Analyzing the Effects of Disk-Pointer Corruption

Lakshmi N. Bairavasundaram, Meenali Rungta[‡], Nitin Agrawal,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift
University of Wisconsin-Madison [‡]*Google, Inc.*

Abstract

The long-term availability of data stored in a file system depends on how well it safeguards on-disk pointers used to access the data. Ideally, a system would correct all pointer errors. In this paper, we examine how well corruption-handling techniques work in reality. We develop a new technique called type-aware pointer corruption *to systematically explore how a file system reacts to corrupt pointers. This approach reduces the exploration space for corruption experiments and works without source code.*

We use type-aware pointer corruption to examine Windows NTFS and Linux ext3. We find that they rely on type and sanity checks to detect corruption, and NTFS recovers using replication in some instances. However, NTFS and ext3 do not recover from most corruptions, including many scenarios for which they possess sufficient redundant information, leading to further corruption, crashes, and unmountable file systems. We use our study to identify important lessons for handling corrupt pointers.

1. Introduction

Much of the value people place in computer systems stems from the value of the data stored therein. The long-term availability of such data is therefore of the utmost importance. An integral part of ensuring the long-term availability of data is ensuring the reliability and availability of *access paths* to data, that is, pointers. Pointers are fundamental to the construction of nearly all data structures. This observation is especially true for file systems, which rely on pointers located in on-disk metadata to access data. Unfortunately, disk pointers are susceptible to corruption for various reasons; the literature is rife with examples of disk, controller, and transport flaws [3, 12, 13, 18, 24] and file system bugs [30] that lead to on-disk corruption.

File systems today use a variety of techniques to protect against corruption. ReiserFS, JFS and Windows NTFS perform lightweight corruption checks like type checking [18]; that is, ensuring that the disk block being read contains the expected data type. These file systems also employ sanity checking (verifying that particular values in data structures follow certain constraints) to detect corruption [18]. ZFS checksums both data and metadata blocks to protect against corruption [8]. The techniques above are useful for detecting corruption. In order to recover from corruption, most systems rely on replicated data structures. For example, JFS and NTFS replicate key data structures, giving them the potential to recover from corruption of these structures [7, 22].

In this paper, we seek to evaluate how a set of corruption-handling techniques work in reality. While conceptually simple, there may be design or implementation details that preclude a file system from reaping the full reliability benefit of these techniques. We evaluate file systems using software fault injection. One difficulty with a pointer-corruption study is the potentially huge exploration space for corruption experiments. To deal with this problem, we develop a new fault injection technique called *type-aware pointer corruption* (TAPC). TAPC reduces the search space by systematically changing the values of only one disk pointer of each type in the file system, then exercising the file system and observing its behavior. We further narrow the large search space by corrupting the disk pointers to refer to each type of data structure, instead of to random disk blocks. An important advantage of TAPC is that it helps understand the underlying causes for observed system behavior. TAPC works outside the file system, obviating the need for source code.

We use TAPC to evaluate two widely-used file systems, Windows NTFS and Linux ext3 [28]. We examine their use of type checking, sanity checking, and replication to deal with corrupt pointers. We ask the simple question: *do these techniques work well in reality?* We focus on NTFS in this paper; our study of ext3 is less-detailed, primarily aimed at demonstrating the general utility of our approach.

We find that NTFS successfully uses type information to defend against many pointer-corruption scenarios. NTFS detects corruption by verifying the presence of a "magic number" in data structures that it accesses. NTFS also replicates key data structures to automatically recover from corruption. TAPC thus enables us to identify the checks performed and techniques used by NTFS to deal with corruption, without knowledge of source code.

Of our 360 different corruption scenarios, NTFS is able to continue normal operation in 61 scenarios (17%). We find that NTFS cannot handle many cases of pointer corruption, leading to data or metadata loss in 102 cases (28%),

system crashes in 22 cases (6%), and unmountable file systems in 133 cases (37%). Despite type information and redundancy, NTFS fails to recover from many pointer corruptions as it does not always correctly use this information.

We examine 93 corruption scenarios in ext3. In contrast to NTFS, we find that ext3 relies more on sanity checks than on type checks, thus detecting different corruptions. Although ext3 extensively replicates key data structures, it never uses the replicas to recover; its typical reaction is to report an error and remount the file system read-only. Thus, ext3 is no better than NTFS in handling pointer corruption.

We use our analyses to identify several lessons and pitfalls for building corruption-proof file systems, including:

- Type checking does not work for all pointers. Detailed sanity checking should also be performed.
- Replication should be managed and used with corruption in mind: systems should compare replicas before overwriting, and use different pointers for replicas.
- Many indexes are simply performance improvements and their loss should not cause the file system to fail.

Had these lessons been applied, NTFS and ext3 could have recovered from an additional 144 and 39 scenarios respectively in which they currently fail.

The rest of this paper is organized as follows. Section 2 discusses the problem of disk corruption. Section 3 describes type-aware pointer corruption. Section 4 presents an overview of NTFS and Section 5 presents the results of our analysis. We discuss related work in Section 6, and conclude in Section 7.

2. Motivation

In this section, we motivate our study by describing how blocks on disk can become corrupted and why we focus on the corruption of pointers.

Disk Corruption: Sources of disk corruption are throughout the storage stack, including errors within file systems, device drivers, bus controller, transport layer, disk firmware, and the electrical, mechanical and media components of the disk. A software bug within the file system, or a corruption of main memory, can cause the file system to write incorrect data to disk. Further, buggy device drivers can issue disk requests with bad parameters or data [10, 11]. Bus controllers have also been shown to incorrectly indicate that disk requests are complete or to swap status bits with data [13]. Drive firmware sometimes silently corrupts data, directs writes to the wrong location, or reports the data has been written when in fact it has not [12, 24]. Within the disk, power spikes, erratic arm movements, media scratches, and "bit rot" (change in bit state over time) could cause disk blocks to become corrupted (although most medium errors are caught by disk ECC) [1, 19, 26]. In a study involving 1.53 million disks in production storage systems, we found that 0.66% of SATA drives and 0.06% of FC drives developed corruption in 17 months of use [3].

