMC eliminates those variables from subsequent hash computations. Different CMC runs can use different threshold values (including infinity) to guard against cutting off searches too soon.

A simple example of how this helps are the many counters and and statistics variables present in protocol implementations that do not affect the execution of the protocol, but whose changed values cause unnecessary blowup of the state space. Ideally, a user would identify such variables and provide them to CMC as annotations. However, providing such manual annotations for the entire TCP implementation (and in our case, the entire Linux Kernel) is impractical. CMC's variable pruning will automatically weed out such counter variables.

## 6 Specifying Correctness Properties

During the state space exploration, CMC automatically checks for certain generic properties such as memory leaks and invalid memory accesses. Also, CMC reports any deadlock states, in which the system can make no progress. To check for protocol-specific properties, the user has to provide additional invariants (written in C). CMC evaluates these invariants in every state it generates.

There are two aspects to the correctness of a network protocol. First, the protocol specification should be correct. Second, the particular implementation should implement the specification correctly. By running the implementation, CMC can simultaneously check for both specification and implementation errors. Any specification error will be promptly represented in the implementation. Given the maturity of the TCP specification, however, it is quite unlikely the specification contains errors and our emphasis has been on detecting implementation errors.

### 6.1 Checking Protocol Conformance

Checking that a TCP implementation conforms to the protocol specification is a challenge. The specification itself [35, 6] is large and complex. Moreover, the TCP specification is ambiguous at many places, leaving room for the implementations to make their own choices.

Along the lines of [33], we check for protocol conformance by ensuring that every transition of the implementation is allowed by a TCP reference model. This reference model consists of the basic state machine transitions literally translated from [35] to C, and is around 500 lines of code. During model checking, CMC provides the same set of input events (system calls, network and timer interrupts) to both the implementation and the reference model, and expects their states and the outputs (network packets and system call return values) to be consistent.

All inconsistencies between the TCP implementation and the reference model are not necessarily errors. They can arise due to the ambiguities in the TCP specification, known errors in the protocol [34], and manual errors in the reference model. We iteratively modified our reference model when we found such false error reports.

### 6.2 Checking for Resource Leaks

A TCP implementation should release all kernel resources at the end of the connection. Failure to do so can result in resource lockup, which subsequently reduces the performance and the availability of the machine. The resource leaks are not necessarily memory leaks, as these resources can still have valid references to them. To check for such resource leaks, CMC requires that the entire kernel eventually reach the initial state after completing a TCP connection. CMC reports any violation as a potential resource leak.

There are, however, valid situations when the kernel might *not* reach the initial state. First, some resources can be *cached* either by the TCP implementation or by the Linux kernel. To account for this fact, CMC traces through a few complete TCP connections to generate a state in which the resources are *already* cached. CMC performs the state space exploration from this state. Second, the initial and final states of a TCP connection can still differ due to the various statistics the protocol maintains. During the initial model checking iterations, CMC progressively learns to factor out these variables from the state (after simple manual inspection).

### 6.3 Checking Implementation Robustness

A TCP implementation should be robust against malformed packets sent by a misbehaving peer. Ideally, we would like to send all possible packets at each implementation state and ensure that the implementation handles them satisfactorily. However, this requires trying a prohibitively large number of transitions (exponential in the size of the packet) at *every* state. Also, most of the packets thus generated will be invalid TCP packets that will be dropped after a few simple checks.

We overcome this problem by generating well-formed TCP packets that are only slightly different from normal packets expected in a particular state. The system consists of two (well-behaved) TCP implementations communicating over an adversarial network. Apart from losing, reordering and duplicating packets, the network can slightly mutate a packet when it is received by the implementation. Mutating a packet includes toggling the control bits in the packet and pruning the packet data to generate very small packets. After performing the mutation, the network ensures that the packet is still well-formed by recomputing the checksum and appropriately modifying the sequence numbers whenever it toggles the SYN and FIN control bits in the packet.

