EXPLODE: A Lightweight, General Approach to Finding Serious Errors in Storage Systems

Junfeng Yang, Paul Twohey, Ben Pfaff, Can Sar, Dawson Engler
*Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.
{junfeng,twohey,blp,csar,engler@cs.stanford.edu}

ABSTRACT

File systems, RAID systems, and applications that require data consistency, among others, assure data integrity by carefully forcing valuable data to stable storage. Unfortunately, verifying that a system can recover from a crash to a valid state at any program counter is very difficult. Previous techniques for finding data integrity bugs have been heavyweight, requiring extensive effort for each OS and file system to be checked. We demonstrate a lightweight, flexible, easy-to-apply technique by developing a tool called EXPLODE and show how we used it to find 25 serious bugs in eight Linux file systems, Linux software RAID 5, Linux NFS, and three version control systems.

1. INTRODUCTION

Many systems prevent the loss of valuable data by carefully forcing it to stable storage. Applications such as version control and mail handling systems ensure data integrity via file system synchronization services. The file system in turn uses synchronous writes, journaling, etc. to assure integrity. At the block device layer, RAID systems use redundancy to survive disk failure. At the application layer, software based on these systems is often trusted with the only copy of data, making data loss irrevocable and arbitrarily serious. Unfortunately, verifying that a system can recover from a crash to a valid state at any program counter is very difficult.

Our goal is to comprehensively test real systems for data persistence errors, adapting ideas from model checking. Traditional model checking [5] requires that the implementor rewrite the system in an artificial modeling language. A later technique, called implementation-level model checking, eliminates this requirement [19, 18, 21] by checking code directly. It is tailored to effectively find errors in system code, not verify correctness. To achieve this goal, it aggressively deploys unsound state abstractions to trade completeness for effectiveness. One major disadvantage of this technique is that it cannot check software without source code and requires porting the entire OS to run on top of a model checker, which necessitates many intrusive, non-portable modifications. Checking a new OS or a different version of the same OS requires a new port. Even checking new file systems often requires about a week of effort.

This paper describes our improved approach that remedies these problems. We reduce the infrastructure needed for checking a system to a single device driver, which can be run inside of a stock kernel that runs on real hardware. This lightweight approach makes it easy to check new file systems (and other storage systems): simply mount and run. Checking a new OS is usually a matter of implementing a device driver.

Our approach is also very general in that it rarely limits the types of checks that can be done: if you can run a program, you can check it. We used EXPLODE to check CVS and Subversion, both open source version control systems, and BitKeeper, a commercial version control system, finding bugs in all three. At the network layer, we checked the Linux NFS client and server. At the file system layer, we checked 8 different Linux file systems. Finally, at the block device layer, we checked the Linux RAID 5 implementation. EXPLODE can find errors even in programs for which we do not have the source, as we did with BitKeeper.

EXPLODE can check almost all of the myriad ways the storage layers can be stacked on one another, as shown on the left side of Figure 1. This ability has three benefits. First, EXPLODE can reuse consistency checks for one specific layer to check all the layers below it. For example, once we implement a consistency check for a file system on top of a single disk, we can easily plug in a RAID layer and check that RAID does not compromise this guarantee. Second, testing entire stacks facilitates end-to-end checking. Data consistency is essentially end-to-end: in a multi-layer system, if one layer is broken, the entire system is broken. Simply verifying that a single layer is correct may have little practical value for end-to-end consistency, and it may require a huge amount of manual effort to separate this layer from the system and build a test harness for it. Third, we can *cross-check* different implementations of one layer and localize errors. For example, if a bug occurs with an application on top of 7 out of 8 file systems, the application is probably buggy, but if it occurs on only one file system, that file system is probably buggy.

The contributions of this paper can be summarized as follows:

*This research was supported by NSF grant CCR-0326227 and 0121481, DARPA grant F29601-03-2-0117, an NSF Career award and Stanford Graduate Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

