Ensuring Data Integrity in Storage: Techniques and Applications*

Gopalan Sivathanu, Charles P. Wright, and Erez Zadok
Stony Brook University
Computer Science Department
Stony Brook, NY 11794-4400
{gopalan,cwright,ezk}@cs.sunysb.edu

ABSTRACT

Data integrity is a fundamental aspect of storage security and reliability. With the advent of network storage and new technology trends that result in new failure modes for storage, interesting challenges arise in ensuring data integrity. In this paper, we discuss the causes of integrity violations in storage and present a survey of integrity assurance techniques that exist today. We describe several interesting applications of storage integrity checking, apart from security, and discuss the implementation issues associated with techniques. Based on our analysis, we discuss the choices and tradeoffs associated with each mechanism. We then identify and formalize a new class of integrity assurance techniques that involve *logical redundancy*. We describe how logical redundancy can be used in today's systems to perform efficient and seamless integrity assurance.

Categories and Subject Descriptors

B.8.0 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**Operating Systems**]: File Systems Management; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*

General Terms

Reliability, Security

Keywords

Storage integrity, File systems, Intrusion detection

1. INTRODUCTION

Reliable access to data is a prerequisite for most computer systems and applications. There are several factors that cause unexpected or unauthorized modifications to stored data. Data can get corrupted due to hardware or software malfunctions. Disk errors are common today [26] and storage software that exists is typically not designed to handle a large class of these errors. A minor integrity violation, when not detected by the higher level software on time, could cause further loss of data. For example, a bit-flip while reading a file system inode bitmap could cause the file system to overwrite an important file. Therefore, prompt detection of integrity violations is vital for the reliability and safety of the stored data.

Integrity violations could also be caused by malicious intrusions. Security advisory boards over the last few years have noticed a steep rise in the number of intrusion attacks on systems [4]. A large class of these attacks are caused by malicious modifications of disk data. An attacker that has gained administrator privileges could potentially make changes to the system, like modifying system utilities (e.g., `/bin` files or daemon processes), adding back-doors or Trojans, changing file contents and attributes, accessing unauthorized files, etc. Such file system inconsistencies and intrusions can be detected using utilities like Tripwire [17, 18, 39].

There are different notions of integrity in storage. File system consistency is one of the common ones. Most file systems today come with integrity checking utilities such as the Unix `fsck` that perform a scan through the storage device to fix logical inconsistencies between data and meta-data. (Tools such as `fsck` are often said to be performing "sanity" checking.) This reduces the likelihood of file corruption and wasted disk space in the event of a system crash. Advanced methods like journaling [12] and transactional file systems [8] ensure file system consistency even in the event of unexpected system faults. File system inconsistency can cause data corruption, but generally may not cause security threats; files might become inaccessible due to inconsistency between the meta-data and data caused by a system crash. Apart from file-system inconsistencies, integrity violations in file data are a major problem that storage system designers have to solve. Even a perfectly consistent file system can have its data corrupted, and normal integrity checkers like `fsck` cannot detect these errors. Techniques like mirroring, parity, or checksumming can be used to detect data integrity violations at the file or block level. Cryptographic hash functions could even detect malicious forging of checksums.

In this paper, we begin by presenting a survey of integrity assurance techniques, classifying them under three different dimensions: the scope of integrity assurance, the logical layer of operation, and the mode of checking. We then discuss the various applications of integrity checking such as security, performance enhancement, etc.

*This work was partially made possible by NSF CAREER EIA-0133589 and CCR-0310493 awards and HP/Intel gifts numbers 87128 and 88415.1.

StorageSS'05, November 11, 2005, Fairfax, Virginia, USA.

We also describe the different implementation choices of integrity assurance mechanisms. Almost all integrity checking mechanisms that we analyzed adopt some form of redundancy to verify integrity of data. Techniques such as checksumming, parity etc., ignore the semantics of the data and treat it as a raw stream of bytes. They explicitly generate and store redundant information for the sole purpose of integrity checking. In contrast to these *physical redundancy* techniques, we identify a new class of techniques for integrity assurance, where the redundant information is dependent on the semantics of the data stored. Such *logical redundancy* techniques often obviate the extra cost of explicitly accessing redundant data and verifying integrity, by exploiting structural redundancies that already exist in the data. For example, if an application stored a B+ tree on disk, with back pointers from the children to parents (for more efficient scanning), those back pointers can also be used to ensure the integrity of pointers within a node. Although some existing systems perform a minimal amount of such logical integrity checking in the form of "sanity checks" on the structure of data, we believe that these techniques can be generalized into first class integrity assurance mechanisms.

The rest of this paper is organized as follows. Section 2 describes causes of integrity violations. Section 3 describes the three most commonly used integrity checking techniques. Section 4 presents a more detailed classification of such techniques under three different dimensions. Section 5 explores several interesting applications of integrity checking. We discuss the various implementation choices for integrity checkers in Section 6. In Section 7 we present the new class of integrity assurance techniques that make use of logical redundancy. We conclude in Section 8.

2. CAUSES OF INTEGRITY VIOLATIONS

Integrity violations can be caused by hardware or software malfunctions, malicious activities, or inadvertent user errors. In most systems that do not have integrity assurance mechanisms, unexpected modifications to data either go undetected, or are not properly handled by the software running above, resulting in software crash or further damage to data. In this section, we describe three main causes of integrity violations and provide scenarios for each cause.

2.1 Hardware and Software Errors

Data stored on a storage device or transmitted across a network in response to a storage request, can be corrupted due to hardware or software malfunctioning. A malfunction in hardware could also trigger software misbehavior resulting in serious damage to stored data. For example, a hardware bit error while reading a file system's inode bitmap could cause the file system to overwrite important files. Hardware errors are not uncommon in today's modern disks. Disks today can corrupt data silently without being detected [1]. Due to the increasing complexity of disk technology these days, new errors occur on modern disks—for example, a faulty disk controller causing misdirected writes [40] where data gets written to the wrong location on disk. Most storage software systems are totally oblivious to these kind of hardware errors, as they expect the hardware to be fail-stop in nature—where the hardware either functions or fails explicitly.

Bugs in software could also result in unexpected modification of data. Buggy device drivers can corrupt data that is read from the storage device. File system bugs can overwrite existing data or make files inaccessible. Most file systems that adopt asynchrony in writes could end up in an inconsistent state upon an unclean system shutdown, thereby corrupting files or making portions of data inaccessible.

The ever growing requirements for storage technology has given rise to distributed storage where data need to be transferred through unreliable networks. Unreliable networks can corrupt data that pass through them. Unless the higher level protocols adopt appropriate error checking and correcting techniques, these errors can cause client software to malfunction.