Why Pointer Corruption: Although any block on disk may become corrupt, some corruptions are more damaging than others. If a data block of a file is corrupt, then only the application that reads the file is impacted. However, if a disk block belonging to file-system metadata is corrupt, then the entire file system can be affected; for example, if the boot sector is corrupt, the file system may not be mountable. In other cases, a corrupt pointer incorrectly referring to data belonging to a different data structure can cause the data to be overwritten and corrupted. Therefore, we focus on effects of corrupt pointers.

3. Type-Aware Pointer Corruption

To identify the behavior of file systems when disk pointers are corrupted, we develop and apply *type-aware pointer corruption* (TAPC). We observe how the file system reacts after we modify different types of on-disk pointers to refer to disk blocks containing different types of data.

A pointer-corruption study is especially difficult because it is nearly impossible to corrupt every pointer on disk to every possible value in a reasonable amount of time. Often, the solution has been to use random values. This approach suffers from two problems: (a) a large number of corruption experiments might be needed to trigger the interesting scenarios, and (b) use of random values makes it more difficult to understand underlying causes of observed behavior.

We use type-awareness to address both problems. Type-awareness reduces the exploration space for corruption experiments by assuming that system behavior depends only on two types: (i) the type of pointer that has been corrupted, and (ii) the type of block that it points to after corruption. Examples are (i) corrupting File A's data pointer is the same as corrupting File B's data pointer, and (ii) corrupting a pointer to refer to inode-block P is the same as corrupting it to refer to inode-block Q (if all inodes in P and Q are for user files). This approach is motivated by the fact that code paths within the file system that exercise the same types of pointers are the same, and disk blocks of the same type of data structure contain similar contents. Thus, TAPC greatly reduces the experimental space while still covering almost all of the interesting cases. Also, by its very design, this approach attaches file system semantics to each experiment, which can be used to understand the results.

Terminology: The following terms are used to describe methodology and discuss results.

- $Container$: disk block in which the disk pointer is present. Corrupting the pointer involves modifying the contents of the $container$.
- $Target_{original}$: disk block that the disk pointer should point to; that is the block pointed to on no corruption.

Term	Description
Cluster	The fundamental unit of disk storage; it consists of a fixed number of sectors, similar to a UNIX disk block.
LCN	A Logical Cluster Number (LCN) is assigned to each disk cluster. This is the same as a physical block number in UNIX-based systems. On-disk pointers contain the LCN of the cluster they point to.
VCN	A Virtual Cluster Number is the same as a file offset (in number of blocks) in UNIX.
Data run	The format of NTFS on-disk pointers, consisting of a base LCN and length, and a series of <offset,length> fields. E.g., if base LCN is X, the length field is a, and the first <offset,length> combination is $< b, c >$, the data being pointed to is located at LCNs X to $X + a$ and then from $X + b$ to $X + b + c$. In our experiments we corrupt the base LCN.
Boot sector	The boot sector is the sector read first by NTFS when the file system is mounted. It is the starting point for discovering the LCNs of all other data structures. The last cluster of the file system contains a copy of the boot sector.
MFT	Master File Table contains an entry for each file (both user and system). First 24 entries are reserved for system files.
MFT entry	Equivalent of a UNIX inode. Most pointers that are corrupted are located in different MFT entries in form of data runs.
MFT VCN 0	This is the first cluster of the MFT. Its LCN is present in the boot sector. The first entry of this cluster is a file that contains LCNs of itself and the rest of the MFT.
MFT mirror	This is a replica of MFT VCN 0. Its LCN is also present in the boot sector.
Index buffer	An index buffer consists of a series of index entries that provide information for indexing into any data structure.
Directory	A directory in NTFS consists of index buffers. The entries in these buffers point to MFT entries of the directory's files.
MFT bitmap	This is a bitmap that tracks whether MFT entries are allocated or not.
Volume bitmap	This is a bitmap that tracks whether disk clusters are allocated or not.
Log file	NTFS implements ordered journaling mode: whenever a user writes data to disk, the data cluster is flushed first, followed by log updates, and finally the metadata clusters. It is organized as a restart area, a redundant copy of the restart area, and a "logging area", which consists of log records that each denote a disk action to be performed.
$Secure	NTFS stores information about the owner of the file and the permissions granted to other users by the owner (in form of ACLs) in a security descriptor. Each unique descriptor is stored in $Secure along with its hash and given a *security id*. This security id is stored in the MFT entry of the file for looking up the correct descriptor from $Secure. The descriptors in $Secure are indexed on the hash of the security descriptor and the security id.
Upcase table	This is an upper case - lower case character conversion table essential for directory path name traversal.

Table 1. NTFS Terminology. *This table provides brief descriptions of NTFS terminology and data structures. The descriptions offer a simplified view of NTFS, eliminating details that are not essential for understanding the experiments.*

- $Target_{corrupt}$: disk block being pointed to by a corrupt disk pointer.

Corruption Model: Any of the sources of corruption discussed in Section 2 could produce a corrupt file system image on disk. Our corruption model reflects the state of a file system on functioning hardware that experienced a corruption event in the past:

- Exactly one pointer is corrupted for each experiment. The rest of the data is not corrupted. Also, other faults like crashes or sector errors are not injected.
- We emulate pointer corruptions that are *persistent*. The corruption is persistent because simply re-reading the pointer from disk will not recover the correct value.
- The pointer corruption is *not sticky*. Future writes to the pointer by the file system can potentially correct the corruption. Reads performed after a write will be returned the newly written data and not the corrupt data.

