Limiting Trust in the Storage Stack

Lakshmi N. Bairavasundaram, Meenali Rungta,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin, Madison
{laksh, meenali, dusseau, remzi}@cs.wisc.edu

ABSTRACT

We propose a framework for examining trust in the storage stack based on different levels of trustworthiness present across different channels of information flow. We focus on corruption in one of the channels, the data channel and as a case study, we apply type-aware corruption techniques to examine Windows NTFS behavior when on-disk pointers are corrupted. We find that NTFS does not verify on-disk pointers thoroughly before using them and that even established error handling techniques like replication are often used ineffectively. Our study indicates the need to more carefully examine how trust is managed within modern file systems.

Categories and Subject Descriptors

D [**4**]: 5. Fault-tolerance

General Terms

Reliability, Security

Keywords

Pointer corruption, Type-aware corruption, Verifiable invariants

1. INTRODUCTION

Long-term reliability, availability, and security of data is of the utmost importance, both in corporate environments as well as in the home. Indeed, much of the value people place in computer systems stems from the value of the data stored therein – as made abundantly clear by web services such as Google and digital photo management software such as iPhoto, it is the information inside our computer systems that makes them valuable to end users.

With valuable information within them, it is crucial that storage systems carefully manage such data, preserving the integrity of the data over long periods of time. However, two trends combine to make reliable management of data more challenging. First, disk complexity increases the likelihood that unusual hardware faults will arise; for example, latent sector errors [7, 16] and block corruption [13] are not uncommon occurrences. Second, the file systems themselves are increasingly complex. As features are added to improve scalability [22], consistency management [25], and other aspects of file storage, the sheer size of the file system has exploded; for example, Linux ext3 is an order of magnitude larger than its non-journaling cousin ext2.

With complex disks managed by complex file systems, a natural set of questions arise: how much trust does the file system place in the correct operation of the disk drive? Further, can the file system be trusted to manage data correctly despite its own sheer size and complexity? (*i.e.*, how much does the file system trust itself?)

We thus believe a study of how trust is managed within the storage system is necessary. In this paper, we outline a framework for studying and managing trust in the storage stack based on *trustworthiness*, the level of trust that can be attributed to any entity, and *channels*, paths in the storage stack that are used to store and retrieve information. We focus further on the effects of a corrupt or malicious data channel. We take a first step by performing a study of Windows NTFS, a popular and important commercial file system. We focus on one aspect of trust management within NTFS, specifically, how much trust it places in its on-disk pointers. It is well-known within the programming languages community that pointers are powerful constructs and must be treated with great care [6, 11, 12]; hence, we were curious to see how much care a file system such as NTFS placed within its on-disk pointers.

To perform our analysis, we use *type-aware corruption*, in which we systematically change the values of disk pointers in the file system, exercise the file system, and then observe the resulting behavior. Type-aware corruption narrows the extremely large search space by corrupting disk pointers to refer to specific types of data structures, instead of to arbitrary and random disk blocks.

From our analysis, we find that NTFS primarily relies on invariants such as the presence of a "magic" number in some block types to detect pointer corruption and uses replicas to recover from the errors. However, NTFS also fails to detect many cases of pointer corruption, leading to possible security problems and system crashes. Also, replica management in NTFS is sometimes ineffective since it is primarily targeted at recovering from data loss and not data corruption.

StorageSS'06, October 30, 2006, Alexandria, Virginia, USA.

Thus, the contributions of the paper are as follows:

- We develop a framework based on channels for exploring trust in the storage stack.
- We develop a type-aware corruption technique to explore the result of on-disk pointer corruption in file systems.
- We perform a case study of NTFS, applying type-aware corruption to the on-disk pointers of NTFS, thereby identifying invariant checks performed by the system and analyzing the corruption detection and recovery mechanisms of NTFS.

The rest of this paper is organized as follows. Section 2 motivates the need to limit trust in the storage stack. Section 3 presents our framework for exploring trust. Section 4 motivates the importance of pointers and the use of invariants in verifying pointers. Section 5 describes type-aware corruption and how it is employed to corrupt NTFS pointers. Section 6 presents the results of our experiments and Section 7 concludes.

2. MOTIVATION

This section motivates the need to rethink trust in the storage stack. We first outline the different sources of errors in the storage stack and the problems that could be caused by such errors. We then present the objectives of a system that limits trust and its potential advantages. Finally, we outline the reasons why checksums do not solve all disk error problems.

