Block Storage Listener for Detecting File-Level Intrusions

Miriam Allalouf, Muli Ben-Yehuda, Julian Satran, Itai Segall
{miriama, muli, Julian_Satran, itais}@il.ibm.com
IBM Research – Haifa, Israel

***Abstract*—An intrusion detection system (IDS) is usually located and operated at the host, where it captures local suspicious events, or at an appliance that listens to the network activity. Providing an online IDS to the storage controller is essential for dealing with compromised hosts or coordinated attacks by multiple hosts. SAN block storage controllers are connected to the world via block-level protocols, such as iSCSI and Fibre Channel. Usually, block-level storage systems do not maintain information specific to the file-system using them. The range of threats that can be handled at the block level is limited. A file system view at the controller, together with the knowledge of which arriving block belongs to which file or inode, will enable the detection of file-level threats.**

In this paper, we present IDStor, an IDS for block-based storage. IDStor acts as a listener to storage traffic, out of the controller's I/O path, and is therefore attractive for integration into existing SAN-based storage solutions. IDStor maintains a block-to-file mapping that is updated online. Using this mapping, IDStor infers the semantics of file-level commands from the intercepted block-level operations, thereby detecting file-level intrusions by merely observing the block read and write commands passing between the hosts and the controller.

I. Introduction

An Intrusion detection system (IDS) is an appliance or application that monitors network and/or system activities for malicious activities or policy violations. There are two main types of IDS systems: network-based and host-based IDS. In a network-based intrusion-detection system, the sensors are located at points in the network to be monitored. The sensor captures all network traffic and analyzes the content of individual packets in order to detect malicious traffic. In a host-based system, the sensor usually consists of a software agent that monitors all activity of the host on which it is installed, including file system, logs and the kernel. In a storage-based IDS, a sensor captures all the traffic (I/O requests) that arrives at the storage controller, and analyzes possible storage system violations or threats.

An online IDS at the SAN controller could handle several types of threats, as follows: (1) Threats that are typically handled at the host level by Tripwire-like [1] tools. Such tools can identify when data or metadata that belongs to files that administrators expect to remain unchanged is modified. Examples of such files include system executables and scripts, configuration files, system header files and libraries. These tools can also identify suspicious patterns of access (usually patterns of updates) to certain files or to the file system. Specific examples to this group of threats are overwriting data in system log files, or reversing file modification times. (2) Threats that can cause storage denial-of-service, when an attacker disables specific services or entire systems by allocating all or most of the free space or by allocating many inodes or other metadata structures. (3) Leakage of sensitive data when written to non-secure machines or disks, which can be prevented by Data Leakage Prevention (DLP) tools. An additional possible feature of a storage-based IDS is the ability to trigger a (typically incremental) antivirus scan upon access. Such storage-based IDS that can identify and alert upon the occurrence of these threats usually perform side-by-side with the host-based IDS. However, only a storage-based intrusion detection mechanism is effective in case hosts are compromised or in case multiple hosts share an attack that can be detected only by the central storage.

Despite the benefits of detecting intrusions at the storage level, no storage-based IDSs exist for the SAN block controller. The very few storage systems that do maintain an online IDS are accessed via file-level protocols, such as CIFS or NFS. The deployment of a block-level storage-based IDS is more complicated. The SAN block storage controller interacts with hosts via block-level protocols, such as iSCSI and Fibre Channel. Usually, block-level storage systems do not maintain information specific to the file-system using them. Thus, the range of threats that can be handled directly from block-level traffic is very limited. A file system view at the controller, together with the knowledge of which arriving block belongs to which file or inode, will enable the detection of file-level threats.

Moreover, obtaining a block-to-file view at a listener appliance is a challenging research problem. Several works have already suggested how to add a file system view to the controller in order to handle file-level threats [2], [3], [4]. However, the previous works are within the controller's I/O path, an approach that suffers from several limitations. First, adding software to the modules that handle the I/O path of a controller is a complicated and error-prone task, with heavy development expenses. Second, the CPU capacity at the controller is designed to handle the arriving I/O requests and may not be able to perform additional computation tasks that are required in order to obtain the file-view. Finally, it is much more appealing for a client with an existing storage solution to add security by plugging in a new external appliance, rather than replacing or patching the existing system.

To solve these shortcomings we designed and developed

978-1-4244-7153-9/10/$26.00

an IDS for block-based storage (IDStor) framework. IDStor has several significant advantages and contributions over previously suggested solutions. First, in order to detect file-level threats, we obtain a full file system view from block-level traffic. Such detailed analysis and hands-on solution were not provided in previous works. Second, we provide a full implementation of such a view at a listener appliance, rather than at the I/O path. These two contributions open the way to set an initial framework for an online intrusion detection system based on the file-system view. This paper presents preliminary work of online detection given online file inference, and highlights its feasibility.

For the purpose of obtaining a file-system view, we focus on the ext3 file system, and on the iSCSI block-level protocol. We maintain an updated block-to-file map where each data block points to the file that owns it. The map is updated in an online manner by capturing the arriving block-level traffic, i.e, iSCSI commands, and translating them back to the file-level commands performed at the host. Initially, this mapping is built by traversing the metadata information kept in the disk and it is then updated in an online manner. The maintenance of a reliable inverse mapping only by listening to block-level commands is a challenging task, since each file-level command is translated into several block-level commands, and those are interleaved with commands originating from other file-level commands. In order to reverse this translation, we maintain state machines for inodes and data blocks, until their state is resolved and inserted into the inverse mapping. One challenge we solved is how to parse the arriving commands in an order that may be different from the one in which they arrived, such that the interpretation will be correct. The details of our solution is presented in Section III.