2.2 Malicious Intrusions

Trustworthy data management in a computer system is an important challenge that hardware and software designers face today. Although highly critical and confidential information is being stored electronically, and is accessed through several different interfaces, new security vulnerabilities arise. For example, in a distributed storage system, data can be accessed from remote locations through untrusted network links; a network eavesdropper can gain access to confidential data if the data is not sufficiently protected by methods such as encryption. Damage to data integrity can often cause more serious problems than confidentiality breaches: important information may be modified by malicious programs or malicious users, or faulty system components. For example, virus code could be inserted into binary executables, potentially resulting in the loss of all data stored on a system. Operating systems that allow access to raw disks can inadvertently aid an attacker to bypass security checks in the file system, and cause damage to stored data.

2.3 Inadvertent User Errors

User errors can compromise data integrity at the application level. A user action can break application level integrity semantics. For example, an inadvertent deletion of a database file can cause a DBMS to malfunction, resulting in data corruption. In general, if user actions invalidate the implicit assumptions that the applications dealing with the data make, integrity violations can occur.

3. COMMON INTEGRITY TECHNIQUES

In this section, we discuss the three most common integrity assurance techniques that exist today in storage. All these techniques maintain some redundant information about the data and ensure integrity by recomputing the redundant data from the actual data and comparing it with the stored redundant information.

3.1 Mirroring

One simple way to implement integrity verification is data replication or *mirroring*. By maintaining two or more copies of the same data in the storage device, integrity checks can be made by comparing the copies. An integrity violation in one of the copies can be easily detected using this method. While implementing this is easy, this method is inefficient both in terms of storage space and time. Mirroring can detect integrity violations caused by data corruption due to hardware errors, but cannot help in recovering from the damage, as a discrepancy during comparison does not provide information about which of the copies is legitimate. Recovery is possible using majority rules if the mirroring is 3-way or more. Mirroring can be used to detect integrity violations which are caused due to data corruption and generally not malicious modification of data. A malicious user who wants to modify data can easily modify all copies of the data, unless the location of the copies is maintained in a confidential manner. Mirroring also cannot detect integrity violations caused by user errors, because in most cases user modifications are carried out in all mirrors. RAID-1 uses mirroring to improve storage reliability, but does not perform online integrity checks using the redundant data.

3.2 RAID Parity

Parity is used in RAID-3, RAID-4, and RAID-5 [24] to validate the data written to the RAID array. Parity across the array is computed using the XOR (Exclusive OR) logical operation. XOR parity is a special kind of *erasure code*. The basic principle behind erasure codes is to transform N blocks of data, into $N + M$ blocks such that upon loss of any M blocks of data, they can be recovered from the remaining N blocks, irrespective of which blocks are lost. The parity information in RAID can either be stored on a separate, dedicated drive, or be mixed with the data across all the drives in the array. Most RAID schemes are designed to operate on fail-stop disks. Any single disk failure in RAID (including the parity disk) can be recovered from the remaining disks by just performing an XOR on their data. This recovery process is offline in nature. Although the parity scheme in RAID does not perform online integrity checks, it is used for recovering from a single disk failure in the array. The organization of RAID-5 parity is shown in Figure 1.

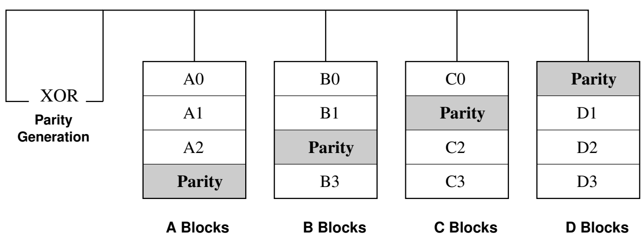

Figure 1: RAID-5: Independent data disks with distributed parity

3.3 Checksumming

Checksumming is a well known method for performing integrity checks. Checksums can be computed for disk data and can be stored persistently. Data integrity can be verified by comparing the stored and the newly computed values on every data read. Checksums are generated using a hash function. The use of cryptographic hash functions has become a standard in Internet applications and protocols. Cryptographic hash functions map strings of different lengths to short fixed size results. These functions are generally designed to be collision resistant, which means that finding two strings that have the same hash result should be infeasible. In addition to basic collision resistance, functions like MD5 [30] and SHA1 [6] also have some properties like randomness. HMAC [19] is a specific type of a hashing function where the hash generated is cryptographically protected. It works by using an underlying hash function over a message and a key, thereby detecting unauthorized tampering of checksum values. It is currently one of the predominant means of ensuring that secure data is not corrupted in transit over insecure channels (like the Internet). This can also be used in the context of storage systems to ensure integrity during read.

4. INTEGRITY CHECKING TAXONOMY

There are several methods used to detect and repair data-integrity violations. Almost all methods that exist today use some kind of redundancy mechanisms to check for integrity. This is because integrity checking requires either syntactic or semantic comparison of the data with some piece of information that is related to the actual data. Comparison of related data helps detect integrity violations, but cannot repair them. The main reason that comparison cannot repair violations is that when there is a mismatch between two different kinds of data being compared, it is usually not possible to determine which of them is legitimate. For example, if checksumming is used to detect integrity violations in file data, a checksum mismatch can only detect an integrity violation, but could not provide information whether the data is corrupted or the checksum itself is invalid. There are a few techniques for recovering from integrity violations once they are detected. Those techniques are closely tied to the mechanism used to perform detection, and the nature of redundancy that is employed.

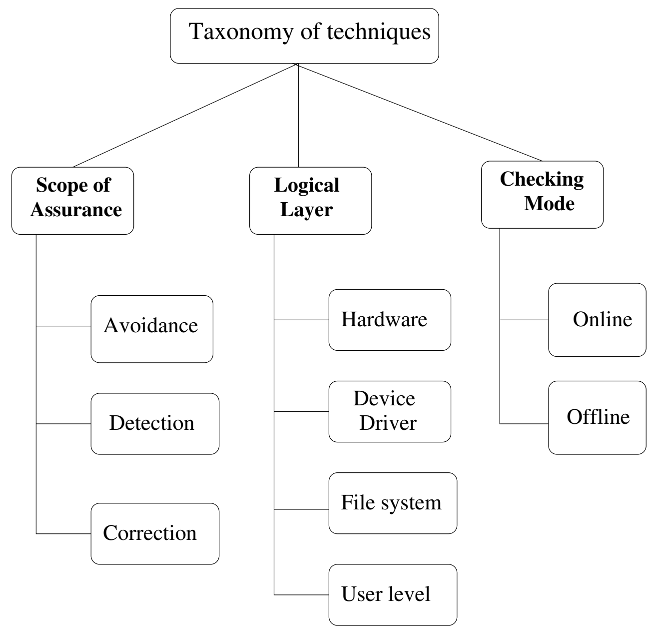

Figure 2: Taxonomy of techniques

In this section we describe the taxonomy of integrity checking techniques that exist today. Figure 2 represents our classification of integrity assurance techniques. We analyze the techniques in three different dimensions: the scope of their assurance, the logical layer at which they are designed to operate, and their checking modes.