Corruption Framework: Our TAPC framework has been designed to work without file system source code. It consists of a *corrupter* layer that injects pointer corruption and a *test harness* that controls the experiments. The corrupter resides between the file system and the disk drivers; the layer has been implemented as a Windows filter driver for NTFS and as a pseudo-device for ext3. This layer corrupts disk pointers and observes disk traffic. Thus, the corrupter has knowledge of the file system's on disk data structures [21]. The test harness is a user-level program that executes file system operations and controls the corrupter. The experiments involve the following steps:

- The test harness creates a file system on disk with a few files and directories. It then instructs the corrupter to corrupt a specific pointer to a specific value and performs file operations (*e.g.*, mount, `CreateFile`, etc. for NTFS and mount, `creat`, etc. for ext3) to exercise the pointer under consideration. We execute the file operations from a user with limited permissions (non-administrator).
- The corrupter intercepts the disk accesses performed by the file system and scans the requests for the $container$ (the disk block containing the pointer). When that disk block is read, exactly one pointer in the data structure is modified to a specific value.
- The corrupter continues to monitor disk accesses. The same corruption is performed on future reads to the $container$. Disk writes to the $container$ may overwrite any corruption and therefore further reads to the disk block are returned the newly-written data.
- All disk accesses, system call return values, and the system event log are examined in order to identify the behavior of the file system. This holistic view of system behavior in co-ordination with type-awareness is essential to understanding the underlying design or implementation flaws that lead to any system failures.

Our experiments are performed on an installation of Windows XP (Professional Edition without Service Pack 2)

Pointer	Container	$Target_{original}$
Boot-MFT0	Boot	MFT VCN 0
Boot-MFTM	Boot	MFT mirror
MFT0-MFT	MFT VCN 0	The MFT clusters (to itself)
MFTBitmap	MFT VCN 0	MFT bitmap
MFT0-MFTM	MFT VCN 0	MFT mirror
LogFile	MFT VCN 0	Log file
RootSecDesc	MFT VCN 1	Root directory security descriptor
RootIndxBuf	MFT VCN 1	Root directory index buffers
SDS	MFT VCN 2	$Secure security descriptors
SDH	MFT VCN 2	Index of security descriptors' hash
SII	MFT VCN 2	Index of security descriptors' ids
UpCase	MFT VCN 2	Upcase table
DirIndxBuf	MFT any VCN	A directory's index buffer
FileData	MFT any VCN	A file's data cluster

Table 2. NTFS Disk Pointers. *This table presents the different on-disk pointers used by NTFS.*

for NTFS and Linux 2.6.12 for ext3. We run them both on top of VMWare Workstation for ease of experimentation. The experiments use a separate 2GB IDE virtual disk. We believe that the use of a VMWare virtual disk does not change the results; since the corrupter layer is between the file system and the virtual disk, we observe all disk requests and responses, and we did not detect any anomaly.

4. NTFS

Although TAPC can be applied to any file system, the specific pointers to be corrupted and the interesting corruption values depend upon the file system under test. We now describe how we have applied TAPC to NTFS. We do not provide ext3 details due to space constraints.

NTFS Data Structures: We provide a brief introduction to NTFS. A detailed description can be found elsewhere [22]. NTFS, the Windows NT File System, is the standard file system for Windows NT, 2000, XP and Vista. It is a journaling file system that guarantees the integrity of its metadata structures on a crash. All user data and metadata structures in an NTFS volume are contained in files, allowing NTFS to flexibly allocate disk space for its metadata. Table 1 defines important NTFS terms and data structures that we use in our descriptions and results. For example, a *cluster* is the NTFS term for a disk block.

NTFS Pointer Corruption: We corrupt 14 of the 15 different pointer types that NTFS uses on disk. Table 2 summarizes these pointers. We give each pointer a unique name based on its $Target_{original}$, and resolving name conflicts by prefixing those names with its $container$. Note that NTFS replicates important data structures like Boot and MFT VCN 0. Thus, the pointers Boot-MFT0, Boot-MFTM, MFT0-MFT, MFTBitmap, MFT0-MFTM, and LogFile are replicated. Security descriptors are also replicated and their indexes can be rebuilt; thus, some form of redundancy exists for the pointers SDS, SDH, and SII.

Workload	Pointer
mount	Boot-MFT0, Boot-MFTM, MFT0-MFT, MFT0-MFTM, LogFile, RootSecDesc, SDS, SII
mount then CreateFile	MFTBitmap, RootIndxBuf, SDH, DirIndxBuf
mount then ReadFile	UpCase
mount then WriteFile	FileData

Table 3. NTFS Workloads. *This table presents the workloads that exercise the disk pointers.* mount *enables the file system volume for use; it consists of a* DeviceIoControl *system call with the control code FSCTL_UNLOCK_VOLUME performed on a previously "locked" volume.* CreateFile *creates a new file of size* 0*,* ReadFile *reads the first cluster of a file, and* WriteFile *writes the first cluster of a file.*

To exercise each pointer, we run a specialized workload; Table 3 indicates the workload used for each of the pointers. Most workloads involves modifications to $Target_{original}$, potentially creating the worst case scenario in case the corruption is not detected. The pointers are corrupted to the 27 different types of values. In addition to using disk locations that belong to all the different NTFS data types (*e.g.*, directory index buffer and MFT cluster), we also include clusters of a certain type that serve a special purpose (*e.g.*, MFT VCN 0, MFT mirror), unallocated clusters, and out-of-range values. Table 4 lists the different types of values used as $Target_{corrupt}$. In most cases, the data structure used as $Target_{corrupt}$ is at a specific location, while for FileData, we create a file and use the location of its data block as the numerical value for corruption. Thus, we perform 360 experiments on NTFS, corrupting 14 different pointers with 27 different values.