IDStor acts as a passive listener and can be located at a listener appliance or a listener module, rather than an active part that is inserted into the controller and can interfere with the ongoing I/O requests. This combination of obtaining a block-to-file mapping together with the listener architecture is a viable path to the customer. This solution is not controller-specific and will not affect the controller's operation.

IDStor can also act as a regular network-based IDS that handles block-level threats, such as detecting overly-long strings that might cause buffer overflows. Essentially, we add an iSCSI interpreter to the list of protocols that are parsed and checked by an IDS. Section V presents only the preliminary work that was done towards the handling of file-level threats while further details are left for future work.

II. IDS FOR BLOCK-BASED STORAGE (IDSTOR) - SYSTEM DESCRIPTION

This section presents a general description of *IDStor*, a storage-based intrusion detection system that acts as a listener, and can be embedded in an appliance. We distinguish between two types of storage-based IDS architectures. In the first architecture, the IDS system is located on the I/O path of the controller, between any traffic that arrives at the controller and the storage. It may delay the arriving command and compare it with previous content before allowing the command to pass. The second architecture detects intrusions by passively listening to the stream between the host and the target. The IDS system is located at a listener appliance, where the arriving traffic can be captured.

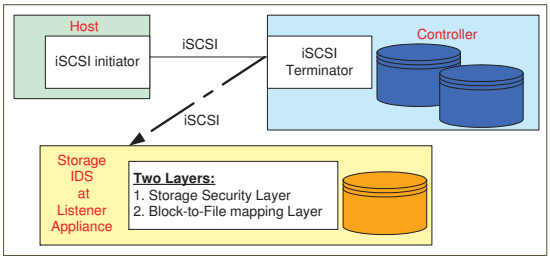

Fig. 1. Storage IDS using the listener architecture.

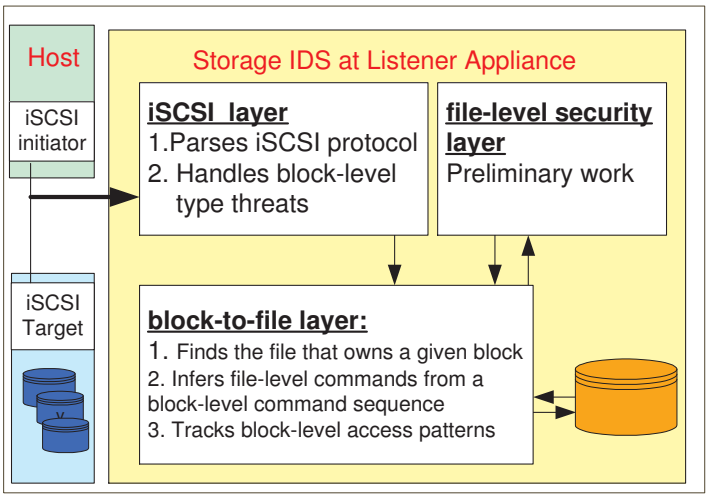

Fig. 2. Storage IDS Software Architecture.

The first architecture suffers from several limitations. Adding software to the modules that handle the I/O path of a controller is a complicated and error-prone task, with heavy development expenses. In addition, the CPU capacity at the controller is designed to handle the arriving I/O requests and may not be able to perform additional computation tasks that are required in order to obtain the file-view. Finally, it is much more appealing for a client with an existing storage solution to add security by plugging in a new external appliance as in the second architecture, rather than replacing or patching the existing system.

Thus in this work, we select the second architecture, where IDStor acts as a listener appliance, as depicted in Figure 1. In case of a write command, this architecture can not compare the arriving packet data against the current content on the disk, since the new data is written in parallel by the target. It also cannot affect the I/O path by delaying the command or the violation.

For the purpose of obtaining a file-system view, we focus on the ext3 file system, highlighting the parts that are general, and applicable to any file system, and the ones that are ext3-specific. We use the iSCSI block-level protocol, and assume that the logical unit (LUN) at the iSCSI target is a raw disk structure without logical volume management (LVM) or

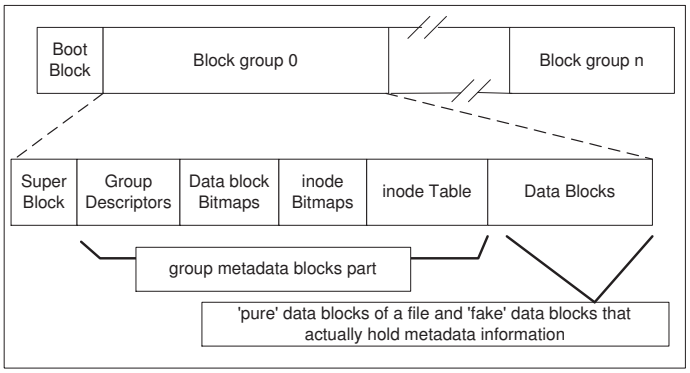

Fig. 3. Layouts of an ext2 partition and ext2 block group. ext3 adds journaling to ext2 and has the same layout.