2.1 Trust-breakers

Storage stack errors can arise from a variety of sources. While some of the sources are the traditionally explored hardware errors, today there is an increasing number of software errors and security problems that can occur. We list the sources below.

- **Unreliable hardware**: The magnetic storage medium is by nature unreliable; "bit rot" is known to corrupt data stored on disk. In addition, the mechanical components of the disk can cause media scratches to occur resulting in latent sector errors thereby rendering the data inaccessible [7, 16]. Bus controller hardware has also been known to cause errors like data corruption [4].
- **Unreliable transport**: The transport between the host and disk may be unreliable and cause transient errors [24].
- **Unreliable software and firmware**: The disk firmware consists of thousands of lines of low-level code that can be buggy and thus could introduce corruption errors [8]. There are bugs that could even lead to writing the correct data to the wrong location (misdirected writes) or reporting an incomplete or failed write as completed (lost writes). In addition to bugs in disk firmware, device drivers are known to be buggy [2, 23] and could cause data corruption.
- **Malicious errors**: While maliciousness has previously been explored in the network or networked storage context [9, 14], it has recently been explored in the context of local file systems [26]. Malicious errors may be caused by a person or application with (perhaps temporary) direct access to the disk. In this case the file system image may be modified with a view to compromise the system. As a possibility, consider a malicious file system image on the widely-used portable flash drives. Even mounting the file system could lead to serious problems [26].
- **The file system**: The file system itself can be a source of storage stack errors. Recent research [27] has shown that even modern file systems have bugs. These bugs may lead to "advanced" corruption that is hard to detect.

2.2 Objectives

The objectives of a system limiting trust should be:

- **Protect against data loss**. An obvious goal of a file system that limits trust is to safeguard the data stored on disk so that it is available for access at all times.
- **Protect against intrusions**. The file system should also possess the ability to protect against security violations wherein data belonging to a user can be read, written or deleted by another unauthorized user. This objective has two parts: (i) to protect components like in-memory kernel data and applications or even file systems on other disks and (ii) protect the data on the faulty disk.
- **Forward progress**. File systems should ensure forward progress. This implies that (i) the file system should not crash the entire system indiscriminately (for example, earlier work [13] showed examples of errors for which ReiserFS crashed the entire system using `panic`), and (ii) the file system should not indefinitely retry operations thereby wasting system resources.

2.3 Advantages of Limiting Trust

An important consideration of advocating less trust on the storage stack is the advantages of such an approach. We enumerate some of these below:

- **Reliability**. According to Merriam-Webster dictionary: "Reliability is a extent to which an experiment, test or measuring procedure yields the same result on repeated trials." A system which limits its trust on the storage stack has mechanisms to detect errors and recover from them, and can therefore return the same data to the user over repeated trials regardless of possibly corrupt data returned by underlying storage system itself. It will thus improve the reliability of the system.
- **Availability**. By ensuring forward progress and not crashing the whole system when a single operation fails or part of metadata gets corrupted, such a system contributes to overall availability. Also, by protecting against data loss, such a system increases data availability.
- **Security**. Security is a crucial aspect of today's systems. Systems like PayPal store sensitive financial information about millions of people worldwide. A proactive non-trusting approach in the design of systems will help keep data and the system more secure.

2.4 Need for a New Approach

Most solutions that deal with the problem of disk integrity rely on checksumming [13, 21]. However, checksums do not solve all integrity issues. A major concern is the fact that the file system itself can contain bugs. In this case, the erroneous data might have the "correct" checksum and will go undetected. Further, in the absence of a signature-based solution, malicious modifications can and will include suitable modifications to the checksums or any replicas if present.

It is therefore important to develop a framework that can serve as a base for exploring different solutions that are suitable for the various problems that may arise. As Riedel *et al.* [14] demonstrate, a framework is very useful for studying security and survivability aspects of storage systems.

3. TRUSTWORTHINESS IN STORAGE

This section presents the different levels of trustworthiness that can found in the storage stack and discusses how these different levels can exist in different channels that comprise the storage stack. This framework is related to the disk failure model proposed in earlier work [13]. The new framework focuses on both the the type of error or trust issue as well as the source of the error in order to provide a better base for examining trust in the storage stack.