5. Results

This section discusses the results. First, we describe some terminology, then our visual representation of the results. Then, we discuss NTFS behavior as observed by the experimenter. Our discussion focuses on how NTFS deals with pointer corruption. Next, we discuss the user-visible results of NTFS pointer corruption. This view is important since the primary concern of end users is the observed data and system reliability. Finally, we present results for ext3. We organize our results into *observations* (facets of system behavior uncovered by TAPC), *lessons* for corruption-handling techniques, and potential design *pitfalls*.

5.1. Terminology for System Behavior

Detection: The file system identifies that either the pointer or the disk block pointed to is corrupt.

Recovery: The file system is able to regenerate the data lost due to pointer corruption using redundant information,

Value	Description
Boot	The boot sector (LCN 0)
LogRes	Log restart area
LogResDup	Copy of Log restart area
LogData	Log data cluster
MFTBitmap	The MFT bitmap
MFT0	MFT VCN 0
MFT1	MFT VCN 1
MFT2	MFT VCN 2
MFTRes	Contains unused, reserved MFT entries
MFTFree	Unallocated MFT entries
MFT6	MFT VCN 6
MFTOthers	Contains user file MFT entries
SDS	Security descriptors
AttrDef	File with definitions of file attributes
SDH	Index of security descriptor hash
SII	Index of security descriptor ids
MFTMirror	The MFT mirror
RootIndxBuf	Root directory index buffer
RootSecDesc	Root dir security descriptor
VolBitmap	Volume bitmap
UpCase	Upcase table
DirIndxBuf	Any directory index buffer
FileData	Any user file data cluster
Unalloc	Unallocated clusters
Last-Size+1	Data Run ends at last cluster
LastCluster	Boot sector copy
Out-of-Bounds	Data Run exceeds disk partition

Table 4. NTFS Pointer Corruption Values. *This table presents the different values used for corrupting disk pointers used by NTFS, sorted in the order of typical disk location. In total, 27 different values are used. Note that the value Last-Size+1 is applicable only for pointers that point to data runs of length > 1.*

thereby continuing execution without errors.

Report: The file system informs the application or user that it has encountered an error.

Retry: The file system repeats the set of disk accesses needed for the mount operation.

Repair: The file system modifies corrupt data structures in order to continue execution. The modification does not necessarily lead to error-free execution.

Detection is essential for the rest of the actions to occur. Recovery is the ideal action the file system can perform. If recovery is not possible, repair is an alternative approach for continuing execution. If a file operation fails due to corruption, the file system is expected to report an error.

5.2. Visualization of Results

We now describe the visualization in Figure 1. In the two figures, each row presents the results of corrupting one pointer (*e.g.*, `Boot-MFT0`). Every row is divided into 27 columns, each corresponding to different $Target_{corrupt}$ values used to corrupt the pointer (*e.g.*, LogData). Each cell is marked with a symbol representing our observations when the pointer for its row is corrupted with the column value. A dot before pointer name indicates that some form of redundancy exists for the pointer or for $Target_{original}$.

We provide an example from Figure 1a to illustrate the interpretation of the figures. The results of corrupting `Boot-MFT0` is presented in the first row. The first cell corresponds to the boot sector (Boot). The symbol in the cell corresponds to "Detects and recovers." This indicates that when the pointer `Boot-MFT0` is corrupted to the value Boot, NTFS detects the corruption and fully recovers from it, thus continuing normal operation. The value MFT0 (column 6) is the correct value for the pointer and hence the "Not applicable" symbol is used. Note that there is no similar correct value for pointers like `FileData` since we can use data locations of a *different* file to corrupt the pointer.

5.3. NTFS Behavior

We discuss the behavior of NTFS when each of its pointers are corrupted. The detailed results are presented in Figure 1a and Table 5. Table 6 summarizes these results. This subsection distills the results into higher-level observations on system behavior and lessons to be learned. The goal is to analyze whether NTFS effectively uses its type information and redundancy, and to understand why NTFS is or is not able to detect and recover from pointer corruption.

Out of 360 corruption experiments, NTFS detects corruption in 238 cases (66%) and recovers in only 51 cases (14%). Despite the availability of redundant information for recovery for most cases, NTFS either simply reports an error to the user or retries the mount operation. Also, despite detecting the corruption, NTFS itself causes further corruption in 42 cases (12%).

5.3.1. Detection

From our experiments, we find that NTFS uses type checking and sanity checking to detect pointer corruption. We discuss each of these techniques below.

Type checking verifies that a disk cluster conforms to the requirements for a data type. Typically, type information for a cluster is encoded in the form of a "magic" number and stored in the cluster. In order to perform type checking, the cluster pointed to should be read.

Sanity checking verifies that certain values in data structures follow constraints. A pointer can be compared with well-known values, such as locations of metadata like the boot sector or disk partition size, to ensure that the pointer is not corrupt. In this case, corruption can be detected even before the cluster pointed to is read.

Observation 1 *NTFS detects corruption errors primarily through type checking.*

We observe that NTFS detects corruption errors *after reading* $Target_{corrupt}$ for many pointers, including `Boot-MFT0`, `MFT0-MFT`, `LogFile`, `RootIndxBuf`, `SII`, and `DirIndxBuf`. An examination of the corresponding data structures shows that they contain "magic" numbers ("FILE" for MFT clusters, "RSTR" for log restart area, "INDX" for index buffers) that identify the clusters as a certain data type.