## 7 Results

The effectiveness of CMC can be measured using two metrics. One is the number of bugs CMC is able to find. The second measure is how well the given system is tested. This section describes our results of model checking the Linux TCP implementation using these two metrics.

### 7.1 Bugs Found

CMC found four instances where the Linux implementation fails to meet the TCP specification. These errors amply reflect the kind of corner cases CMC is able to test.

The first bug CMC found involves the processing of RST packets. The Linux implementation fails to honor a RST packet in SYN_RCVD state unless the ACK bit is also set. The TCP specification requires that in response to a RST packet an implementation free any resources held by the connection and gracefully inform the application. Figure 1 shows the code containing this bug. The function `tcp_check_req` processes incoming packets in the SYN_RCVD state. The bug was inadvertently introduced while trying to handle a case of a maliciously generated packet. The fix causes the function to prematurely exit before processing the RST flag. This bug can potentially lead to lockup of kernel resources long after the TCP connection is dead.

The second bug involves incorrect handling of a duplicate SYN_ACK packet. While the specification requires that any duplicate packets be ignored, the Linux implementation fails to do so and faithfully uses the acknowledgment to open its congestion window. The error happens because of an incorrect sequence number check in the ESTABLISHED state.

CMC also found that the implementation fails to implement one transition in the TCP state diagram. If an application prematurely closes in SYN_RCVD state, the specification requires the implementation to gracefully close the connection using the $FIN$ handshake. An acceptable alternative is to perform an abnormal close by sending a RST packet. CMC found that in this case, the Linux implementation dropped the connection without notifying its peer.

```
// net/ipv4/tcp_minisocks.c
// Process an incoming packet for SYN_RECV
// sockets represented as an open_request.
struct sock *tcp_check_req(...) {
  ...
  /* You would think that SYN crossing is
     impossible here, since we should have
     a SYN_SENT socket (from connect()) on
     our end, but this is not true if the
     crossed SYNs were sent to both ends by
     a malicious third party. We must defend
     against this,
     ...
     Note: This case is both harmless, and
     rare.  Possibility is about the same
     as us discovering intelligent life on
     another plant tomorrow.
    ...
   */
  if (!(flg & TCP_FLAG_ACK))
          return NULL;

  /// BUG: Should have checked the RST field
  ///      before checking the ACK field

  // Invalid ACK: reset will be sent by listening
  // socket
  if (TCP_SKB_CB(skb)->ack_seq != req->snt_isn+1)
          return sk;
  ...
  //  RFC793: "second check the RST bit" and
  //         "fourth, check the SYN bit"
  if(flg & (TCP_FLAG_RST|TCP_FLAG_SYN))
     goto embryonic_reset;
  ...
}
```

Figure 1: The Linux TCP implementation (version 2.4.19) does not honor the RST flag in the SYN_RECV state unless the ACK bit is set. The bug was inadvertently introduced while fixing a "harmless and rare" case. The code prematurely returns when the ACK field is not set on an incoming packet.

|  | Model 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 4: Coverage achieved during model refinement. The branching factor is a measure of the state space size.

On examining the source later, we found a comment that acknowledges the incorrect handling of this case.

The final "bug" involves a subtle processing of ACK packets. A TCP implementation is required to abort a connection by sending a RST packet when it receives data that cannot be transferred to the application. This can happen, for instance, when the application closes the connection before the data transfer is complete. The bug involves the case when the implementation receives a packet containing both data and an ACK after the application has closed the connection. The Linux implementation blindly processes the ACK field before processing the data. If the ACK opens the congestion window, the implementation exhibits a peculiar behavior. It sends a stream of data packets and immediately follows with a RST packet aborting the connection. While we are not sure if we should count this as a *bug*, we found this behavior interesting to report.

CMC also detected one instance where the TCP specification might be ambiguous. This concerns the transmission of a FIN packet on a zero window. When the receiver advertises a zero window, the Linux implementation queues a FIN packet, even if no *data* is queued. This is definitely the behavior expected by the TCP specification as the sequence number of FIN lies outside the send window. However, it seems that an acceptable solution is to send the FIN packet as zero window probe.