4.1 Scope of Integrity Assurance

Data integrity can be guaranteed by several ways. Online integrity checks help to detect and in some cases recover from integrity violations. Some systems, instead of performing checks for integrity, employ *preventive methods* to reduce the likelihood of an integrity violation. In this section we classify integrity assurance mechanisms into three main types, based on their goals: those that perform preventive steps so as to avoid specific types of integrity violations; those that perform integrity checks and detect violations; and those that are capable of recovering from damage once a violation is detected.

4.1.1 Avoidance

Some systems provide a certain level of integrity guarantee for the data they store, so as to avoid explicit integrity checking mechanisms. These systems come with an advantage that they do not incur additional overheads for integrity verification. The mechanism used to provide integrity guarantees could incur some overhead, but it is generally smaller than separate checking mechanisms. Moreover, these systems avoid the hassle of recovering from an integrity damage once it is detected. In this section we discuss four existing methods that provide different levels of integrity assurances. Read-only storage is a straight-forward means to avoid integrity violations due to malicious user activity or inadvertent user errors. Journaling ensures file system consistency, encryption file systems prevent malicious modification of file data with say, virus code, and transactional file systems provide ACID transactions which applications can use to ensure semantic integrity of information.

Read-only Storage. Making the storage read-only is a simple means to ensure data integrity. The read-only limitation can be imposed at the hardware or software level. Hardware level read-only storage is not vulnerable to software bugs or data modification through raw disk access. However, they are vulnerable to hardware errors like bit-flipping. File systems that enforce read-only characteristics are still vulnerable to hardware and software errors, and to raw disk access. However, they can prevent integrity violations due to user errors. SFSRO [7], Venti [27], and Fossilization [14] are systems are read-only to ensure data integrity.

Journaling. Journaling file systems were invented partly to take advantage of the reliability of logging. Modern examples include Ext2, NTFS, Reiserfs, etc. A journaling file system can recover from a system crash by examining its log, where any pending changes are stored, and replaying any operations it finds there. This means that even after an unexpected shutdown, it is not necessary to scan through the entire contents of the disk looking for inconsistencies (as with `scandisk` on Windows or `fsck` on Unix): the system just needs to figure out whatever has been added to the journal but not marked as done. In a journaling file system, the transaction interface provided by logging guarantees that either all or none of the file system updates are done. This ensures consistency between data and meta-data even in the case of unexpected system failures. Although journaling cannot protect data from malicious modifications or hardware bit errors, it can ensure file system consistency without performing any explicit integrity checks for each file.

Cryptographic File Systems. Cryptographic file systems encrypt file data (and even selected meta-data) to ensure the confidentiality of important information. Though confidentiality is the main goal of encryption file systems, a certain degree of integrity assurance comes as a side effect of encryption. Unauthorized modification of data by malicious programs or users, such as replacing system files with Trojans, becomes nearly impossible if the data is encrypted with an appropriate cipher mode. Although data can be modified, it is not feasible to do it in a predicable manner without the knowledge of the encryption key. However, integrity violations due to hardware errors cannot be prevented by using encryption. Thus, cryptographic file systems provide protection of integrity for a certain class of threat models. Several file systems such as Blaze's CFS [3], NCryptfs [41, 42], etc., support encryption.

Transactional File Systems. We are working towards building a transactional file system that exports an ACID transaction facility to the user level. In addition to supporting custom user-level transactions for protecting the semantic integrity of data that applications see, our ACID file system aims at providing intra-operation transactions (e.g., an individual `rename` operation is transaction protected), such that atomicity and consistency guarantees are provided for every file system operation. This maintains the file system in a consistent state and makes the file system completely recoverable to a consistent state even in the event of unexpected system failures. We are planning to build our file system using the logging and locking features provided by the Berkeley Database Manager [32] in the Linux kernel [15].

4.1.2 Detection

Most of the storage integrity assurance techniques that exist today perform detection of integrity violations, but do not help in recovering from the violation. In this section, we discuss those techniques.

Checksumming. The checksumming techniques discussed in Section 3 help in detecting integrity violations. They generally cannot help recovery for two reasons. First, a mismatch between the stored value and the computed value of the checksums just means that one of them was modified, but it does not provide information about which of them is legitimate. Stored checksums are also likely to be modified or corrupted. Second, checksums are generally computed using a one-way hash function and the data cannot be reconstructed given a checksum value.

Mirroring. The mirroring technique described in Section 3 can detect a violation by comparing the copies of data, but suffers from the same problem as checksumming for correcting the data.

CRC. Cyclic Redundancy Check (CRC) is a powerful and easily implemented technique to obtain data reliability in network transmissions. This can be employed by network storage systems to detect integrity violations in data transmissions across nodes. The CRC technique is used to protect blocks of data called *frames*. Using this technique, the transmitter appends an extra N-bit sequence to every frame, called a Frame Check Sequence (FCS). The FCS holds redundant information about the frame that helps the transmitter detect errors in the frame. The CRC is one of the most commonly used techniques for error detection in data communications.

Parity. Parity is a form of error detection that uses a single bit to represent the odd or even quantities of '1's and '0's in the data. Parity usually consists of one parity bit for each eight bits of data, which can be verified by the receiving end to detect transmission errors. Network file systems benefit from parity error detection, as they use the lower level network layers, which implement parity mechanisms.

4.1.3 Correction

When an integrity violation is detected by some means, some methods can be used to recover data from the damage. We discuss two in this section.

Majority Vote. The majority vote strategy helps to resolve the problem of determining whether the actual data or the redundant information stored is unmodified (legitimate), in the event of a mismatch between both of them. The majority vote technique can be employed with detection techniques like mirroring. When there are N copies of the data ($N > 2$), upon an integrity violation, the data contained in the majority of the copies can be believed to be legitimate, to some level of certainty. The other copies can then be recovered from the content in the majority of the copies.

RAID Parity. RAID parity (e.g., RAID 5) uses an erasure code to generate parity information at the block level or bit level. The individual disks in RAID should be fail-stop in nature, which means that in the event of a failure of some means, the disks should stop working thereby explicitly notifying the RAID controller of a problem. A failure in the disk can mean several things including a violation in the integrity of their data. Once a failure (integrity violation) is detected by the fail-stop nature of the hardware, the parity information stored can be used to reconstruct the data in the lost disk. The procedure to reconstruct the data in a RAID array is discussed in Section 3.

4.1.4 Detection and Correction

Error detection and correction algorithms are used widely in network transport protocols. These algorithms combine the functionalities of detection of integrity violations and even correcting them to a certain level. Some of these algorithms are now being used in storage devices also, for detecting and correcting bit errors. In this section we discuss three algorithms employed in local and network storage systems today.