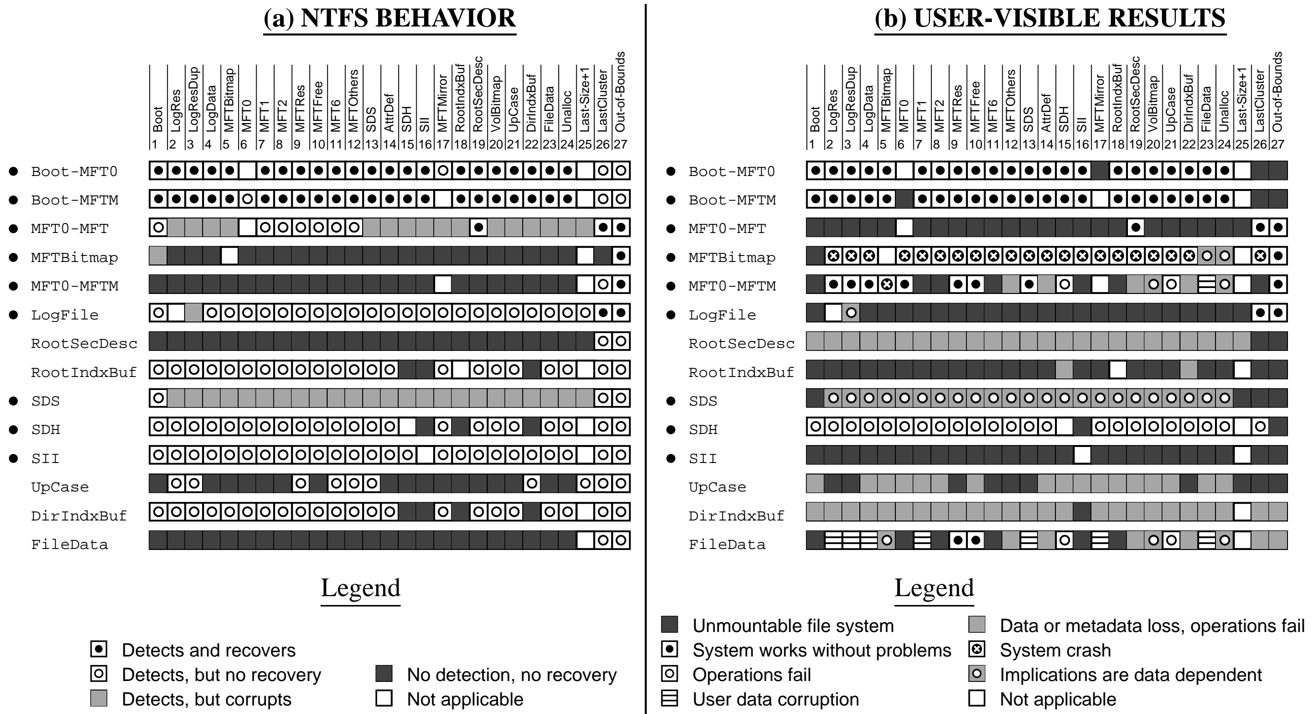

Figure 1. NTFS Corruption Behavior and Implications. *These figures present (a) the corruption behavior of NTFS, and (b) the implications of this behavior for the user. Each row (horizontal strip) characterizes the behavior for the given pointer. Each cell in a row is marked with the corruption behavior/implications observed for the given pointer when it is corrupted with the value of that column. Of the different values, Last-Size+1 denotes Last Cluster - Size of data run + 1 and is applicable only for data runs of length greater than 1. A large dot next to a pointer name for any row implies that some form of redundancy exists; in the ideal case NTFS would be able to recover from any corruption to these pointers. Note that in the case of unallocated clusters, further corruption just implies that the cluster is overwritten since, by definition, the cluster cannot be "corrupted".*

Lesson 1 *Type checking is useful for detecting pointer corruption. However, systems that use type checking should not overload the data types.*

NTFS does not detect corruption when one index buffer pointer (`RootIndxBuf`, `SDH`, `SII`, or `DirIndxBuf`) points to a wrong index buffer. In this case, the type "INDX" is *overloaded*; it is used to represent different data structures used for different purposes. Not detecting corruption in these cases leads to further corruption by NTFS. Thus, when a data type is used for different purposes in different places, it must be assigned a different type identifier to prevent corruption across uses.

Pitfall 1 *Inadequate / inconsistent use of sanity checks.*

We observe that NTFS detects corruption to any pointer with an out-of-bounds value without reading $Target_{corrupt}$. Similarly, the corruption is detected immediately when `Boot-MFTM` is assigned the value MFT0 (Row 2, column 6 in Figure 1a). These immediate detections indicate the use of sanity checks. However, while NTFS detects the above corruption scenario where pointers `Boot-MFTM` and `Boot-MFT0` are equal, it allows `MFT0-MFTM` and `MFT0-MFT` to be equal (Row 5, column 6 in Figure 1a), although the $Target_{original}$ for each pointer is the same as before. This difference in behavior points to the lack of a consistent approach. There are more examples of inconsistencies – pointers for which some corruptions are recovered from, while others are not even detected.

Lesson 2 *Type checks do not work for all pointers. Therefore, detailed sanity checks should be performed.*

Type checking is not useful for pointers like `FileData` since a type identifier cannot be stored in a user data cluster. In these cases, sanity checking assumes greater significance. However, NTFS does not perform many simple sanity checks that can determine whether a pointer is corrupt. For example, NTFS does not check whether a pointer is pointing to the boot sector (Boot).

We note that not all NTFS behavior can be explained based on sanity or type checking. NTFS detects corruption of `UpCase` after reading $Target_{corrupt}$ for some experiments but does not detect for others. It is not clear what kind of check is used for this pointer.