## 7.2 Coverage Metrics

While one measure of the effectiveness of a tool like CMC is the number of bugs it finds, another measure is the extent to which a given implementation is tested. We resort to two coverage metrics which are described below. As CMC deals with tremendously large state spaces, CMC typically runs out of resources before completing the search. The coverage metrics are very useful in evaluating the different state space reduction techniques employed by CMC as well as the various models provided by the user.

While various coverage metrics have been studied in software testing [2], we use two metrics that are easy and straightforward to compute. 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.

Table 4 describes the coverage achieved while checking four iteratively refined models. Apart from the two coverage metrics, the table reports the branching factor of the state space as a measure of its (exponential) size. The branching factor is calculated from the number of states seen at a particular depth from the initial state.

The first model described in Table 4 consists of a standard TCP client connecting to a standard server and performing bidirectional data transfer before closing the connection. In the second model, the server nondeterministically decides to initiate the connection, thereby exploring the simultaneous connect mechanism. The third model refines the second model by allowing the client and the server to nondeterministically close the connection before the data transfer is complete. The fourth model introduces an adversary to mutate packets in the network. This tremendously improves coverage at the cost of an increase in state space size. These refinements were made iteratively after investigating the parts of the protocol not covered in a previous model.

The combined coverage in Table 4 reports the coverage achieved cumulatively over the four models. CMC achieves a combined protocol coverage of 92.1%, which represents almost complete coverage of the properties being checked by the TCP reference model. The uncovered lines consist of error condition checks that should not be triggered in a correctly functioning protocol.

## 8 Related Work

We compare our approach to protocol verification techniques, generic bug finding approaches, and other model checking efforts.

**Protocol reliability.** Reliability of protocol implementations has been a long running theme in networking research.

One approach is to design a domain-specific language aimed at making networking protocols concise and easy to specify, thereby (hopefully) reducing the chance of error. Examples include ESTEREL [5], LOTOS [4], and Prolac [24]. A drawback of language-based approaches is that, historically, the networking community rarely adopts them — to our knowledge, all widely-used TCP implementations are written in C or C++.

A different but related method aims to reduce errors by providing a more natural infrastructure for networking implementations. Examples include the x-kernel [21], Scout [27] and, to a degree, the Fox project [3]. This approach mainly helps protocol construction, rather than focusing on ways to find errors in the implementation.

There have been numerous attempts to test a TCP implementation. One approach involves transmitting carefully designed packets to the implementation and observing its response [11, 14]. In contrast, x-sim [7] executes an unmodified TCP implementation in a simulator to test for performance related problems.

Complementary to the testing approaches, tcpanaly [33] *passively* analyzes packet traces to detect abnormal behavior of TCP implementations. While this relies on large trace sets to achieve coverage of TCP behavior, this approach has a particularly attractive low up-front cost and scales well to large number of instances.

As stated in the introduction, by using model checking we have a higher up-front cost than these approaches but can explore protocol state spaces much more deeply.

**Generic bug finding.** There has been much recent work on bug finding, including both better type systems [15, 18, 17] and static analysis tools [13, 1, 9, 29, 16]. While the latter approaches can be easier to apply than model checking (the former can require more manual labor), both are limited to checking relatively shallow rules ("*lock* must be paired with *unlock*"). Model checking can do end-to-end checks out of their reach ("the routing table should not have loops").

**Software Model Checking.** Several recent verification tools use the idea of executing and checking systems at the implementation level.

Java PathFinder [8] uses model checking to verify concurrent Java programs for deadlock and assertion failures. It relies on a specialized virtual machine that is tailored to automatically extract the current state of a Java program.

SLAM [1] is a tool that converts C code into abstracted skeletons that contain only Boolean types. SLAM then model checks the abstracted program to see if an error state is reachable.