ECC. Error Correction Codes (ECCs) [13] are an advanced form of parity detection often used in servers and critical data applications. ECC modules use multiple parity bits per byte (usually 3) to detect double-bit errors. They are also capable of correcting single-bit errors without raising an error message. Some systems that support ECC can use a regular parity module by using the parity bits to make up the ECC code. Several storage disks today employ error correcting codes to detect and correct bit errors at the hardware level. Usually, to correct an N-bit sequence, at least $lg(N)$ bits of parity information are required for performing the correction part. Hamming codes are one popular class of ECCs [25].

FEC. Forward Error Correction (FEC) [2] is a popular error detection and correction scheme employed in digital communications like cellular telephony and other voice and video communications. In FEC, the correction is performed at the receiver end using the check bits sent by the transmitter. FEC uses the Reed-Solomon algorithm [28] to perform correction. FEC is generally not used in storage hardware, but network file systems that use lower level network protocols benefit from it.

RAID Level 2. RAID-2 [24] uses memory-style error correcting codes to detect and recover from failures. In an instance of RAID-2, four data disks require three redundant disks, one less than mirroring. Since the number of redundant disks is proportional to the $log()$ of the total number of disks in the system, storage efficiency increases as the number of data disks increases. The advantage of RAID-2 is that it can perform detection of failures even if the disks are not fail-stop in nature. If a single component fails, several of the parity components will have inconsistent values, and the failed component is the one held in common by each incorrect subset. The lost information is recovered by reading the other components in a subset, including the parity component, and setting the missing bit to 0 or 1 to get the proper parity value for that subset. Thus multiple redundant disks are required to identify the failed disk, but only one is needed to recover the lost information.

4.2 Logical Layers

Integrity assurance techniques can operate at various system levels, depending on the nature of the system and the requirements. Designing the integrity assurance technique at each of these system levels have distinct security, performance, and reliability implications. In this section we classify integrity assurance techniques into five different levels: hardware, device driver, network, file system, and user levels.

4.2.1 Hardware Level

The lowest physical level where an integrity assurance mechanism can operate is the hardware level. Mechanisms operating at the hardware level have two key advantages. First, they usually perform better than software level mechanisms, because simple integrity checkers implemented using custom hardware run faster. Second, hardware-level integrity checkers do not consume CPU cycles, thereby reducing the CPU load on the main host processor. There are two disadvantages of operating at the hardware level. First, the amount of information available to ensure integrity is usually more limited at the hardware level than at the upper levels such as the device driver or the core OS levels. Therefore, semantic data integrity cannot be ensured at the hardware level. For example, an on-disk error correcting mechanism can only ensure integrity at a block level and cannot guarantee file system consistency, as it generally has no information about which blocks are data blocks and meta-data blocks. An exception to this is Semantically-Smart Disk Systems (SDS) [34] which decipher file-system–level information at the firmware level. Second, integrity checking at the hardware level can capture only a small subset of the integrity violations, as the checked data should pass through several upper levels before it finally reaches the application, and hence there is enough room for subsequent data corruptions at those levels. Therefore, hardware-level error checks are generally rudimentary in nature, and they have to be augmented with suitable software level mechanisms so as to ensure significant integrity assurance. In this section we discuss three existing systems that employ hardware level integrity assurance mechanisms.

On-Disk Integrity Checks. Data read from a disk is susceptible to numerous bursts of errors caused by media defects, thermal asperity and error propagation in electronics. Thermal asperity is a read signal spike caused by sensor temperature rise due to contact with disk asperity or contaminant particles. Error bursts can be many bytes in length. The basis of all error detection and correction in storage disks is the inclusion of redundant information and special hardware or software to use it. Each sector of data on the hard disk contains 512 bytes or 4,096 bits of user data. In addition to these bits, an additional number of bits are added to each sector for ECC use (sometimes also called *error correcting circuits* when implemented in hardware). These bits do not contain data, but contain information about the data that can be used to correct many problems encountered trying to access the real data bits. There are several different types of error correcting codes that have been invented over the years, but the type commonly used on magnetic disks is the Reed-Solomon algorithm, named for researchers Irving Reed and Gus Solomon. Reed-Solomon codes are widely used for error detection and correction in various computing and communications media, including optical storage, high-speed modems, and data transmission channels. They have been chosen because they are faster to decode than most other similar codes, can detect (and correct) large numbers of missing bits of data, and require the least number of extra ECC bits for a given number of data bits. On-disk integrity checks suffer from the general problem of hardware level integrity checkers: they do not have much information to perform semantic integrity checks, and they capture only a subset of integrity violations.

Semantically-Smart Disk Systems. Semantically Smart Disk Systems attempt to provide file-system–like functionality without modifying the file system. Knowledge of a specific file system is embedded into the storage device, and the device provides additional functionality that would traditionally be implemented in the file system. Such systems are relatively easy to deploy, because they do not require modifications to existing file system code. As these systems decipher information from the file system running on top, they can perform file system integrity checks at the hardware level, thereby combining the performance advantages of hardware level integrity checkers, with as much information needed for performing semantic integrity assurance.

Hardware RAID. The hardware RAID parity discussed in Sections 3 and 4.1 are implemented in the RAID controller hardware.

4.2.2 Device Driver Level

In this section we discuss two systems that employ integrity assurance techniques at the device driver level.

NASD. Network Attached Secure Disks (NASDs) [10] is a storage architecture for enabling cost effective throughput scaling. The NASD interface is an object-store interface, based loosely on the inode interface for Unix file systems. Since network communication is required for storage requests such as read and write, NASDs perform integrity checks for each request sent through the network. NASDs' security is based on cryptographic capabilities. Clients obtain capabilities from a file manager using a secure and private protocol external to NASD. A capability consists of a public portion and a private key. The private key portion is a cryptographic key generated by the file manager using a keyed message digest (MAC). The network attached drive can calculate the private key and compare it with the client-supplied message digest. If there is a mismatch, NASD will reject the request, and the client must return to the file manager to retry. NASDs do not perform integrity checks on data, as software implementations of cryptographic algorithms operating at disk rates are not available with the computational resources expected on a disk. Miller's scheme for securing network-attached disks uses encryption to prevent undetectable forging of data [21].

Software RAID. Software RAID is a device driver level implementation of the different RAID levels discussed in Sections 3 and 4.1. Here the integrity assurance and redundancy techniques are performed by the software RAID device driver, instead of the RAID controller hardware. The advantage with software RAID is that no hardware infrastructure is required for setting up the RAID levels between any collections of disks or even partitions. Generally, software RAID does not perform as well as hardware RAID. Secondly, atomicity guarantees for data update and parity update are generally weaker in software RAID than in hardware RAID.