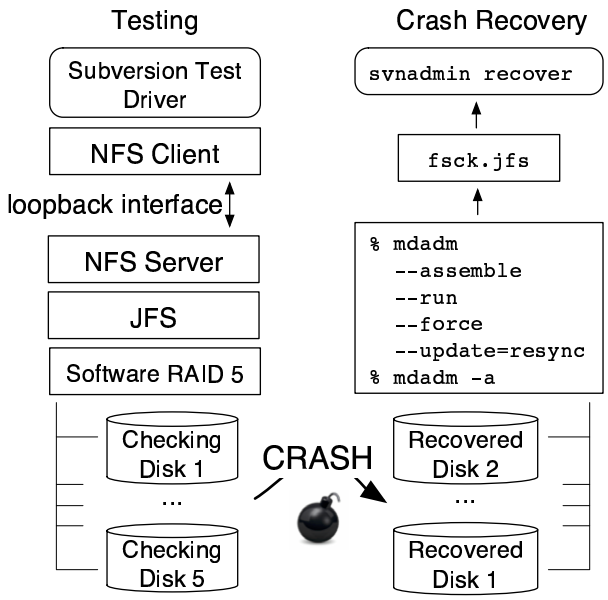

Figure 1: A snapshot of Explode **system with a stack of storage systems being tested on the left and the recovery tools being run on the right after** Explode **"crashes" the system to generate a recovery scenario.**

- A lightweight, minimally invasive approach for checking storage systems.
- Model checking every layer in a complex storage stack from RAID at the bottom to version control systems running over NFS at the top.
- A series of new file system specific checks for catching bugs in the data synchronization facilities used by applications to ensure data integrity.

This paper is organized as follows. Section 2 gives an overview of the checking system. Section 3 discusses the new challenges for our approach. Section 4 explains the basic template work needed to check a given subsystem, and how to apply this template to different storage systems. Finally, Section 5 discusses related work and Section 6 concludes.

2. SYSTEM OVERVIEW

Explode has three parts: a user-land model checking library (MCL), a specialized RAM disk driver (RDD), and a test driver. We briefly describe each of them, highlighting their most important functions and interactions. Figure 2 shows an overview.

MCL is similar in design to our prior implementation-level model checking work [19, 18, 21]. It provides a key mechanism to enumerate all possible choices at a given "choice point," which is a program point where abstractly a program can do multiple actions. For example, dynamic memory allocation can either succeed or fail. When such an allocator is properly instrumented, Explode can explore both branches.

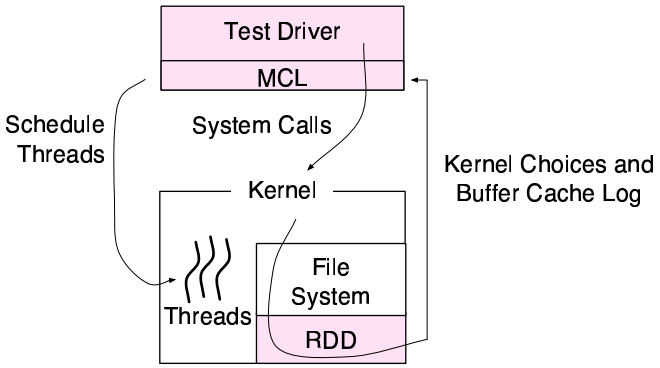

Figure 2: Explode **overview with a local file file system as an example storage system. Components of** Explode **are shaded.**

RDD provides the needed infrastructure so Explode can explore all possible behaviors for a *running* Linux kernel, using the "choice point" mechanism provided by MCL. It exports a log of buffer cache operations, allows Explode to schedule a kernel thread to run, and provides a generic interface to install and remove "choice points" inside a running kernel. We defer the discussion of why we need this functionality to §3.2.

The test driver exercises storage system operations on our specialized RAM disks to trigger potential disk writes. It also performs system-specific consistency checks once Explode is done simulating crash-recovery. To simulate all possible crashes, Explode clones current disks, applies all possible subsets of the potential disk writes[1] to the disk clones, and invokes system-specific utilities to recover them. Figure 1 shows how Explode "crashes" and recovers a running stack of a storage system.

3. IMPLEMENTATION

As mentioned in the introduction, our current approach is much more lightweight and general than our previous approach. However, this approach also poses new challenges, discussed in the following sections.

3.1 State Checkpoint and Restore

One challenge in checking live storage systems is how to checkpoint and restore storage system states. State checkpointing and restoring is essential for exploring all possible behaviors of a system. Our previous work [21] runs an entire kernel on top of a model checker, which makes checkpointing and restoring states as easy as copying the entire kernel memory and simulated disks. Since extracting relevant kernel data from a live system is difficult, we no longer checkpoint the kernel memory. Instead, to checkpoint a file system state, we store the sequence of choices that led to the current state, starting from the initial pristine disk. To

[1]In practice, if the set of potential writes grows larger than a user specified limit, Explode no longer tests all subsets but randomly tests a few.

restore a state, we unmount the current disk, then mount a copy of the initial pristine disk and replay all the previously made choices. Storage systems have the convenient property that unmounting a file system clears its in-memory state, and replaying the same set of operations on a pristine storage system is deterministic in terms of the content contained in the storage system. MCL also allows users to selectively checkpoint non-pristine disks, given that all the dirty blocks are flushed to disk.