3.1 Levels

The different elements that comprise the storage stack may exhibit different levels of trustworthiness. The following are some specific levels:

- **Perfect**. The highest level is perfect: the element is reliable, available, and secure.
- **Lossy**. A lossy element is prone to losing information and is therefore less trustworthy. The hard disk drive medium is lossy. Latent sector errors can lead to data loss.
- **Corrupting**. A corrupting element could modify information randomly. For example, software and firmware bugs may lead to corruption of data stored on disk.
- **Malicious**. At the lowest level of trustworthiness, a malicious element can try to perform special modifications to the information that passes through the element.

How minimally the storage stack can be trusted depends on the hardware resources available and the performance requirements. At least some minimal level of trust is required because even if hardware and performance pose no constraints, the file system has to trust itself to a certain extent.

3.2 Channels

A channel is a path through which information flows. The storage stack can be decomposed into four channels: data, address, control and error. The data channel consists of the data and metadata that the file system stores on disk. This is the only channel that exists across different mounts of the file system. The address channel is used to specify the location on disk to read or write. The control channel specifies the operation to be performed by the storage stack, for example, to read a block. The error channel is the path used by the disk and rest of the storage stack to inform the file system of an operation's result.

Each of these channels can be associated with some level of trustworthiness. For example, a data channel in which latent sector errors occur is a lossy data channel while a data channel that consists of a maliciously generated file system image is a malicious data channel. Similarly, an address channel is considered corrupt when misdirected reads or writes can occur and the control channel is lossy when lost writes take place. We discuss below the different levels of trustworthiness applicable to each of the channels, situations in which they occur and techniques that have been or can be used to deal with lower levels of trust.

3.2.1 Data Channel

The data channel is probably the most explored channel with respect to errors that could occur. The data channel could potentially be lossy, corrupt or malicious. Replication and parity based techniques have been used by many systems over the years to deal with lossy data channels. Corrupt or malicious data channels are harder to deal with. Checksums can be used to a limited extent in dealing with corrupt data channels [20, 21]. As discussed earlier, checksums do not protect against corrupt data channels where the source of corruption is the file system itself. In this paper, we explore problems caused by a corrupt or malicious data channel. It is important to note that a malicious data channel does not imply that the data on disk need not be protected. It only implies that some of the data (or metadata) might be maliciously generated.

The easiest method that can be used to create a malicious data channel is to modify the on-disk pointers of the file system. Pointers and data can be verified against invariants by the file system to protect against pointer corruption. Section 4 discusses pointers and invariants in greater detail.

3.2.2 Address Channel

The address channel could potentially be corrupt or malicious. An example of a corrupt address channel is one that causes misdirected reads or writes. Checksums located outside of the data blocks that they checksum, like used in earlier work [13, 20], can help in detecting problems due to a corrupt address channel. A malicious address channel is difficult to deal with. One needs redundant hardware paths to safely access the data on disk when the address channel could be malicious.

3.2.3 Control Channel

The control channel could be lossy, corrupt or malicious. Only lossy control channels can be dealt with by software-only methods. Verifying data written to the disk by performing a subsequent read can help detect control channel write losses. Sanity checking and suitable memory initialization prior to reads can possibly detect control channel read losses. These techniques need to explored in detail in future work.

3.2.4 Error Channel

The error channel could be lossy, corrupt or malicious. When the error channel is lossy, the effect is similar to that of a control channel loss and needs to be dealt with in similar fashion. A corrupt or malicious error channel may indicate the wrong type of error when an error occurs. Some types of errors can invoke retries by the file system and drivers. A malicious error channel could possibly invoke indefinite retries thereby wasting system resources. File systems should employ defenses against threats to forward progress by limiting retries.

In this paper, we focus on limiting trust in the data channel, specifically on corrupt or malicious data channels. Limiting trust in the other channels presents a great opportunity for future work.

4. POINTERS AND INVARIANTS

This section first outlines the importance of protecting pointers and safeguarding pointer-based operations in the context of a corrupt or malicious data channel, and then discusses the use of *verifiable invariants* to protect pointer-based reads and writes.

Pointers are powerful constituents of the data channel. The on-disk state of most file systems consists of a set of data structures which relate to one another via pointers [19]. For example, a UNIX inode typically contains N direct disk pointers, which are addresses of the first N data blocks of the file. References may take different forms as well. For example, a UNIX directory contains a list of (filename, inode number) tuples; the inode number here serves as an index into the on-disk inode array, and hence combining it with a base offset results in a true disk pointer.