4.2.3 File System Level

The file system is one of the most commonly used level for implementing data integrity assurance mechanisms. This is because the file system level is the highest level in the kernel that deals with data management, and has the bulk of information about the organization of data on the disk, so as to perform semantic integrity checks on the data. Moreover, since file systems run inside the kernel, the extent of security that they provide is generally higher than user-level integrity checkers.

On-Disk File Systems. Almost all on-disk file systems perform offline consistency checking using user-level programs like `fsck`. Most of these consistency checking programs run at startup after an unclean shutdown of the operating system, or they are explicitly initiated by the administrator. For this reason, they cannot capture dynamic transient bit errors in the hardware that could compromise file system consistency at run time. Journaling file systems like Ext3, IBM's JFS, and ReiserFS use transactional semantics for avoid file system inconsistencies. Solaris's ZFS [37] avoids data corruption by keeping the data on the disk self-consistent at all times. It manages data using transaction groups that employ copy-on-write technology to write data to a new block on disk before changing the pointers to the data and committing the write. Because the file system is always consistent, time-consuming recovery procedures such as *fsck* are not required if the system is shutdown in an unclean manner. ZFS is designed to provide end-to-end 64-bit checksumming for all data, helping to reduce the risk of data corruption and loss. ZFS constantly checks data to ensure that it is correct, and if it detects an error in a mirrored pool, the technology can automatically repair the corrupt data.

The Protected File System (PFS) [35] is an architecture for unifying meta-data protection of journaling file systems with the data integrity protection of collision resistant cryptographic hashes. PFS computes hashes from file system blocks and uses these hashes to later verify the correctness of their contents. The hashes are computed by an asynchronous thread called *hashd* and are stored in the file system journal log for easy reading. Using write ordering based on journaling semantics for the data and the hashes, PFS ensures the integrity of every block of data read from the disk.

Stackable File Systems. Our implementation of checksummed NCryptfs [33] and I^3FS [16] perform integrity checking at the Virtual File System (VFS) level. Stackable file systems are a way to add new functionality to existing file systems. Stackable file systems operate transparently between the VFS and lower file systems, and require no modifications of lower-level file systems.

Distributed File Systems. Most of the distributed file systems perform integrity checks on their data, because the data could get transmitted back and forth through untrusted networks in a distributed environment. Distributed file systems that exist today adopt a wide range of mechanisms to ensure integrity. The Google File System [9] is a scalable distributed file system that stores data in 64MB chunks. Each *chunkserver* uses checksumming to detect corruption of the stored data. Every chunk is broken up into 64KB blocks and a 32-bit checksum value is computed for each of them. For reads, the chunkserver verifies the checksum of the data blocks before returning any data to the requester. Therefore, chunkservers do not propagate corruptions to other machines. The checksum read and update procedures are highly optimized for better performance in the Google File System.

SFSRO [7] is a read-only distributed file system that allows a large number of clients to access public, read-only data in a secure manner. In a secure area, a publisher creates a digitally signed database out of a file system's contents, and then replicates the data on untrusted content-distribution servers, allowing for high availability. SFSRO avoids performing any cryptographic operations on the servers and keeps the overhead of cryptography low on the clients. Blocks and inodes are named by *handles*, which are collision-resistant cryptographic hashes of their contents. Using the handle of the root inode of a file system, clients can verify the contents of any block by recursively checking hashes. Storing the hashes in naming handles is an efficient idea adopted by SFSRO, which not just improves performance, but also simplifies integrity checking operations.

4.2.4 Application Level

There are several application level utilities that run at the user level, performing file system integrity checks. We discuss four commonly used utilities in this section.

Tripwire. Tripwire [17, 18] is a popular integrity checking tool designed for Unix, to aid system administrators to monitor their file systems for unauthorized modifications. Tripwire reads the security policy for files in the file system and then performs scheduled integrity checks based on checksum comparison. The main goal of Tripwire is to detect and prevent malicious replacement of key files in the system by Trojans or other malicious programs.

Samhain. Samhain [31] is a multi-platform, open source solution for centralized file integrity checking and host-based intrusion detection on POSIX systems (e.g., Unix, Linux, and Windows). It was designed to monitor multiple hosts with potentially different operating systems from a central location, although it can also be used as a standalone application on a single host. Samhain supports multiple logging facilities, each of which can be configured individually. Samhain offers PGP-signed databases and configuration files and a stealth mode to protect against attempts to subvert the integrity of the Samhain client.

Radmind. Radmind [5] is a suite of Unix command-line tools and a server designed to administer the file systems of multiple Unix machines remotely. At its core, Radmind operates as Tripwire: it detects changes to any managed file system object (e.g., files, directories, links, etc.), and once a change is detected, Radmind can optionally reverse the change.

Osiris. Osiris [23] is a host integrity monitoring system that periodically monitors one or more hosts for change. It maintains detailed logs of changes to the file system, user, and group lists, resident kernel modules, etc. Osiris can be configured to email these logs to the administrator. Hosts are periodically scanned and, if desired, the records can be maintained for forensic purposes. Osiris uses OpenSSL for encryption and authentication in all components.

4.3 Online vs. Offline Integrity Checks

Some systems like I^{3}FS [16], SFSRO [7], PFS [35] etc., adopt online integrity checking which means that they ensure integrity in the critical path of a read or write operation. Such systems are much more effective than offline integrity checkers like `fsck`, Tripwire, Samhain, etc. This is because online integrity checkers can detect integrity violations before the violation could cause damage to the system. For example, I^{3}FS can detect malicious replacement of binary executables at the time they are read (for executing) and hence can prevent execution immediately. With offline methods that perform integrity checking in scheduled intervals of time, there is a window of vulnerability during which they cannot prevent the damage caused by an intrusion. Therefore, from the security viewpoint, online integrity checkers are better than offline ones. However, online methods mostly come with performance costs. Performing operations like checksum comparison in the critical section of a file system read could slow down the system noticeably. Offline integrity checkers generally run in an asynchronous manner and hence do not pose significant performance problems. Depending on the importance of the data to be protected, a suitable integrity assurance mechanism can be chosen by the administrator. For vital data, online integrity checkers are better suited.

5. USES OF INTEGRITY CHECKING

5.1 Security

Data-integrity assurance techniques go a long way in making a computer system secure. A large class of attacks on systems today are made possible by malicious modification of key files stored on the file systems. If authorized modifications to files are detected in time, damage caused by the intrusion can be reduced or even prevented. In this section we discuss three different applications of integrity assurance in the viewpoint of systems security.