This method for state checkpointing and restoration is identical to the state compression technique used in traditional model checking where each state is represented by a trace starting from the initial state to the current state. Unlike traditional model checking where state compression is simply a time/space tradeoff, it is essential to EXPLODE because it is the only practical method for a live kernel.

3.2 Exploring All Kernel Behaviors

With MCL we can explore the choice points we can see. However, kernel choice points are often not exposed to user-land. RDD provides three key mechanisms to expose these kernel choice points to user-land, so EXPLODE can explore all possible kernel behaviors.

First, RDD monitors all buffer cache operations and stores them in a temporal log. EXPLODE retrieves this log using an `ioctl` command provided by RDD and replays it to reconstruct the set of all possible disk images that could result from a crash. We use temporal logging instead of getting the set of dirty buffers directly from the kernel because Linux 2.6, the OS we tested, has a complicated unified buffer and page cache that makes the latter very difficult.

Second, our driver provides a general interface for installing and removing "choice points" within the kernel. Practically, EXPLODE uses this mechanism to induce artificial errors at critical places in the kernel. RDD provides a new `ioctl` that allows a process to request a failure for the nth kernel choice point in its next system call. Using this mechanism for each system call, EXPLODE fails the first choice point, then the second, and so on until all the choice points have been tested in turn. We could use this mechanism to fail more than one choice point at a time, but in our experience kernel developers are not interested in bugs resulting from multiple, simultaneous failures.

Third, our driver allows EXPLODE to schedule any kernel thread to run. Storage systems often have background threads to commit disk I/O. Controlling such threads allows EXPLODE to explore more system behaviors. As an added benefit, this makes our error traces deterministic. Without this feature EXPLODE would still find errors but it would be unable to present traces to the user which would reliably trigger the bug when replayed. Our driver implements this function by simply setting a thread to have a very high priority. Although this method is theoretically not guaranteed to run the thread in all cases, in practice it works reliably.

4. CHECKING STORAGE SYSTEMS

EXPLODE can check almost any storage system that runs on Linux, be it a file system, a RAID system, a network file system, a user-land application that handles data, or any combination thereof.

Checking a new storage system at the top of the stack is often a matter of providing EXPLODE utilities to set up, tear down and recover the storage system, and writing a test driver to mutate the storage system and check its consistency. One nice feature is that one test driver can exercise every storage system below it in the storage hierarchy.

FS	`mount sync`	`sync`	`fsync`	`O_SYNC`
ext2	✘	✓	✘	✘
ext3	✓	✓	✓	✓
ReiserFS	✘	✓	✘	✘
JFS	✘	✓	✘	✘
MSDOS	✘	✘	n/a	n/a
VFAT	✘	✘	n/a	n/a
HFS+	✘	✘	✘	?
XFS	✘	✓	✓	?

Table 1: Sync checking results. ✓: no errors found; ✘: one or more errors found; n/a: could not complete test; ?: not run due to lack of time.

This section discusses how we checked different storage systems in detail. For each of them, we first list its setup, tear-down, and recovery utilities, then describe how the test driver mutates the storage system and what consistency checks are performed. Lastly, we show the errors we found.

4.1 Checking File Systems

To set up a file system, EXPLODE needs to create a new file system (`mkfs`) and mount it. To tear it down, EXPLODE simply unmounts it. EXPLODE uses `fsck` to repair the FS after a crash.

EXPLODE's FS test driver enumerates topologies containing less than a user-specified number of files and directories. At each step the test driver either modifies the file system topology, by creating, deleting, or moving a file or directory, or a file's contents, by writing in or truncating a file. To avoid being swamped by many different file contents, the FS test driver only writes out 5 different possible file contents chosen to require varying numbers of indirect and doubly indirect blocks.

To avoid wasting time re-checking similar topologies, we memoize file systems that have isomorphic directory structure and identical file contents, disregarding file names and most metadata.

To check FS-specific crash recovery, FS developers need to provide EXPLODE with a model of how the file system should look after crash recovery. Following the terminology in [21], we call this model the StableFS. It describes what has been committed to stable storage. We call the user-visible, in-memory state of the file system the VolatileFS, because it has not necessarily been committed to stable storage.