Previous work in *on-disk consistency management* has realized the salience of disk pointers. For example, many early UNIX file systems [10, 15] carefully order writes to prevent the creation of bad on-disk pointers (*e.g.*, FFS makes sure to initialize a directory data block *Dir* before writing a structure that has a pointer to *Dir* to disk). Subsequent work on soft updates [3] and journaling file systems [5, 17] also treat pointers with care to maintain metadata consistency in the presence of crashes. Recent work [18] explores the need to protect pointers from corruption and proposes the use of type safe disks that understand on-disk pointers and can therefore better protect the data stored on disk from pointer corruption.

Less well known is how file systems behave or should behave when confronted with a corrupt or malicious data channel that contains a disk pointer with the wrong value. Corrupt pointers can lead to many problems within the file system. For example, given that one must know where a data structure is located on disk in order to reference it, a corrupt pointer can lead to inaccessible data. Further, a corrupt pointer incorrectly referring to data belonging to another data structure can cause the unlucky data to be overwritten and corrupted as well.

We propose the use of *verifiable invariants* as a file system level technique that can be used to deal with incorrect or malicious pointers. Many file systems have invariants that can be verified against during file system operation. A list of some possible invariants follow. (i) A basic invariant true for all file systems is that the same location on disk cannot contain two different data objects. For example, the data block of a user file cannot co-exist with the superblock of the file system. (ii) An invariant for almost all file systems is that the superblock occupies a known location on disk, typically block 0. (iii) A third invariant is that a block belonging to the file system cannot be present outside the boundaries of the file system. The given invariants for a file system can be combined to generate rules for system operation. For example, combining invariants (i) and (ii), block 0 cannot be the data block for a user file. This invariant can now be used when reading or writing data to a user file to protect against illegal read or write operations to the superblock. While the invariants listed above are simple and static, and can be verified in the form of "assertions" while reading metadata, a more advanced file system needs techniques to deal with a large number of invariants that could be specific to each file system image.

Identifying invariants used by a file system helps in characterizing a significant portion of the capabilities of the file system in protecting against pointer corruption. In this paper, we explore how NTFS deals with pointer corruption and identify invariants used by NTFS for this purpose.

5. TYPE-AWARE CORRUPTION

To identify the behavior of NTFS when disk pointers are corrupted, we develop and apply *type-aware corruption*. With this approach, we directly observe how NTFS reacts after we have corrupted important disk pointers to values within ranges of different types. Type-awareness is necessary because it would be nearly impossible to corrupt a pointer on disk to every possible value in a reasonable amount of time.

Type-awareness assumes that pointers of the same type behave similarly. This assumption has two components. First, we assume that corrupting a pointer A of type Z to value X induces the same behavior as corrupting a pointer B of type Z to value X; for example, corrupting a pointer to file data of File A is same as corrupting a pointer to file data of File B. Second, we assume that corrupting a pointer to value X that exists in a region of type Z induces the same behavior as corrupting to a value Y that also exists in a region of type Z; for example, corrupting a pointer to refer to the beginning of the log is the same as referring to the middle of the log. Thus, type-aware corruption greatly reduces the experimental space while still covering almost all of the interesting cases.

Our framework uses two layers, one on each side of the file system, to control experiments and to corrupt data structures read by the file system. The first layer is a user-level *microbenchmark* layer that executes file system operations above NTFS. The second layer is the *corrupter* that resides between NTFS and the disk drivers. This layer corrupts on-disk pointers and bitmaps and observes disk traffic, and has been implemented as an upper filter driver.

NTFS uses many different pointers on disk. A detailed description of NTFS data structures and pointers can be found elsewhere [1]. We have performed experiments that corrupt all the pointer types used by NTFS using values that correspond to the entire range of cluster[1] types on disk. In this paper, we discuss experiments with 5 pointers: the pointer to the Master File Table (MFT) present in the boot sector (Boot-MFT), the pointer to the logfile present in the first cluster of the MFT (MFT-Logfile), the pointer to the MFT Bitmap (which controls allocation of MFT records) present in the first cluster of the MFT (MFTBitmap), the pointer to a directory index buffer (which contains directory entries) present in the MFT record for a directory (DirIndxBuf) and the pointer to a data cluster present in the MFT record for a user file. Each pointer is corrupted to point to a range of different values. We use the following ranges of values: (a) values that point to essential clusters (*e.g.*, boot sector), (b) values that point to different data types (*e.g.*, directory index buffer and MFT cluster), (c) unallocated clusters, and (d) out of range values. Each experiment corrupts exactly one pointer to one value in the above range.