Intrusion Detection. In the last few years, security advisory boards have seen an increase in the number of intrusion attacks on computer systems. A large class of these intrusion attacks are performed by replacing key binary executables like the ones in the `/bin` directory with custom back-doors or Trojans. Integrity checking utilities like Tripwire [18], Checksummed NCryptfs [33], and I^{3}FS [16] detect unauthorized modification or replacement of files with the help of checksums. Online notification of integrity violations and immediate prevention of access to the corrupted file helps in reducing the damages caused by the intrusion. Self-Securing Storage [36] prevents intruders from undetectably tampering with or permanently deleting stored data, by internally auditing and logging operations within a window. System administrators can use this information to diagnose intrusions.

Non-Repudiation and Self-Certification. Distributed storage systems like SFSRO [7] and NASD [10] have public- or private-key-based signatures for integrity assurance. Each request sent between nodes of the network is appended with a public-key-based signature generated from the request contents. This method provides authentication and assurance about the integrity of the request received at the receiver's end, and it also helps in ensuring non-repudiation and self-certification because only the right sender can generate the signature.

Trusting Untrusted Networks. Distributed file systems exchange control and data information over untrusted networks. Integrity assurance mechanisms like tamper-resistant HMAC checksums and public key signatures verify that the information sent through untrusted networks is not modified or corrupted.

5.2 Performance

The design of a certain class of integrity assurance mechanisms takes advantage of already existing redundant information to improve system performance. We discuss two examples where redundant information helps improve performance.

Duplicate Elimination. Low-Bandwidth Network File System (LBFS) [22] and Venti [27] use checksums for eliminating duplicates in their data objects. Since duplicate data objects share the same checksum value, a reasonably collision-resistant checksumming scheme could help identify duplicates by comparing their checksums. This method of duplicate identification is efficient because the length of data that needs to be compared is usually 128-bit checksums, compared to data blocks which could be of the order of kilobytes. Duplicate elimination helps in reducing storage space and enables better cache utilization, and hence improves performance. Checksums are used for duplication elimination in the Rsync protocol [38].

Indexing. Checksums are a good way to index data. Object disks can use checksums for indexing their objects [20]. SFSRO uses collision resistant checksums for naming blocks and inodes. Though highly collision-resistant checksums can be slightly larger than traditionally used integers, they help achieve dual functionality with a small incremental cost. Using checksums for naming handles offers an easy method for retrieving the checksums associated with blocks and thus it improves integrity checking performance.

5.3 Detecting Failures

Integrity checks on raw data can be used to identify disk failures. Data corruption is an important symptom of a disk failure and disks that are not fail-stop in nature could still continue to operate after silently corrupting the data they store. Fail-stop disks stop functioning upon a failure, thereby explicitly notifying the controller that they failed. Non-recoverable disk errors have to be identified in time so as to protect at least portions of the data; integrity checking techniques can be used to achieve this goal. Some modern disks already detect failures using on-disk integrity checking mechanisms. In systems like RAID-5 that assume the fail-stop nature of individual disks to detect failures, checksumming can be added to enable them to perform automatic detection and recovery of disk failures, even in the case of non-fail-stop disks.

6. IMPLEMENTATION CHOICES

6.1 Granularity of Integrity Checking

There are several different granularities at which integrity checks can be performed. For example, if checksumming is used, the size of data to be checksummed can be either a byte, a block, or even a whole file stored on disk. Choosing the right granularity of data to be checksummed for a particular system is important for achieving good performance and security. Operating at a finer granularity for integrity checks generally results in too many computations of redundant information, especially when the access pattern is large bulk reads. On the other hand, having a large enough granularity of data could result in more I/O for smaller reads, because an entire chunk of data needs to be read for performing an integrity check. Therefore, the optimal choice of granularity depends on the nature of the system and also on the general access pattern. In this section we discuss four different choices of granularities at which integrity checks can be performed, along with their trade-offs.

Block Level. Usually, hardware level and device driver level integrity assurance mechanisms operate at the granularity of disk blocks. This is mainly because higher level abstractions like page, file, etc., are not visible at these lower levels. Most of the configurations of RAID and NASD discussed in Section 4.2 perform integrity checks at the block level. For physical redundancy checking like checksumming, checksums are computed for each block on disk and then stored. Upon reading a disk block, its checksum will be recomputed and compared with the stored value. In block-level checksumming, for large reads that span a large number of disk blocks, the number of checksum computations and comparisons required can be large. To mitigate this problem, most of the block level integrity checkers are implemented in hardware. Block level integrity checkers cannot perform file system level semantic integrity checking and hence they are limited in their scope. The Protected File System (PFS) discussed in Section 4.2 employs checksumming at a block granularity, but at the file system level.

Network Request Level. In distributed file systems, the requests sent between different nodes of the network need to be authentic so as to ensure security. Several distributed storage systems such as NASD adopt public key signatures or HMAC checksums to authenticate requests sent between nodes. The sender appends a signature with every request and it is verified at the receiver end. NASDs adopt integrity checking for requests but not for data, because network request transfers are more important than data transfers, and forging network requests could break confidentiality.

Page Level. Data integrity assurance techniques at the file system level generally operate at a page granularity, as the smallest unit of data read from the disk by a file system is usually a page. Since file system page sizes are usually larger than disk block sizes, operating at a page level often results in a better balance between the number of checksums computations required in proportion to the size of data read. The checksummed NCryptfs discussed in Section 4.2 performs checksumming at a page level. Every time a new page is read from the disk, NCryptfs recomputes the checksum and compares it with the stored value to verify the integrity of the page data. I^{3}FS also has a mode for performing per-page checksumming [16].

File Level. Application level integrity checkers such as Tripwire perform checksumming at the file level. The advantage with file-level checksumming is that the storage space required for storing the checksums is reduced compared to page level checksumming, because each file has only one checksum value associated with it. In page level checksumming, large files could have several checksum values associated with them, requiring more storage space and efficient retrieval methods. For that reason, I^{3}FS also includes a mode to perform whole-file checksumming.

6.2 Storing Redundant Data

Integrity checking requires the management of redundant information (physical or logical) persistently. Because many integrity assurance methods are online in nature and operate during the critical section of reads and writes, efficiency is a key property that the storage and retrieval mechanisms should have. There are a several different techniques that existing mechanisms adopt. In this section we discuss a few of them.

SFSRO stores collision-resistant cryptographic checksums of file system objects in the form of naming handles [7]. Venti [27], a network storage system, uses unique hashes of block contents to identify a block. In SFSRO, blocks and inodes are named using the checksums of their contents, so that given the naming handle of the root inode of a file system, a client can verify the contents of any block by recursively checking hashes. This is an efficient way of optimizing performance for storing and retrieving checksums. On-disk parity information for error-correcting codes is stored at the block level, much closer to the relevant blocks, to avoid additional disk rotation or seeks for reading the parity information.