5.3.2. Reactions

NTFS reacts in various ways on detecting corruption. It either recovers from corruption, or reports an error to the

Pointer	NTFS Behavior Details
Boot-MFT0	**Reports** error and **retries** mount for values MFTMirror, LastCluster and Out-of-bounds; **recovers** using replica for others.
Boot-MFTM	**Reports** error and **retries** mount for values MFT0, LastCluster and Out-of-bounds; **recovers** using replica for others.
MFT0-MFT	**Recovers** using MFT mirror for values RootSecDesc, LastCluster and Out-of-bounds; **reports** error and **retries** mount for others – however, both $Target_{corrupt}$ and the replica (MFT Mirror) are corrupted if the value is *not* an MFT entry or Boot.
MFTBitmap	**Recovers** only for an out-of-bounds value; **reports** error for the value Boot (however, NTFS corrupts Boot); does not detect all other cases corrupting $Target_{corrupt}$ and possibly an MFT entry.
MFT0-MFTM	**Recovers** for an out-of-bounds value; **reports** error for LastCluster; does not detect all other cases and corrupts $Target_{corrupt}$. Interestingly, this corruption of $Target_{corrupt}$ is reversed for LogRes and LogResDup due to the order of disk operations.
LogFile	**Recovers** for an out-of-bounds value or LastCluster; attempts **repair** but corrupts clusters for LogResDup; reports error and retries for others but corrupts the replica of the pointer in MFT mirror.
RootSecDesc	**Reports** error and **retries** mount for values LastCluster and Out-of-bounds; other cases are undetected.
RootIndxBuf	**Reports** error and **retries** mount for all values except for other index buffers (SDH, SII or DirIndxBuf) which go undetected thus corrupting $Target_{corrupt}$.
SDS	**Reports** error and **retries** for Boot, LastCluster Last-Size+1 and out-of-bounds (For Last-Size+1, report and retry occur after corrupting it); attempts to **repair** data structure for other cases, resulting in corruption of $Target_{corrupt}$.
SDH	**Reports** and **retries** during mount for an out-of-bounds value; **reports** error during CreateFile for other values except for index buffers (SII, RootIndxBuf and DirIndxBuf) which go undetected thus corrupting $Target_{corrupt}$.
SII	**Reports** and **retries** mount for all values.
UpCase	**Reports** error and **retries** mount for the 10 detected cases (refer Figure 1); undetected cases do not cause further corruption.
DirIndxBuf	**Reports** an error for all values except for other index buffers (these go undetected, thus corrupting $Target_{corrupt}$).
FileData	**Reports** an error for values Last Cluster and out-of-bounds; others are not detected leading to corruption of $Target_{corrupt}$. The corruption is reversed for LogRes, LogResDup, MFT0, and MFTMirror due to the order of disk operations.

Table 5. NTFS Behavior Details. *The table presents the details of NTFS behavior when its pointers are corrupted.*

application, or retries the mount operation, or attempts to repair a seemingly corrupt data structure.

Observation 2 *NTFS typically uses replication to recover from corruption.*

We observe that NTFS uses replication of MFT VCN 0 to recover from corruption to the pointer Boot-MFT0. In this case, it uses the MFT mirror to obtain the required information. Similarly, NTFS uses redundant information in MFT VCN 0 to recover from corruption to Boot-MFTM. Interestingly, for both pointers, this recovery is *temporary*; that is, NTFS does not overwrite the corrupt pointer with the correct value. Thus, the same recovery has to be performed for each mount. This approach could lead to unrecoverable data loss in the event of a second failure (loss or corruption). When an out-of-bounds value is used for the pointers MFT0-MFT, MFTBitmap, MFT0-MFTM, and LogFile, NTFS performs *permanent* recovery; that is, the pointer value is overwritten with the correct value, thus completely healing the file system image.

Observation 3 *NTFS uses error reporting and retries in response to corruption when it is unable to recover.*

As described in Table 5, typically, NTFS reports an error to the application when corruption is detected. For a subset of cases, NTFS also retries the mount operation, perhaps hoping that the corruption is transient and mount will succeed the second time. These retries do not succeed since the corruption is persistent. Examples of pointers for which this behavior is observed include MFT0-MFT and LogFile.

Observation 4 *NTFS attempts to repair certain data structures that it believes to be corrupt.*

When the pointer SDS is corrupted, NTFS assumes that the security descriptors pointed to by SDS are corrupt and attempts to reinitialize the data structure, thus corrupting $Target_{corrupt}$. Similar behavior occurs when LogFile points to LogResDup instead of LogRes (the log restart area). In this case, the first cluster of the data region of the log is corrupted.

Pitfall 2 *Detecting that a pointer target is corrupt instead of detecting that the pointer is corrupt.*

The instances under Observation 4 above show that NTFS trusts the pointer to be correct, while not trusting the cluster pointed to. Thus, attempting to repair a seemingly corrupt target causes more harm than good if the corruption is actually to the pointer.

In general, we observe that there are multiple instances where NTFS does not detect the corruption or detects the corruption but does not recover from it despite possessing type information to detect corruption and redundancy to recover from corruption. Table 6 shows that despite possessing redundant information, NTFS detects an error but does not recover from it in 87 cases, and in fact, causes further corruption in 88 cases. From these failures, we derive more potential pitfalls when handling pointer corruption.

Pitfall 3 *Ineffective replica management: (a) not using replicas when available, (b) destroying secondary replicas without verifying the primary, and (c) not maintaining independent access paths for replicas.*

(a) When pointers in MFT VCN 0 are corrupted, NTFS does not use the copy of pointers available in the MFT mirror for most scenarios. For some pointers, NTFS could but does

Pointer	Redundancy?	Detects & Recovers	Detects, but no recovery	Detects, but corrupts	No detection	Further corruption	Replica destroyed
`Boot-MFT0`	√	22	3				
`Boot-MFTM`	√	22	3				
`MFT0-MFT`	√	3	7	16		16	16
`MFTBitmap`	√	1		1	23	24	24
`MFT0-MFTM`	√	1	1		23	20	
`LogFile`	√	2	23	1		1	24
`RootSecDesc`			2		25		
`RootIndxBuf`			22		3	3	
`SDS`	√		3	24		24	
`SDH`	√		22		3	3	
`SII`	√		25				
`UpCase`			10		17		
`DirIndxBuf`			22		4	4	
`FileData`			2		24	20	
Total		51	145	42	122	115	64
Total recoverable	√	**51**	**87**	**42**	**49**	**88**	**64**