RAID. We also require that during the initialization of IDStor, there is no other I/O traffic to the monitored LUNs, thus letting IDStor build an initial mapping of the file system.

IDStor looks for file-level violations using a file-aware view. The different components and layers of IDStor are depicted in Figure 2. The *iSCSI layer* is part of the monitoring layer that captures all the arriving traffic and parses the iSCSI packets. It first looks for block-level violations, and then the block command is transferred to the *Block-to-File Layer*. This layer infers the file-level commands and maintains the block-to-file mapping (see Section III). Each inferred file-level command is passed to the *security layer* that detects an intrusion to a file. The security layer (see Section V) matches the file-level command (delivered by the block-to-file layer) to the file-level rules in its database, and announces an intrusion in case of a violation.

III. How to Obtain File-Awareness at a Block-Level Storage?

The *block-to-file layer* in IDStor is composed of several parts:

- **Building and maintaining a block-to-file mapping**. This mapping enables us to answer questions like: "given a block number, which inode owns it?" and "what is the file name of this inode?".
- **Inferring file-level commands** from the arriving block-level commands, thus supplying access-related information. This feature enables us to answer questions such as "what was the order, the rate, or the pattern of access inside a specific file?" and "which commands were operated on this file?".

Figure 3, adapted from the book "Understanding the Linux Kernel" by Bovet and Casati, provides the ext3 file system layout [5]. The disk is partitioned into block groups. Each block group starts with metadata blocks followed by data blocks. Among other information on the structure of the files and their hierarchy, the metadata blocks hold an inode table for each inode in this block group. An inode is an internal structure representing a "physical" file entity. The inode structure in the inode table keeps 128 Bytes containing various inode attributes and the list of the blocks belonging to this inode (using the *i_block* field). An inode of a regular file points to the list of its data blocks. An inode representing a directory points to blocks that contain the inode numbers that are under this directory, along with their filenames. When using hard links, each inode can be referred to by several names, where the filenames are kept in the data blocks of the respective parent directories.

The data blocks portion of the layout consists of data blocks that keep the contents of regular files, to which we refer as *pure* data blocks, and ones that actually keep metadata information, to which we refer as *fake* data blocks. One example for the latter is the data blocks of a directory inode, as described above. Another example is indirect addressing blocks. To save memory, the inode structure maintains up to 3 levels of addressing indirection. Indirected address blocks are allocated only when a file grows and more data blocks are required. The file system allocates pure blocks and fake blocks from the data blocks pool indiscriminately, and thus a data block can be a pure data block when associated with one inode, and later (after being freed) reallocated as a fake one for a different inode (or vice versa).

The file-system metadata information is organized in a way that enables answering host-level requests such as "list the blocks that belong to a certain inode" or "list the files that belong to a certain directory". On the other hand, in order to handle file-level threats, IDStor has to be able to answer questions such as: "given a block number, which inode owns it?" and "what is the file name and parent directory of this inode?". The layout of ext3, like most file systems, was not designed to resolve this easily. In order to find an inode owning a block, using only the standard file system data structures, one must traverse the entire inode table until an inode that contains the required block is found. Moreover, when the filename is also required, and since the directory hierarchy is kept separately from the inode information, an additional mapping has to be resolved and the directory structure must also be found and traversed.

This problem is even more crucial when considering the online model, as we are doing, where we infer the file view from captured block level commands, while requiring as little memory as possible, and acting as a listener. The fact that the non-metadata block range holds also metadata information, such as file names and indirect block pointers, amplifies the challenges in this framework even further.

We will first show how to map a block to its inode, and then explain what is required in order to map an inode to a file name.

A. Block-Level Protocol Translation

At the file system level, a file is viewed using its name and is composed of logical blocks of given size (e.g., 4KB). The SAN block storage controller is not aware of the semantic meaning of the content it stores and treats it as a sequential layout of device blocks (e.g., 512B). A controller that is connected via block-level protocols, such as iSCSI and Fibre Channel, accepts a stream of read and write block-level commands. Our goal in IDStor is to build an inverse mapping, where

File Command	iSCSI Operation,Block Type
Read File	Read the data block
	Write `atime` field in the inode table
Write File	Write the data block
Overwrite	Write inode table fields
Append in last block	Write inode table fields
Append new blocks	Write superblock
	Write group desc.
	Set the bit in data bitmap
	Write inode table fields
Partial file truncate	Write superblock
	Write group desc.
	Unset the bit in data bitmap
	Write inode table fields
New File	Write superblock
	Write group desc.
	Set the bit in data bitmap
	Set the bit in inode bitmap
	Write inode table fields
	Write parent inode table fields
	Write data blocks
	Write parent data (the file name)
Delete File	Write superblock
	Write group desc.
	Unset the bit in data bitmap
	Unset the bit in inode bitmap
	Write inode table fields
	Write parent inode table fields
New Directory	Write superblock
	Write group desc.
	Set the bit in data bitmap
	Set the bit in inode bitmap
	Write inode table fields
	Write parent inode table fields
	Write data block
	Write parent data (the directory name)
Remove Directory	Write superblock
	Write group desc.
	Set the bit in data bitmap
	Set the bit in inode bitmap
	Write inode table fields
	Write parent inode table fields
	Write parent data (remove the directory name)

TABLE I
EXAMPLES OF FILE-LEVEL COMMANDS TRANSLATED TO ISCSI COMMANDS

blocks refer to the inodes that own them. In order to build and maintain this mapping, IDStor performs the following steps: captures the block-level commands, identifies the type of the iSCSI command and the type of the block specified in the command, inserts the relevant information to the state machine and finally adds the block-to-inode information to the inverse mapping.