Our goal is to do comprehensive, deep, end-to-end checks of system correctness rather than checking a limited number of properties (as in Pathfinder) or relatively shallow, type-system level ones (as in SLAM).

Verisoft [20] systematically executes and verifies actual code and has been used to successfully check communication protocols written in C. We expect that the techniques we have developed in this paper could be applied to it as well.

As stated in the introduction, we used a prototype version of CMC in prior work [30]. This paper applies it to protocols an order of magnitude more complex, and has led to a complete overhaul of the approach.

## 9 Conclusions

CMC is successfully able to scale to systems as large and complex as the Linux TCP implementation. Also, CMC has found four bugs in the implementation.

As described in Section 7 and Table 4, CMC is successful in achieving a good coverage of the properties checked. We believe that the results reported in this paper can be improved by using a better reference model that checks for a wider range of properties. For instance, the current model does not check for congestion control properties (that the congestion window should reduce on a packet loss) and timer related properties (e.g. that a retransmission timer is appropriately scheduled for every transmitted packet).

To obviate the need for a separate hand-written reference model, we are currently exploring the possibility of simultaneously executing two different TCP implementations where one can check the behavior of the other. However, there are additional challenges in ignoring acceptable differences in the two implementations.

## 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] Boris Beizer. *Software Testing Techniques*. Electrical/Computer Science and Engineering Series. Van Nostrand Reinhold, 1983.

[3] Edoardo Biagioni. A structured TCP in standard ML. In *SIGCOMM*, pages 36–45, 1994.

[4] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. In *Computer Networks and ISDN Systems*, pages 14:25–59, 1986.

[5] Frederic Boussinot and Robert de Simone. The esterel language. Technical report, INRIA Sophia-Antipolis, July 1991.

[6] R. Braden. Requirements for internet hosts – communication layers. RFC 1122, USC/Information Sciences Institute, October 1989.

[7] Lawrence S. Brakmo and Larry L. Peterson. Experiences with network simulation. In *Measurement and Modeling of Computer Systems*, pages 80–90, 1996.

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

[9] 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.

[10] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[11] Douglas Comer and John C. Lin. Probing TCP implementations. In *USENIX Summer*, pages 245–255, 1994.

[12] 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.

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

[14] Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on six commercial TCP implementations using a software fault injection tool. *Software Practice and Experience*, 27(12):1385–1410, 1997.

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

[16] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

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

[18] J.S. Foster, T. Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the*

*ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[19] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for single-source shortest path trees. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, pages 112–224, 1994.

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

[21] N.C. Hutchinson and L.L. Peterson. The x-kernel: an architecture for implementing network protocols. *IEEE Trans. on Soft. Eng.*, 17(1), January 1991.

[22] Radu Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *Proceedings of 16th IEEE Conference on Automated Software Engineering*, 2001.

[23] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[24] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the prolac protocol language. In *SIGCOMM*, pages 3–13, 1999.

[25] S. McCanne and S. Floyd. UCB/LBNL/VINT network simulator - ns (version 2), April 1999. http://www.isi.edu/nsnam/ns/.

[26] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.

[27] Allen Brady Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman. Scout: A communications-oriented operating system (abstract). In *Operating Systems Design and Implementation*, page 200, 1994.

[28] Madanlal Musuvathi. CMC: A model checker for network protocol implementations. Technical Report PhD Thesis, Stanford University, January 2004. http://verify.stanford.edu/madan/thesis/main.pdf.

[29] Madanlal Musuvathi and Dawson R. Engler. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.

[30] 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.

[31] Paolo Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng. New dynamic SPT algorithm based on a ball-and-string model. In *INFOCOM (2)*, pages 973–981, 1999.

[32] G. Nelson. *Techniques for program verification.* Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.

[33] Vern Paxson. Automated packet trace analysis of TCP implementations. In *SIGCOMM*, pages 167–179, 1997.

[34] Vern Paxson and et.al. Known TCP Implementation Problems. RFC 2525, March 1999.

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

[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] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

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