Table 6. NTFS Behavior Summary. *The table summarizes observed NTFS behavior on corruption for the different pointers. The first column indicates whether some form of redundancy exists for either the pointer or* $Target_{original}$*. Columns 2 to 5 summarize the number of cases for which NTFS behaves in a certain manner (from Figure 1). The last two columns indicate the total number of cases for which further corruption occurs and for which the replica of the pointer is destroyed. The penultimate row is the sum of all rows and the last row is the sum of rows that have a √ for the "Redundancy?" column.*

not use the replica for comparing and detecting that the pointer is possibly corrupt. An example is `MFTBitmap`. For other pointers, NTFS detects corruption through different means (type or sanity checking). However, NTFS does not use the replica for recovering from the corruption. (b) There are 64 instances where the replica of the pointer is overwritten by NTFS with the corrupt value (the last column of Table 6). In particular, in the cases where the primary MFT (MFT VCN 0) is corrupt, but the MFT mirror is correct, NTFS erroneously synchronizes the two copies by overwriting the MFT mirror with data in the corrupt MFT. (c) For some of the data structures in NTFS, the replica is placed at a fixed virtual offset from the regular copy, thus often using a single pointer value to access both. The security descriptors are an example. Corruption to the pointer `SDS` will thus make both the regular copy and the replica inaccessible (Figure 1a shows that NTFS does not recover when `SDS` is corrupted).

Pitfall 4 *Not realizing that most indexes are simply performance improvements and that their unavailability should not cause complete failure.*

NTFS uses two indexes SDH and SII for its security descriptors in $Secure. The security descriptors contain all information necessary to rebuild both the indexes. However, when either `SDH` or `SII` is corrupted, NTFS does not recover despite detecting the corruption.

5.4. User-Visible NTFS Results

The previous subsection detailed NTFS behavior in response to pointer corruption. However, understanding these actions does not imply an understanding of how they manifest to users or applications. The primary concern for users is data and system reliability. Hence, in this subsection, we discuss user-visible results of NTFS behavior. Figure 1b presents the user-visible results.

Observation 5 *The system works correctly when NTFS recovers from corruption.*

The system works without problems in 61 scenarios (17%), primarily because NTFS detects and recovers from corruption. For example, corruption of any one pointer field (MFT, MFTMirror) in the boot sector does not affect normal operation. In 10 other cases, even though NTFS does not recover, pointer corruption does not cause problems due to the order of disk operations or due to non-use of $Target_{corrupt}$.

Observation 6 *The most frequent user-visible result is an unmountable file system.*

The file system becomes unmountable when NTFS detects corruption to a pointer used during mount, but is unable to recover. This situation applies to many pointers across the range of values used. An example of such a pointer is `LogFile`. The file system could also become unmountable when undetected pointer corruption (*e.g.*, for `FileData`) causes key data structures to be corrupted. The file system is rendered unmountable in 133 scenarios (37%).

Observation 7 *Other user-visible results include: (a) loss of data or user-visible metadata, (b) failure of many file operations, and (c) corruption of user file data.*

(a) Data or metadata loss occurs in 102 scenarios (28%). Data is rendered inaccessible when the pointers `DirIndxBuf`, `RootSecDesc`, `SDS`, and `UpCase` are corrupted. (b) For some corruption scenarios, file operations fail since NTFS does not recover from the pointer corruption. An example is corruption to `SDH`; attempts to create files fail while files already created can be accessed. Note that operations also fail when data or metadata is lost. In total, file operations fail in 127 scenarios (35%). (c) User data corruption occurs in 8 scenarios (2%), when user file data is overwritten with other data or metadata, e.g., when a file data pointer points to another file's data clusters.

Lesson 3 *Undetected pointer corruption can pose a significant security risk.*

One would expect that pointer corruption might affect data on a particular disk. However, it could be worse; most experiments involving the pointer `MFTBitmap` result in a system crash (22 cases), thus affecting the entire system. By systematically setting bits contained in $Target_{corrupt}$

Pointer	Redundancy?	Detects & Recovers	Detects, but no recovery	Detects, but corrupts	No detection	Further corruption	Replica destroyed
Block bitmap	√		1		12	12	
Inode bitmap	√		5		8	8	
Inode table	√		13				
Journal superblock			13				
Root directory			11		2	2	
Directory data			11		3	3	
File data			1		13	13	
Total		0	55	0	38	38	0
Total recoverable	√	**0**	**19**	**0**	**20**	**20**	**0**

Table 7. Ext3 System Behavior Summary. *The table summarizes observed ext3 behavior on corruption. The columns are the same as in Table 6.*

(the disk block being pointed to after corruption), we observe that the system crash happens whenever the allocation status bits corresponding to the system files $Quota, $ObjId and $Reparse happen to be zero (instead of one), resulting in their MFT entries getting re-used (and hence corrupted). Thus, a particular series of operations (mount, `CreateFile`) can be performed on specifically corrupted file system images to cause crashes. Such malicious disk images [29] could become a security threat with the use of portable flash drives and disk image downloads.

In certain pointer corruption scenarios, the user-visible results depend on the actual data present in various clusters. Corrupting `MFTBitmap` with the location of a file data cluster (FileData) is an example. Depending on the exact values of bits in the file data cluster, there may be a system crash, or data might be lost.

5.5. Ext3 Results

We corrupt 7 primary ext3 pointers with 14 values each, chosen in similar fashion to NTFS. Table 7 presents a summary of ext3 results.

- Unlike NTFS, ext3 relies more on sanity checks than on type checks. For example, it verifies that bitmap and inode table pointers point within the block group. Also, when allocating inodes ext3 verifies that the inode bitmap has marked “reserved” inodes as allocated, unlike NTFS' (mis)handling of `MFTBitmap`. However, lack of type checks causes ext3 to use the superblock as directory data.
- Like NTFS, ext3 typically assumes that the cluster pointed to (rather than the pointer) is corrupt.
- Even though ext3 replicates the group descriptors, it never uses these replicas even when a pointer in the primary copy is detected as corrupt.
- The typical reaction on detecting corruption is to report an error and remount the file system as read-only. Ext3 does not recover even in one corruption scenario.