Each host-level command is usually translated into several different block-level commands that have no explicit relationship to each other. For example, a host-level command that creates a new file is translated into block-level commands that: (1) updates the superblock fields specifying the number of allocated blocks and inodes, (2) updates the field in the group descriptor of the appropriate block group specifying the number of free inodes, (3) sets the relevant bit in the inode bitmap to true, (4) sets the relevant bits in the data bitmap to true, corresponding to the additional data blocks that were allocated for this file, (5) creates a new inode structure in the inode table, (6) updates the access time fields in the parent directory's inode structure (assuming the file system was not mounted using the *no_atime* switch) , (7) adds the file name to the relevant data block of the parent inode, and finally, (8) writes the data blocks of this inode. Table I provides the list of the file manipulation commands and the corresponding list of the block-level manipulations. A consistent file system requires that all the block-level commands that compose a certain file-level command will finally arrive to the disk, but often there are no requirements on the order in which they do so. We can obtain a feasible inverse mapping by parsing the block level commands. The challenge of this task arises from the fact that the file system information can be delayed in the host's cache and flushed to the storage in any order and usually after some delay. However, eventually all the information should be flushed to the disk and captured at the block-level.

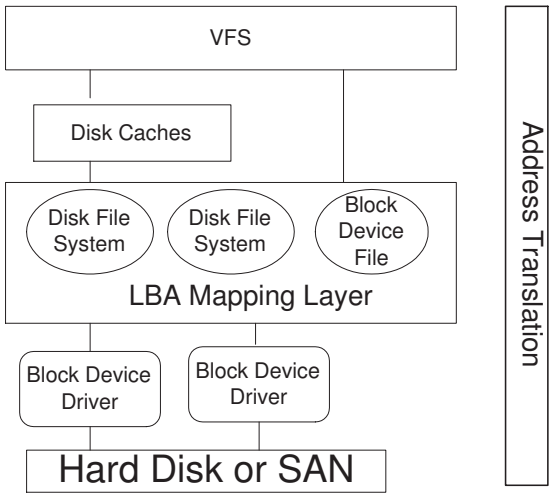

Fig. 4. Block Device Operation - stages.

An iSCSI command can be a "read" or a "write" command that accesses some arbitrary block on the disk. This block can be either a metadata block or a data block. Given a block number, IDStor translates it from the block device address space into the file-level address space by taking into account the relation between the filesystem block size and the disk sector size, and the on-disk offset of the partition (Figure 4 describes the different stages and components that a block device operation passes). Then, in the case of a write to a metadata block, it determines its type, i.e, to which block in the block group it belongs (see Figure 3). The sizes of block groups and metadata structures are set at file system creation and are held in the superblock. The block type is identified by performing simple division and modulo operations on the retrieved block group size and metadata size. The block numbers in commands $1-6$ in the example above were identified as metadata blocks while command 7 refers to a fake data block. Command 8 describes a write of a pure data block.

The identification of metadata blocks (as in commands $1-6$) is not enough for maintaining an inverse block to file mapping. For instance, we identify in command 5 that it accesses the

inode table block, but we do not know which inode in the inode table was accessed. Each inode table block contains many inodes (as many as the block size divided by 128B, so an inode table block with a logical block size of 4KB can accommodate 32 inodes). As mentioned above, since we are working as a listener to the network traffic, we do not have access to the older copy of this block when encountering a new write to it. One option of attacking this problem would be to keep a copy of all such blocks, thus being able to compare the new and old versions. There is a clear tradeoff here between memory consumption and performance. We opted for another option – we do not hold the older copy, but rather parse the entire inode table block whenever it is written. Note that some inodes are not in use, indicated by their non-zero deletion time field, and there is no need to parse them any further. For the rest, we extract the fields necessary to us from the new inode, e.g., the inode type and the list of block it contains, and update our data structures accordingly.

In order to identify when inodes and blocks are added to the system, IDStor maintains intermediate state machines for inodes and blocks that were encountered but are not yet *valid*, i.e., for which not all commands required for complete inference have arrived. We first present the state machine logic assuming the iSCSI commands that compose a single file-level command are not interleaved with other commands. Then, we show that our parsing can be affected by certain cases in which the commands are interleaved, and describe how this can be handled.

1) ***Inferring a single host-level command****:* In this section, we assume that the block-level commands related to a single host-level command are not interleaved with other commands. Clearly, this assumption is not realistic and will be relaxed in the next section. IDStor keeps track of the status of each inode and block for their validity since only the valid inodes and data blocks are considered in the inverse block-to-file mapping. It gathers the relevant iSCSI commands to infer the host-level commands and to update the state machines and the file view. It is enough to parse only the write commands, and only part of those are necessary. In addition, pure data blocks need not be parsed at all. Only metadata and fake data blocks, such as indirect addressing blocks, are parsed. Data blocks that belong to directories need to be parsed only in case we need to provide an inode-file name mapping (see below).