6. EXPERIMENTS

This section presents the results of pointer corruption experiments on NTFS. We first outline the experimental setup and workloads used, then present the results and finally analyze the results obtained.

6.1 Experimental Setup and Workloads

All our experiments are performed on an installation of Windows XP (Professional Edition without Service Pack 2) run on top of VMWare Workstation 4.5.2. The experiments use a separate 2GB IDE virtual disk.

The workloads used to observe NTFS reactions to pointer corruption are as follows. The file system is mounted to

[1]NTFS refers to data blocks as *clusters* and we adopt the terminology for the rest of the paper.

exercise the boot sector's MFT pointer (Boot-MFT) and the MFT's logfile pointer (MFT-Logfile). A file is created to exercise the MFT Bitmap pointer and the Directory Index Buffer pointer (DirIndxBuf). A file is written to in order to exercise the file data pointer.

6.2 Results

We first discuss corruption cases that are not detected by NTFS, then discuss cases that are detected, then list invariants used by NTFS to perform detection, and finally discuss the mechanisms used by NTFS to deal with detected corruption events.

6.2.1 No detection

NTFS does not detect pointer corruption when the MFT Bitmap pointer or the File data pointer are corrupted, the only exceptions occurring when the pointer points outside the file system boundary. This has two direct effects on the system. First, in both cases, NTFS overwrites the cluster pointed to by the pointer, thereby corrupting the cluster. Second, in the case of the MFT Bitmap, NTFS also uses the information in the cluster pointed to for allocating an MFT record entry for the file being created, possibly overwriting a valid file entry. This behavior in turn causes NTFS to overwrite the file entry of a system file which eventually leads to a system crash (blue screen).

NTFS also does not detect a pointer corruption when a directory index buffer pointer points to the some other directory's index buffer, thereby corrupting the other directory by adding a new file to it (the workload is file create).

Not detecting file data pointer corruption is a security problem. After a single malicious corruption event, the entire file system could possibly be compromised forever.

6.2.2 Detection

NTFS detects pointer corruption when the pointer in question is Boot-MFT, MFT-Logfile or DirIndxBuf. It also detects corruption if any pointer points beyond the file system boundary.

The *timing* of detection can vary. Specifically, detection can occur immediately, or after the cluster pointed to is read, or during a delayed write:

- **Immediate:** Whenever the pointer directly points outside the disk partition, NTFS does not require a disk read to detect the corruption, *i.e.*, the detection is immediate. Immediate detection also occurs when Boot-MFT has the same value as the pointer to MFT's replica, also in the boot sector. This shows that NTFS checks to ensure that the two replicas do not point to the same location.
- **After cluster pointed to is read:** In some cases, NTFS reads the cluster pointed to, and detects the error when it performs type checking. For example, directory index buffers are typed with the magic number "INDX". When a file is created, NTFS reads the directory index buffer cluster, detects the absence of the magic number, and returns an error stating that the directory is corrupt.
- **During "delayed write":** This detection timing is specific to some cases where the pointer points to the last cluster (which contains a copy of the boot sector) and there is a write to the cluster. The write operation returns success to the application, and the corruption is detected when the cluster is later written to disk. This typically results in a "pop-up" that informs the user that a delayed write failed.

6.2.3 Invariants

We can use the results of the experiments, especially whether detection occurred and the detection time, to develop the set of invariants used by NTFS. We summarize the invariants below:

- A pointer cannot point outside the file system boundary.
- Replicas cannot occupy the same location on disk.
- Directory index buffers contain the magic number "INDX", and the first log file cluster (the log restart area) contains the magic number "RSTR"
- The first MFT cluster contains the magic number "FILE" and contains the MFT record number as 0.

However, NTFS does not use simple invariants like "a user file write cannot overwrite block 0 (boot sector) with random data". Given that the NTFS source code is not available, obtaining information regarding these invariants demonstrates the utility of type-aware corruption.