Without an FS-specific StableFS, EXPLODE checks that the four basic sync services available to applications that care about consistency honor their guarantees. These services are: synchronous mount, which guarantees that after an FS operation returns, the StableFS and the VolatileFS are identical; `sync`, which guarantees that after `sync` returns, the StableFS and the VolatileFS are identical; `fsync`, which guarantees that the data and metadata of a given file are identical in both the StableFS and VolatileFS; and the `O_SYNC` flag for `open`, which guarantees that the data of the opened file is identical in both the StableFS and VolatileFS.

Table 1 summarizes our results. Surprisingly, 7 of the 8 file systems we tested have synchronous mount bugs – ext3 being the only tested file system with synchronous mount correctly implemented. Most file systems implement `sync`

correctly, except MSDOS, VFAT and HFS+. Crash and `fsck` on MSDOS and VFAT causes these file systems to contain directory loops, which prevents us from checking `fsync` and `O_SYNC` on them. Intuitively, `fsync` is more complicated than `sync` because it requires the FS to carefully flush out only the data and metadata of a particular file. Our results confirm this intuition, as the `fsync` test fails on three widely used file systems: ext2, ReiserFS and JFS. The JFS `fsync` bug is quite interesting. It can only be triggered when several `mkdir` and `rmdir` operations are followed by creating and `fsync`ing a file and its enclosing directories. After a crash this causes the file data to be missing. JFS developer Dave Kleikamp quickly confirmed the bug and provided a fix. The problem resided in the journal-replay code, triggered by the reuse of a directory inode by a regular file inode, so that reproducing the bug requires the journal to contain changes to the inode both as a directory and as a file.

Note that when we say a service is "correct" we simply mean EXPLODE did not find an error before we stopped it or all the possible topologies for a given number of file system objects were enumerated. We found most bugs within minutes of starting our test runs, although some took tens of minutes to discover.

4.2 Checking RAID

We tested the Linux software implementation of RAID 5 along with its administration utility `mdadm`. To set up a test run, we assembled several of our special RAM disks into a RAID array. To tear down, we disabled the RAID array. Crash recovery for RAID was not complex: we simply `fsck` the file system running on top of RAID. To recover from a disk failure, we use the `mdadm` command to replace failed disks. Read failures in individual disks in the RAID array are simulated using the kernel choice point mechanism discussed in §3.2.

We reused our file system checker on top of the RAID block device. The consistency check we perform is that the loss of any one disk in a RAID 5 array should not lead to data loss—the disk's contents can always be reconstructed by computing the exclusive-or of the $n-1$ remaining disks [20].

EXPLODE found that the Linux RAID implementation does not reconstruct bad sectors when a read error occurs. Instead, it simply marks the disk faulty, removes it from the RAID array, and returns an I/O error. EXPLODE also found that when two sector read errors happen on different disks, requiring manual maintenance, almost all maintenance operations fail. Disk write requests also fail in this case, rendering the RAID array unusable until the machine is rebooted. Software RAID developer Neil Brown confirmed that the above behaviors were undesirable and should be fixed with high priority.

4.3 Checking NFS

We used EXPLODE to check Linux's Network File System version 3 (NFSv3) and its in-kernel NFS server. To set up an NFS partition, we export a local FS as an NFS partition over the loopback interface. To tear it down, we simply unmount it. We use the `fsck` for the local file system to repair crashed NFS partitions. Currently we do not model network failures.

As NFS is a file system built on top of other file systems we are able to leverage our existing FS test driver to test the Linux NFS implementation. We can also reuse our consistency checker for synchronously mounted file systems as NFS should have identical crash recovery guarantees [4]

We found one inconsistency in NFS, in which writing to a file, then reading the same file through a hard link in a different directory yields inconsistent data. This was due to a Linux NFS security feature called "subtree checking" that adds the inode number of the file's containing directory to the file handle. Because the two links are in different directories, their file handles differ, causing the client to cache their data separately. This bug was not known to us until EXPLODE found it, but the NFS developers pointed us to a manpage describing subtree checking. The manpage said that the client had to rename a file to trigger it, but the checker did not do so. The NFS developers then clarified that the manpage was inaccurate. It described what could be done, not what Linux NFS actually implemented (because it was too difficult).

We found additional data integrity bugs in specific file systems exported as NFS, including JFS and ext2.

4.4 Checking Version Control

We tested the popular version control systems CVS, Subversion, and BitKeeper. We found serious errors that can cause committed data to be permanently lost in all three. Our test driver checks out a repository, does a local modification, commits the changes, and simulates a crash on the block device that stores the repository. It then runs `fsck` and the version control system's recovery tool (if any), and checks the resulting repository for consistency.