We maintain three data structures for reaching the required metadata information, as follows: the *inodes hash table* holds all of the inodes that exist in the system, *block-to-inode BTree* holds the numbers of all the allocated data blocks in the file system, their role in the file system (e.g., pure data, or indirect addressing), and their owning inodes, and the *data block tracker list* holds the data blocks that were encountered so far and that cannot yet be associated with any of the inodes with certainty. The flowchart in Figure 5 assumes that the arrived iSCSI command belongs to a single host-level command. It identifies the block number and type, and by parsing its content and using the information in the data structure it sets the associated inode or data block validity as described below. The following states are determined according to the content of the arriving metadata blocks and these data structures.

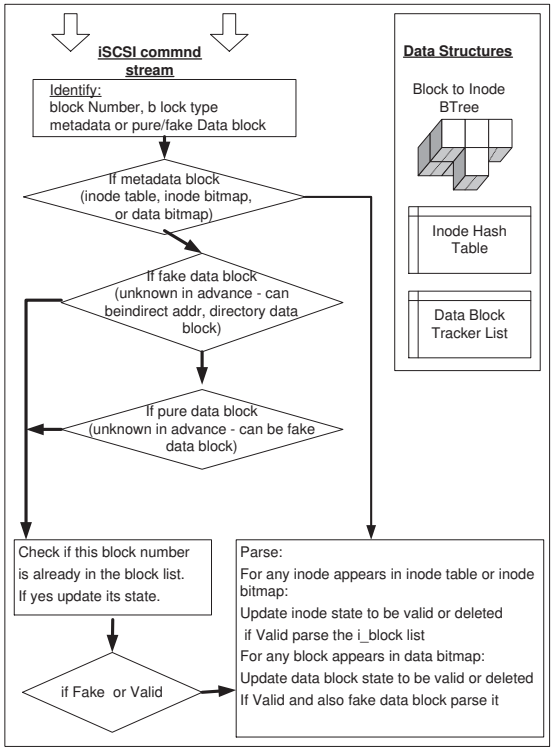

Fig. 5. An algorithm to infer both a single host-level command and the validity of the associated inodes and data blocks. It assumes the commands arrive in order and gathers the commands that indicate the states of the associated inode or data block until its state is resolved (valid or deleted). The command is parsed if it is a metadata command, otherwise it is kept and parsed only when it is declared as valid.

- An inode is considered to be *valid* at the storage level only when all the iSCSI commands that are relevant to it have arrived and were written in the storage disk. IDStor considers an inode to be valid after receiving and parsing the following iSCSI commands (in any order):
  - The inode was written in the inode table, with a *dtime* field equal to zero (indicating this inode has not been deleted).
  - The corresponding part of the inode bitmap was written, in which the bit for this inode was set to true (indicating the inode is not free).

  Whenever an inode becomes valid, the "New File" or "New Directory" host-level command is announced, according to the type of the inode.
- An inode is considered *deleted* after receiving the following commands:
  - The inode was written in the inode table, with a non-zero *dtime* field (indicating this inode has been deleted).
  - The corresponding part of the inode bitmap was written, in which the bit for this inode was set to false (indicating the inode is free).

  Whenever an inode is deleted, the "Delete File" or "Delete Directory" host-level command is announced.
- A data block is *valid* after receiving the following block-level commands:

  - A write command that updates this data block.
  - A write command that updates the address to this block. There are two ways to point at a block: a) A valid inode was written to the inode table, with this data block listed in the *i_block* field (i.e., the inode points directly at this data block), or b) a block known to be an indirect addressing block was written, with this block listed in it.
  - The corresponding part of the data bitmap was written, in which the bit for this block was set to true (indicating the block is not free).
- A data block is declared *deleted* after it was deleted in the data bitmap only. Looking for any other evidence for this deletion requires storing the list of the blocks that belong to a certain inode. This information is kept in the file system and requires additional memory. For our inverse map it suffices to keep only the block to inode pointer.

While this prototype deals with the iSCSI protocol, switching to a different block-level protocol requires adapting a very small part of the process, namely the high parsing layer that parses the block-level protocol and extracts the sequence of block-level operations from it. The described flow above assumes that the arrived iSCSI commands belong to a single host-level command without interleaving of commands that may change the ownership and consequently the type of a data block. The next section provides a modified algorithm that considers the usual scenario with an interleaving and an order change of the iSCSI commands.

*2) **Inferring host-level commands assuming interleaving of commands**:* The file system information can be delayed in cache at the host and flushed to the disk or storage at any order, and usually after some delay. Although we maintain asynchronous validity state machines, there are still several ordering patterns that might lead to an ambiguous inode identification by our online parser. For example, consider the following scenario:

Assume block 1000 is currently assigned to inode inode_a. Inode inode_a is truncated, thus block 1000 is freed but this fact was not flushed to disk yet. Now a large amount of data is added to inode inode_b, such that it needs an indirect addressing block, and block 1000 is assigned for that. Two things should happen: a) block 1000 has to be written with the indirect data, and b) the inode of inode_b has to be written to update that 1000 is an indirect block belonging to it.