6.2.4 Reactions

The various ways in which NTFS reacts upon detecting an error are summarized below:

- **Recovery:** If possible, NTFS uses redundancy to obtain the required information. The first cluster of the MFT is replicated and the boot sector contains a pointer to the replica. Therefore, when Boot-MFT is corrupted, NTFS reads the cluster pointed to, detects that the cluster is not the right one, then chooses to use the information in the replica of the MFT.
- **Retry the entire mount:** NTFS uses retries to recover from corruption in some cases. This behavior could be useful if the corruption was not persistent on disk, but due to a transient malfunctioning of some component in the system. In particular, we observe retries when the MFT's pointer to the logfile is corrupted.
- **Propagate error to application:** This action is observed when type checks fail, and recovery is not possible because of lack of redundancy. For example, when a directory's index buffer points to any cluster that does not have the magic number "INDX", the error is propagated to the application.

6.3 Analysis

Limiting trust in the storage stack should form one of the core design goals of a file system. Our experiments with NTFS have demonstrated some of the dangers that arise when not designed so. In this section, we outline the lessons learned towards techniques used to limit trust.

- *Filesystems should place a low level of trust on the data channel even for user files.* A lossy data channel has a low impact when user files are accessed. However, a corrupt or malicious data channel can pose significant problems even when user files are accessed, as demonstrated by the ability to read or write any portion of the disk with a corrupt user file data pointer in the case of NTFS.
- *Filesystems should devise strong, useful invariants.* NTFS experiments show that simple type checking doesn't work if one directory points to another, and the file system can be left in an inconsistent state. It might be more useful to devise more sophisticated invariants that take into consideration, for example, the directory's ID along with the type.
- *A replica is only useful if it is actually used.* We have

seen that NTFS does not take advantage of the correct information that is stored in a replica when the primary has been corrupted. For example, NTFS does not always use the information in the MFT replica, even after detecting that primary MFT is corrupt. A specific case in point is when NTFS detects that the log file (or the pointer to it) is corrupt and simply retries the mount without checking the copy of the pointer in the MFT replica. In other cases, NTFS could have used a replica to detect corruption, but did not bother to perform the comparison; for example, NTFS does not detect corruption of the MFTBitmap pointer even though a copy is readily available in the MFT replica (which is also read).

- *A replica is only useful if it not destroyed.* In some instances, NTFS actually destroys the contents of valid replicas. NTFS treats each replica strictly as either a primary or a secondary, that is NTFS places greater trust in one of the replicas, even though there is no reason why that replica is more trustworthy; for example, in the case where the primary MFT is corrupt, but the MFT replica is correct, NTFS erroneously synchronizes the two copies by overwriting the correct MFT replica with the contents of the corrupt MFT.

While some of the lessons are well-established, it simply underscores the fact that even a widely-used commercial file system lacks the capability to deal with corruption effectively. Detailed studies of other file systems like ZFS [21] will help understand current systems better.

7. CONCLUSIONS

File systems have long placed a great deal of trust in the disks they place data upon and within the correctness of the file system code itself. In this paper, we show the potential ill-effects of such trust: less reliable and secure file systems than one might hope for. As we consider building the next generation of file systems, it is important to carefully consider exactly what levels of trust the file system places in the disk subsystem and within its own code. By making such trust a first-class consideration, the reliability, availability, and security of storage is likely to significantly improve.

8. REFERENCES

[1] A. Altaparmakov. The Linux-NTFS Project. http://linux-ntfs.sourceforge.net/ntfs, August 2005.

[2] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[3] G. R. Ganger and Y. N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[4] R. Green. EIDE Controller Flaws Version 24. http://mindprod.com/jgloss/eideflaw.html, 2005.

[5] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[6] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[7] H. H. Kari. *Latent Sector Faults and Reliability of Disk Arrays*. PhD thesis, Helsinki University of Technology, September 1997.

[8] B. Lewis. Smart Filers and Dumb Disks. NSIC OSD Working Group Meeting, April 1999.

[9] J. Li, M. Krohn, D. Mazires, and D. Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[10] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[11] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[12] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[14] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 14–29, Monterey, California, January 2002.

[15] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[16] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proceedings of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Volendam, Netherlands, October 2004.

[17] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, San Diego, California, June 2000.

[18] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06) (to appear)*, Seattle, WA, November 2006.

[19] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A Logic of File Systems. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 1–15, San Francisco, California, December 2005.

[20] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[21] Sun Microsystems. ZFS: The Last Word in File Systems. http://www.sun.com/2004-0914/feature/, 2004.

[22] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[23] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[24] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, April 1999.

[25] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[26] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *IEEE Security and Privacy*, Berkeley, California, May 2006.

[27] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.