The errors in both CVS and BitKeeper are similar: neither attempts to sync important version control files to disk, meaning an unfortunate crash can permanently lose committed data. On the other hand, Subversion carefully `fsync`s important files in its database directory, but forgets to sync others that are equally important. (Adding a `sync` call fixes the problem for all three systems.)

If our system was not modular enough to allow us to run application checkers on top of arbitrary file systems we would have missed bugs in both BitKeeper and Subversion. On ext3 `fsync` causes all prior operations to be committed to the journal, and by default also guarantees that data blocks are flushed to disk prior to their associated metadata, hiding bugs inside applications. This ability to *cross-check* different file systems is one of the key strengths in our system.

To demonstrate that EXPLODE works fine with software to which we do not have source, we further checked BitKeeper's atomic repository syncing operations "push" and "pull." These operations should be atomic: either the merge happens as a whole or not at all. However, EXPLODE found traces where a crash during a "pull" can badly corrupt the repository in ways that its recovery tool cannot fix.

5. RELATED WORK

In this section, we compare our approach to file system testing techniques to software model checking efforts and other generic bug finding approaches.

File system testing tools. There are many file system testing frameworks that use application interfaces to stress a "live" file system with an adversarial environment. These testing frameworks are non-deterministic and less comprehensive than our approach, but they are more lightweight and work "out of the box." We view stress testing as complementary to our approach — there is no reason not to

both test a file system and then test with EXPLODE (or vice versa). In fact, our approach can be viewed as deterministic, comprehensive testing.

Software Model Checking. Model checkers have been previously used to find errors in both the design and the implementation of software systems [16, 15, 17, 2, 19, 18, 21, 6, 1]. Verisoft [15] is a software model checker that systematically explores the interleavings of a concurrent C program. Unlike the technique we use, Verisoft does not store states at checkpoints and thereby can potentially explore a state more than once. Verisoft relies heavily on partial order reduction techniques that identify (control and data) independent transitions to reduce the interleavings explored. Determining such independent transitions is extremely difficult in systems with tightly coupled threads sharing large amount of global data. As a result, Verisoft would not perform well for these systems, including the storage systems checked in this paper.

Java PathFinder [2] uses related techniques to systematically check concurrent Java programs by checkpointing states. It relies on a specialized virtual machine that is tailored to automatically extract the current state of a Java program. The techniques described in this paper are applicable to Java PathFinder as well.

Generic bug finding. There has been much recent work on bug finding, including both better type systems [9, 14, 12] and static analysis tools [8, 1, 7, 3, 10, 13]. Roughly speaking, because static analysis can examine all paths and only needs to compile code in order to check it, it is relatively better at finding errors in surface properties visible in the source ("lock is paired with unlock") [11]. In contrast, model checking requires running code, which makes it much more strenuous to apply (days or weeks instead of hours) and only lets it check executed paths. However, because it executes code it can more effectively check the properties implied by code, e.g. that the log contains valid records, that `cvs commit` will commit versioned user data to stable storage. Based on our experience with static analysis, the most serious errors in this paper would be difficult to find with that approach. But, as with testing, we view static analysis as complementary to our lightweight model checking—it is easy enough to apply that there is no reason not to apply it and then use lightweight model checking.

6. CONCLUSION

This paper demonstrated that ideas from model checking offer a promising approach to investigating crash recovery errors and that these ideas can be leveraged with much less work than previous, heavyweight implementation-level model checkers. This paper introduced a lightweight, general approach to finding such errors using a minimally invasive kernel device driver. We developed an implementation, EXPLODE, that runs on a slightly modified Linux kernel on raw hardware and applied it to a wide variety of storage systems, including eight file systems, Linux software RAID 5, Linux NFS client and server, and three version control packages, including one for which we did not have source code. We found serious data loss bugs in all of them, 25 in all. In the future we plan on extending EXPLODE in two directions: checking more comprehensively by exploring better search heuristics, and checking other mission critical storage systems such as databases.

7. REFERENCES

[1] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 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. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. RFC 1813: NFS version 3 protocol specification, June 1995.

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

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

[7] SWAT: the Coverity software analysis toolset. `http://coverity.com`.

[8] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[9] 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.

[10] 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)*, Sept. 2000.

[11] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Invited paper: Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI04)*, pages 191–210, Jan. 2004.

[12] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[13] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. 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.

[14] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 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] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[17] G. J. Holzmann. From code to models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pages 3–10, Newcastle upon Tyne, U.K., 2001.

[18] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, 2004.

[19] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[20] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Conference*, pages 109–116, June 1988.

[21] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, Dec. 2004.