If b) happens before a), our online parser will realize that block 1000 is owned now by inode_b and not by inode_a and upon the arrival of block 1000, it will know it is an indirect block (fake data block) and will parse it appropriately. However, if a) happens before b), our inverse map still holds block 1000 as a valid pure data block belonging to inode_a, so we ignore it. Now when b) happens we mark 1000 as indirect, but wait for the data (which has already arrived, and will therefore not arrive again).

In order to be able to treat such scenarios correctly, our algorithm must not rely on the order in which the block commands are flushed to disk. Specifically, we need to use additional information, and should parse arriving data blocks only after verifying their validity and ensuring which file owns them and with what role. In other words, we must parse the data block after parsing the relevant metadata blocks. Figure 6 a1) illustrates this scenario assuming interleaving arriving commands and a2) shows the parsing order when the data blocks are ignored for the validity state machine and their parsing is delayed until the end of the metadata period. 6 b1) and b2) illustrate the order arrival and the parsing order in ext3 when there is no way to differentiate between pure and fake blocks and both are viewed as data blocks. Thus both types of data blocks, pure and fake, should be delayed until the parsing of the metadata blocks (Figure 6 b2)). This is due to the fact that the only way to differentiate between pure and fake blocks at the block level is by identifying their correct inode association and their role within that inode. As a result, we must identify a self contained epoch in an arriving stream, containing a set of data blocks and all their associated metadata blocks, or in other words all the block writes that are associated with one or several host-level commands.

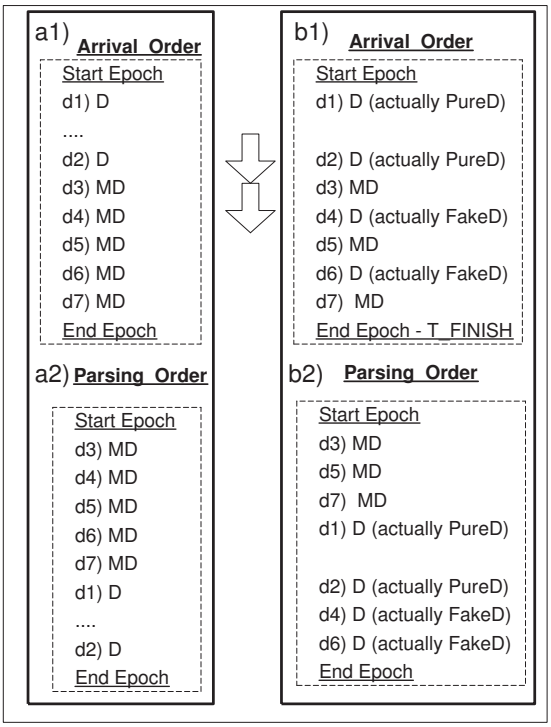

Fig. 6. (a1) presents a possible arrival order when the metadata blocks are interleaved with the data blocks (a2) shows the parsing order when the metadata blocks are parsed immediately and the data blocks are ignored for the validity state machine and their parsing is delayed until the end of the metadata period.(b2) illustrate the order arrival and the parsing order in ext3 when there is no way to differentiate between pure and fake block and both are viewed as data blocks. Thus both types of data blocks, pure and fake, should be delayed until the parsing of the metadata blocks.

Unfortunately, in a file system without journaling there is no way to know in advance when the relevant metadata per each data block will appear, and thus there is no way to know until when to delay the parsing of a data block. The situation is different when adding journaling modes to a FS, like in ext3 [6]. Not surprisingly, the order in which commands are flushed to disk is affected by the journaling type that is

used. ext3 adds three journaling methods to ext2, Journal, Ordered and Writeback modes. Each is different in the type of information that is kept in the journal and can be recovered in case of a failure.

In the *Journal* mode of the ext3 journaling, all file system data and metadata changes are logged into the journal. This mode minimizes the chance of losing the updates made to each file, but it requires many additional disk accesses. The *Ordered* mode only logs changes to file system metadata, but before performing the actual metadata modifications, it requires all pure data blocks to be flushed to disk. The *Writeback* mode logs only file system metadata, without any ordering guarantees.

Each journaling transaction consists of several atomic operations, where each atomic operation logs all the relevant blocks relative to a single high-level change of the file system. The transaction stops accepting new handles to log when either a fixed amount of time has elapsed, typically 5 seconds, or when there are no free blocks in the journal left for a new handle.

When the ordered mode is used, the sequence of events for committing a transaction is as follows. First, all pure data blocks are flushed to disk, then a journal entry is written, recording the fact that the transaction changed status to 'T_FINISHED'. Finally,the metadata is flushed to disk. Metadata in this context refers both to metadata blocks and to fake data ones. Thus, with a minimum amount of parsing of the journal, namely, finding the 'T_FINISHED' entries, one can identify the point in which the flushing switches from data blocks to metadata ones (both metadata and fake data). With this information, the self-contained epoch can be identified.