Checksummed NCryptfs [33] uses parallel files to store the page-level checksums for each file. Each file in the file system has an associated hidden checksum file which stores the checksums of each of the file pages. Whenever a file is read, the associated checksum is read from the hidden file and compared for integrity violations. The advantage of this method is that it can be easily implemented in a transparent manner across different underlying file systems. Checksummed NCryptfs can be used with any underlying on-disk or network file systems. The problem with using parallel files is that each time a file is opened, it gets translated to opening two files (the checksum file also). This affects performance to some extent.

I^{3}FS uses in-kernel Berkeley databases (KBDB) [15] to store both page level and whole file checksums. Since KBDB supports efficient storage and retrieval of $\langle$`key,value`$\rangle$ pairs in the form of B-trees, hash tables, etc., it enables easy storage and retrieval of checksums, keyed by inode numbers and page numbers. PFS [35] stores block level checksums for both data and meta-data blocks as part of the journal, indexed by a logical record number. It also has an in-memory hash table for easy retrieval of checksums from the journal.

7. LOGICAL REDUNDANCY

Techniques that involve logical redundancy exploit the semantics of data to verify integrity. Logical redundancy mechanisms can be used to perform integrity checks by exploiting the inherent semantic redundancy that already exists in the data. An example of a system that employs logical redundancy is the Pilot operating system [29] whose file system meta-data is *advisory* in nature in that the entire meta-data of a file can be regenerated by scanning through the file data. It stores meta-data for faster access to the file, but it can potentially use this information to perform integrity checks on the file when needed by just comparing the regenerated value and the stored value of the meta-data. Similarly, in some size-changing stackable file systems [43] the index files that store mapping information for sizes are totally reconstructible from the file data. This can also be used for performing online integrity checks to detect inconsistencies between file sizes.

Logical redundancy is a common method used by databases to maintain the semantic integrity of information they store [11]. We have identified that the general technique of logical redundancy can be exploited by storage systems at various levels, to implement seamless integrity checking. The main advantage of this method is that there is minimal additional overhead imposed by the integrity checking mechanism, because the storage and perhaps retrieval of redundant information is already performed by the system for other purposes. There are several areas in storage where logical redundancy can be exploited to ensure the integrity of data. We discuss three in this section.

7.1 Pilot File System

The Pilot file system is part of the Pilot Operating System [29]. The Pilot file system's uniqueness is its robustness. This is achieved primarily through the use of reconstructible maps. Many systems make use of a file *scavenger*, a startup consistency checker for a file system, like `fsck`. In Pilot, the scavenger is given first-class status, in the sense that the file structures were all designed from the beginning with the scavenger in mind. Each file page is self-identifying by virtue of its label, written as a separate physical record adjacent to the one holding the actual contents of the page. Conceptually, one can think of a file page access as proceeding to scan all known volumes, and checking the label of each page encountered until the desired one is found. In practice, this scan is performed only once by the scavenger, which leaves behind maps on each volume describing what it found there. Pilot then uses the maps and incrementally updates them as file pages are created and deleted. The logical redundancy of the maps does not imply lack of importance, because the system would be not be efficient without them. Since they contain only redundant information, they can be completely reconstructed should they be lost. In particular, this means that damage to any page on the disk can compromise only data on that page. The primary map structure is the volume file map, a B-tree keyed on file-UID and page-number which returns the device address of the page. All file storage devices check the label of the page and abort the I/O operation in case of a mismatch; mismatch does not occur in normal operation and generally indicates the need to scavenge the volume. The volume file map uses extensive compression of UIDs and run-encoding of page numbers to maximize the out-degree of the internal nodes of the B-tree and thus minimize its depth.

The volume allocation map is also an important part of the Pilot file system. It is a table that describes the allocation status of each page on the disk. Each free page is a self-identifying member of a hypothetical file of free pages, allowing reconstruction of the volume allocation map.

The Pilot file system is a very good example for how logical redundancy can be exploited for performing integrity checks. However, the Pilot operating system is just a research OS and is not in use today. In general, with logical redundancy, although we cannot ensure block data integrity, we can ensure meta-data integrity without storing any additional information. For key files in the file system whose integrity has to be monitored, its meta-data can be reconstructed each time the file is read, and can be compared with the stored meta-data. This can detect inconsistencies between the data and meta-data. This method of online integrity checking is useful in cases where important files are frequently updated by concurrent processes, giving room for inconsistencies during a system fault.

7.2 File System Bitmaps

Current Unix file systems such as Ext2 use several bitmaps for managing meta-data. For example, in Ext2, inode table entries are identified through inode bitmaps. Each bit represents the current state of an entry in the inode table within that group, where 1 means "used" and 0 means "free/available." In addition to this bitmap, the file system also stores the current state of the inode which exists in the inode table. Whenever an inode is deleted, the file system marks the entry in the inode table as "deleted." This is redundant information maintained by Ext2 which is presently used by file system startup integrity checkers like `fsck` to reconstruct the bitmaps when the file system is corrupted due to a system crash.

This logical redundancy can be used for performing online integrity checks. In this scenario, integrity checks help prevent two kinds of potential problems: unexpected loss of data, and wasted inodes and disk blocks. A transient hardware bit error while reading the inode bitmap could make an existing file inaccessible. If a bitmap value for a used inode is read as free, then the file system could overwrite the existing inode for a newly created file, thereby making the file pointed to by the older inode unreachable. This also makes the blocks occupied by the older file inaccessible, resulting in wasted disk space. Similarly, if while reading the inode bitmap, a bit error occurs, this could result in a free inode being read as used, and the corresponding inode will be wasted. By performing online integrity checks based on logical redundancy, bit errors while reading bitmap blocks can easily be identified, and their effects can be prevented.

7.3 On-Disk Data Structures

Several indexing schemes and data storage patterns are being employed for efficient retrieval of data from secondary storage devices. Hash files and B-trees are the most common examples. B-trees are balanced trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive time-consuming operations, a B-tree tries to minimize the number of disk accesses. For example, a B-tree with a height of 2 and a branching factor of 1,001 can store over one billion keys but requires at most two disk accesses to search for any node. Each node other than the leaf nodes in a B-tree has pointers to several children. One of the common integrity problems with B-trees are pointer corruptions. A pointer corruption in a single node of a B-tree can cause serious problems while reading the data stored in it, as the entire branching path will be changed due to the wrong pointer value. Several methods can be used to perform integrity checks in B-trees. The most common among them is to store the checksums of the child nodes along with the pointers in each node. With these checksums, each time a pointer is followed, the checksum of the child node can be computed and compared with the stored value. This method can effectively detect integrity violations in B-trees, but is not quite efficient. A modification to a single leaf node data requires the recomputation and storing the checksums of the entire ancestor path up to the root. Moreover, computing and comparing checksums for each pointer access could seriously affect performance.