In summary, our analysis of ext3 shows that it is no better than NTFS in pointer protection. Our analysis also demonstrates that TAPC can be applied to very different file systems. One advantage with ext3 is that we have verified our results by reading ext3 source code.

5.6. Discussion

Using TAPC to characterize system behavior yields many lessons for handling corruption. If NTFS and ext3 follow these lessons, they can completely recover from over 55% and 40% corruption scenarios respectively. We discuss general issues related to TAPC and corruption handling.

First, TAPC does not consider the likelihood of different values used for corruption. This likelihood depends on the source of corruption. For example, if the corruption values are arbitrary, more than 99% of the values will be out-of-bounds, while corruption due to bit flips will imply that the corrupt value is “closer” to the correct value. While our likelihood-agnostic approach does not provide probabilities for file system failures due to corruption, it provides interesting insights into how a file system handles corruption.

Second, a question that arises from the results is whether type and sanity checks are the right techniques to use, especially when there are many pitfalls involved. While it is true that the use of checksums (like in ZFS [8]) might significantly improve corruption handling, it does not subsume the protection offered by type and sanity checks. For example, checksums cannot protect against file system bugs that place the wrong pointer value and checksum it as well.

Third, it is non-trivial to add checksums and other protection to a file system without changing the on-disk format. Type-aware pointer corruption helps identify potential sanity checks that can be used without format changes.

6. Related Work

Software fault injection: A multitude of software fault-injection techniques and frameworks have been developed over the years [6, 9, 15, 16, 17, 27]. The FTAPE [27] framework is most related to our work – it consists of a workload generator and a device-driver-level disk-fault injector. The fault-injection frameworks and techniques have been employed in various studies of real systems. For example, Gu et al. [14] examine the behavior of the Linux kernel when errors are injected into the instruction stream.

File system studies: Recent research efforts [29, 30] have used static-analysis and model-checking techniques instead of fault injection to extract bugs in file-system code. Our study is also related to previous fault-injection-based failure-behavior analyses [2, 18] from our research group. These analyses use type information for fault injection in order to understand the behavior of systems for disk errors and randomly-corrupted disk blocks, while this paper examines the effects of corrupt pointers and analyzes NTFS in detail; indeed, we obtain new insights into file-system behavior. Type-aware pointer corruption and some initial NTFS experiments are discussed in our position paper [4].

Pointer integrity: Research efforts have looked at protecting systems from pointer errors. Particularly related are research on data-structure redundancy [25] and data protection in highly-available systems using checksums [5]. Various file systems have been built to protect data and metadata using checksums [8, 18, 23]. It would be interesting to use TAPC on Sun ZFS [8] to understand the tricky details of using checksumming. Another related effort is on *type-safe* disks [20] which ensure that file systems do not use corrupt on-disk pointers to access data.

7. Conclusion

File systems rely on on-disk pointers to access data. As file systems employ different and newer techniques to protect against corrupt pointers, we need to understand how these techniques perform in reality.

We develop type-aware pointer corruption as a way to rapidly and systematically analyze the corruption-handling capability of file systems. We apply type-aware pointer corruption to NTFS and ext3, and find that despite their potential to recover from many pointer-corruption scenarios, they do not, causing data loss, unmountable file systems, and system crashes. We use this study to learn important lessons on how to handle corrupt pointers.

We believe that future file systems should be more careful in implementing pointer protection techniques. A first step would be to develop a consistent corruption-handling policy and the corresponding machinery that can be used by all file system components.

Acknowledgments

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance and Sun Microsystems. We would like to thank our shepherd Marco Vieira and the anonymous reviewers for their detailed comments that helped improve the paper. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface: SCSI vs. ATA. In *FAST '03*, Apr. 2003.

[2] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Dependability Analysis of Virtual Memory Systems. In *DSN '06*, June 2006.

[3] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST'08*, 2008.

[4] L. N. Bairavasundaram, M. Rungta, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Limiting Trust in the Storage Stack. In *StorageSS'06*, June 2006.

[5] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Trans. on Dependable and Secure Computing*, 1(1):87–96, Jan. 2004.

[6] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4):1105–1118, April 1990.

[7] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[8] J. Bonwick. ZFS: The Last Word in File Systems. http://www.opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[9] J. Carreira, H. Madeira, and J. G. Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Trans. on Software Engg.*, 1998.

[10] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. E. gler. An Empirical Study of Operating System Errors. In *SOSP '01*.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. C. helf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01*.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP '03*, Bolton Landing, NY, October 2003.

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

[14] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux Kernel Behavior Under Error. In *DSN '03*, pages 459–468, San Francisco, CA, June 2003.

[15] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems. In *Proceedings of IPDS'95*.

[16] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computing*, 44(2), 1995.

[17] W. lun Kao, R. K. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. In *IEEE Transactions on Software Engineering*, pages 1105–1118, 1993.

[18] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*.

[19] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. L. a nd Andy Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *MASCOTS '04*.

[20] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-Safe Disks. In *OSDI '06*, Seattle, WA, November 2006.

[21] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *FAST '03*.

[22] D. A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, 1998.

[23] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *USENIX '01*, June 2001.

[24] R. Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html, February 2006.

[25] D. J. Taylor, D. E. Morgan, and J. P. Black. Redundancy in Data Structures: Improving Software Fault Tolerance. *IEEE Trans. on Software Engg.*, 6(6), 1980.

[26] The Data Clinic. Hard Disk Failure. http://www.dataclinic.co.uk/hard-disk-failures.htm, 2004.

[27] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *MASCOTS '95*.

[28] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The 4th Annual Linux Expo*, Durham, NC, May 1998.

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

[30] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.