*3) **inode-to-filename mapping**:* The framework described above can map a block to an inode and also to a file name in case there is only one file name per an inode. When using hard links and there are several different file names to the same inode, another level of translation and another set of state machines has to be maintained. Such an additional level of inverse mapping requires a lot of memory since it has to hold the directory hierarchy information with all the filenames. The decision whether to deploy it depends on the application, and whether it has to answer the question: "What was the exact file name that is related to a certain inferred host-level command, given an inode?". In Section V we will discuss briefly how it can be used by the security layer of IDS for block-based storage.

For now, we describe the involved iSCSI commands when adding and deleting a hard link and what are the additional challenges in its deployment. In this paper, we only describe the challenges in handling hard links, and general ideas toward handling them. We leave the details of this solution as future work. It is hard to infer the addition or the deletion of another file name to an inode because of the following reasons.

The files that belong to a directory are listed in data blocks that are associated with the directory inode. These data blocks hold an unsorted list of records, one record for each file name. The record keeps the filename and its inode ID. A directory that has a large number of files can span over many data blocks. In order to discover whether a new file name was added or deleted to the list, we need to keep a copy of the entire directory hierarchy at the listener. A creation of a new link at the host-level is translated into two block-level write command: a) A write to a data block that belongs to the parent directory with the additional name (hard link) added to the list. b) A write to the inode table, in which the links counter of the inode is updated. To find out which file name was added and to which inode, we can traverse the list written to the directory data block, and add any previously unknown file name to our data structures.

Identifying the deletion of a hard link is somewhat more challenging, and requires keeping the current version of the directory data blocks at the listener. A deletion of a hard link is also translated into a block level commands similarly to addition of a link, i.e. a) A data block that belongs to the parent directory is written, this time omitting this file name and inode, and b) A write to the inode table, in which the links counter of the inode is updated. Note that in this case, the deleted filename record will simply not appear in the list. Therefore, for identifying that a file name was listed there before, we must keep some information on which file names are listed, and where. One option, heavy on memory requirements, is to simply keep the current version of all directory data blocks. This additional deployment would enable us to infer file-level commands by their file names and not only by their inode number.

B. Data Structures

Unlike previous approaches, IDStor works in a listener mode such that it does not interfere with the ongoing I/O requests. The difficulty when working in a listener mode is that each captured write command overwrites the content of the block, thus preventing the comparison of the new content with the old one. In a naïve solution, the entire disk content could be stored at the listener, which would require an amount of memory equal to the size of the disk. Instead, we provide a reliable map with significantly less memory requirements, by maintaining an efficient data structure and keeping track only of required metadata information. For this purpose, we keep three data structures, as follows: inodes hash table, block-to-inode BTree and data block tracker list.

Inodes hash table This hash table holds all the inodes that exist in the system, valid and semi valid. For each inode, we keep an inode structure that reflects the inode state, as described in III-A. An inode is deleted from the list whenever it is inferred as deleted. The keys in this table are the inode numbers and the values are pointing to the appropriate inode structures.

Block-to-inode BTree Holds the numbers of all the allocated data blocks in the file system, their role in the file system (e.g., pure data, or indirect addressing), and their owning inodes. The Block-to-inode map is built as a ranged BTree in a way that given a block number, its owner inode can be fetched efficiently. The keys in this BTree are consecutive ranges of data blocks that carry the same role and are owned by the same inode, and the values are the inode structures. This tree contains only valid blocks, pointing to valid inodes.

Data block tracker list This list holds the data blocks that were encountered so far (during a period of time) and that

cannot yet be associated with any of the inodes with certainty. For each such data block, we temporarily keep its state and its content until its ownership and type are verified. We keep the actual data since in cases where a block turns out to be a fake data block, its data has to be parsed. This content can be deleted once it is parsed.

Note that even though the file system metadata representation is different from one file system to another, our data structure is mostly independent of the file system. The data structures described above are first initialized and then updated online. At application initialization, an offline builder module traverses over the file system metadata that is stored at the target disk and inserts each encountered inode, along with its blocks, into the data structures. The data structures are then updated online as described in Section III-A. For an extent based file-system, such as ext4 or ntfs the described data structures are correct but not optimal.

Memory Consumption We now study the extent to which our data structures save memory compared to the basic approach.

The required amount of memory is divided into information that is required by the block-to-inode mapping and information that is required by inode-to-file mapping and is also specific to the security application layer. The data structures, as described in III-B, are used for the block-to-inode map. On disk, most of the file system metadata space is occupied by the inode table information where a structure is held per each inode (free and not free). In our case, for the purpose of keeping an updated list of inodes and their state, we must have an inode structure for each used inode. The inode structure should hold only those fields in the file system data structure that are necessary for the block-to-inode inference.

Regarding data blocks, both pure and fake ones, there is no need to keep the data itself. It suffices to maintain a list of block numbers, each with a pointer to the owning inode. Therefore, we keep at most 8 bytes per each data block. For example, for a logical block of size 4KB, since we keep at the listener only 8 bytes per block, the ratio between the amount of memory we require for data blocks and their actual size on disk will be the 4KB divided by 8, which is 500.

Note that this is an overestimated computation due to the range-aware BTree data structure, in which the keys represent a series of consecutive blocks rather than a single block number.

Additional memory should be kept for the data block tracker list. Each block is kept for a short interval, as short as the time it takes for the journal mechanism to be flushed to disk. During this interval, we need to keep the encountered data blocks (both pure and fake). For example, consider a flush period of 5 seconds, 2500 data block commands per second, each block size of 4K bytes. In this case, we need to keep 4000*5*2500 = 50Mbytes for temporary data.