Logical redundancy techniques can be employed for performing efficient integrity checks in B-trees. If each child node has a back pointer to its parent, every time we follow a pointer, we can check if the new child node visited points back to its parent. This way pointer corruptions can easily be detected without the hassle of generating and comparing checksums. Although this method cannot ensure the integrity of data in the child nodes, it can effectively detect pointer corruptions with minimal space and time overheads.

8. CONCLUSIONS AND FUTURE WORK

This paper presents a survey of different integrity assurance mechanisms that are in use today. We have analyzed integrity assurance techniques from three different dimensions in our taxonomy: the scope of assurance, logical layer, and checking mode. We have also discussed several interesting applications of integrity assurance. We analyzed how existing systems that employ integrity checks can use redundant data to improve their performance and add new functionality. We presented real examples for some of the systems. We discussed several implementation choices for integrity checking granularity and managing redundant information.

We formalized a new class of efficient integrity assurance mechanisms called *logical redundancy* and discussed three examples where it can be used. In our taxonomy we describe integrity assurance techniques in four different viewpoints: the redundancy mechanisms used, their scope, their level of operation, and the frequency at which checks are performed. We discussed the operation of several existing systems in each of those viewpoints.

Our experience describing the taxonomy of integrity assurance techniques has helped us focus our thinking on exploring more logical redundancy techniques for integrity checking at low cost, in a highly efficient manner. We hope to explore further systems that maintain redundant information as part of their normal operation, but do not quite use them for performing integrity checks. These systems can be made more secure and efficient, by making use of the information that they maintain.

9. REFERENCES

[1] W. Barlett and L. Spainbower. Commercial fault tolerance: A tale of two systems. In *Proceedings of the IEEE Transactions on Dependable and Secure Computing*, pages 87–96, January 2004.

[2] E. W. Biersack. Performance evaluation of Forward Error Correction in ATM networks. In *SIGCOMM '92: Conference proceedings on Communications architectures & protocols*, pages 248–257, New York, NY, USA, 1992. ACM Press.

[3] M. Blaze. A Cryptographic File System for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, VA, 1993. ACM.

[4] CERT Coordination Center. CERT/CC Overview incident and Vulnerability Trends Technical Report. `www.cert.org/present/cert-overview-trends`.

[5] W. Craig and P. M. McNeal. Radmind: The Integration of Filesystem Integrity Checking with File System Management. In *Proceedings of the 17th USENIX Large Installation System Administration Conference (LISA 2003)*, October 2003.

[6] P. A. DesAutels. SHA1: Secure Hash Algorithm. `www.w3.org/PICS/DSig/SHA1_1_0.html`, 1997.

[7] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and Secure Distributed Read-Only File System. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181–196, San Diego, CA, October 2000.

[8] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *Usenix '05: Proceedings of the Usenix Annual Technical Conference*, pages 89–104, Anaheim, CA., USA, 2005. Usenix.

[9] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, NY, October 2003.

[10] G. A. Gibson, D. F. Nagle, W. Courtright II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. NASD Scalable Storage Systems. In *Proceedings of the 1999 USENIX Extreme Linux Workshop*, Monterey, CA, June 1999.

[11] L. Golubchik and R. Muntz. Fault tolerance issues in data declustering for parallel database systems. *Bulletin of the Technical Committee on Data Engineering*, 14–28, September 1994.

[12] R. Hagmann. Reimplementing the cedar file system using logging and group commit. *ACM SIGOPS Operating Systems Review*, 21(5):155–162, 1987.

[13] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, Vol XXVI, April 1950.

[14] W. Hsu and S. Ong. Fossilization: A Process for Establishing Truly Trustworthy Records. *IBM Research Report*, 2004.

[15] A. Kashyap, J. Dave, M. Zubair, C. P. Wright, and E. Zadok. Using the Berkeley Database in the Linux Kernel. `www.fsl.cs.sunysb.edu/project-kbdb.html`, 2004.

[16] A. Kashyap, S. Patil, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004. USENIX Association.

[17] G. Kim and E. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. In *Proceedings of the Usenix System Administration, Networking and Security (SANS III)*, 1994.

[18] G. Kim and E. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer Commuications and Society (CCS)*, November 1994.

[19] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Technical Report RFC 2104, Internet Activities Board, February 1997.

[20] M. Mesnier, G. R. Ganger, and E. Riedel. Object based storage. *IEEE Communications Magazine*, 41, August 2003. ieeexplore.ieee.org.

[21] E. Miller, W. Freeman, D. Long, and B. Reed. Strong Security for Network-Attached Storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 1–13, Monterey, CA, January 2002.

[22] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-Bandwidth Network File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[23] Osiris. Osiris: Host Integrity Management Tool, 2004. `www.osiris.com`.

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

[25] E. W. Patterson and E. J. Weldon. Error-correcting codes. *MIT Press Cambridge*, 2nd ed., 1972.

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

[27] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies*, pages 89–101, Monterey, CA, January 2002.

[28] R. J. McEliece and D. V. Sarwate. On sharing secrets and Reed-Solomon codes. *Commun. ACM*, 24(9):583–584, 1981.

[29] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An operating system for a personal computer. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 106–107, 1979.

[30] R. L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. In *Internet Activities Board*. Internet Activities Board, April 1992.

[31] Samhain Labs. Samhain: File System Integrity Checker, 2004. `http://samhain.sourceforge.net`.

[32] M. Seltzer and O. Yigit. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*, pages

173–184, Dallas, TX, January 1991. USENIX Association.

[33] G. Sivathanu, C. P. Wright, and E. Zadok. Enhancing File System Integrity Through Checksums. Technical Report FSL-04-04, Computer Science Department, Stony Brook University, May 2004. `www.fsl.cs.sunysb.edu/docs/nc-checksum-tr/nc-checksum.pdf`.

[34] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, , and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[35] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying file system protection. In *Proceedings of the Annual USENIX Technical Conference*, pages 79–90, Boston, MA, June 2001. USENIX Association.

[36] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 165–180, San Diego, CA, October 2000.

[37] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. `www.sun.com/software/solaris/ds/zfs.jsp`.

[38] A Trigdell and P Mackerras. The rsync algorithm. Technical report, Australian National University, 1998.

[39] Tripwire Inc. Tripwire Software. `www.tripwire.com`.

[40] G. Weinberg. The solaris dynamic file system. Technical report, urlmembers.visi.net/thedave/sun/DynFS.pdf, 2004.

[41] C. P. Wright, J. Dave, and E. Zadok. Cryptographic File Systems Performance: What You Don't Know Can Hurt You. Technical Report FSL-03-02, Computer Science Department, Stony Brook University, August 2003. `www.fsl.cs.sunysb.edu/docs/nc-perf/perf.pdf`.

[42] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.

[43] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 289–304, Boston, MA, June 2001. USENIX Association.