IV. Evaluation

Our tests were designed to evaluate the feasibility of the block to file translation and the quality of our inference mechanism. We ran several scripts and "live" benchmarks to check the consistency of the inverse data structure with regard to the file system and to examine whether each host-level command was inferred. We built a system as depicted in Figure 1 using three x86 machines – a host, an iSCSI target and an IDStor listener. The listener captures all traffic in and out of the target via traffic mirroring at the switch.

An example of a test is provided in Table II. The left column lists a simple shell script that creates a three-leveled directory structure with some files, as illustrated in Figure 7. It then appends and truncates some data from each of the files. The right column presents the inferred commands as they are printed by the IDStor listener application. For clarity, we do not show the printing of each iSCSI identification, but only the inference result. The test consists of the following four parts.

1) Creating the directory structure and files. A small amount of data is written to each new file, causing a single block of data to be allocated for each file. Thus, for each directory creation and file creation the listener reports inferring that the new directory or file exists, and a single data block being assigned to it. For clarity, we list some of the inference printouts aligned with the actual host commands, but in fact all inference printouts are a result of the sync at the end of the batch that caused all modifications to be flushed to disk, thus to be captured and parsed by the listener.
2) Appending some data to files. The `dd` command is used in order to write some random data to two files, of lengths 12 KB and 8 KB, which caused three or two new data blocks to be assigned to these files, respectively.
3) Truncating a file to the length of 4500 bytes, thus leaving it with two blocks of data, and releasing the rest.
4) Removing the entire directory structure, which causes inference of all data blocks being freed, and all directories and files being deleted.

In addition to running script examples, several **benchmarks** were carried out, in which compilation of a large libraries was executed on the host and captured by the listener. During this test, 200 object files were truncated and then re-created, and 2.5MB of data was written to them. Some of the files were large enough to require one level of indirect addressing. In order to check correctness of the file-level inference, i.e., consistency of our inferred data structure with regard to the actual file system, at the end of the test we built another inverse block-to-file data structure according to the actual state of the file system structure at that moment. This data structure was then compared to the one we maintained throughout the test, and found to be identical. This shows that indeed all commands were correctly inferred and that the data structure has been updated appropriately. Note that no files are created and then deleted during the test, nor data written and then freed, thus the state of the file system at the end of the test reflects all the commands that were executed in it.

The goal of our evaluation was to verify the feasibility of our inference algorithm rather than evaluate its performance or CPU utilization. Nonetheless, it is important to note that the functionality that is added to the block-to-file mechanism

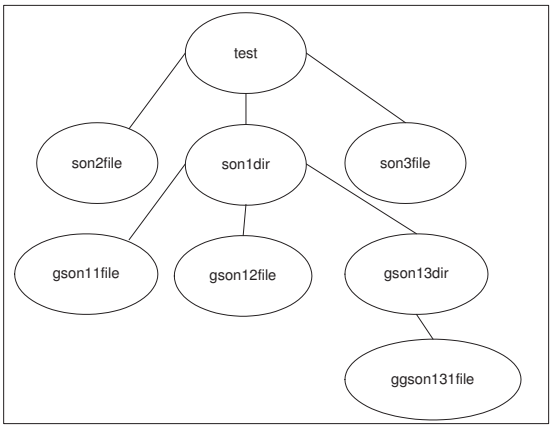

Fig. 7. Directory hierarchy created by the test Shell script (Table II,left column).

is mainly composed of updating the data structure; it is lightweight and can handle the arriving stream of commands at line rate.

V. IDS FOR BLOCK-BASED STORAGE (IDSTOR) - SECURITY LAYER

This work provides only a preliminary design of the security layer of IDStor that given the block-to-file layer is capable of detecting both block-level and file-level threats. This section presents the various threats that can be detected at storage level and how the block-to-file features we presented can contribute to detect them. Clearly, the iSCSI parser layer enables the detection of block-threats, i.e. violations of the iSCSI protocol. Specific examples are checking for 'too long names' in the iSCSI login stages that may cause buffer overflows, or detecting data that is sent in chunks larger than the negotiated maximum.

Detecting more sophisticated threats requires file-level knowledge. Several papers [4], [7], [8] present different types of malicious threats that can be handled at the storage as presented in the list below. Most of these threats belong to the Tripwire-like threat model [1] and are usually handled by the Tripwire intrusion detection tool running at the host.

- **Unexpected change of system files:** Data or meta-data changes to files that administrators expect to remain unchanged (except during explicit upgrades). Examples of such files include system executables and scripts, configuration files, system header files and libraries.
- **Unexpected file access pattern:** Suspicious patterns of access to certain files or to the file system, in particular updates. Specific examples to this group of threats are non-append modification of system log files and reversing of inode times.
- **Denial-of-service (DoS) attacks:** An attacker may disable specific services, or entire systems, by allocating all or most of the free space or by allocating many inodes or other metadata structures. When the system reaches predetermined thresholds of allocated resources and allocation rate, warning the administrator is appropriate even in non-intrusion situations – attention is likely to be necessary soon.
- **Suspicious content appearance:** The most obvious suspicious content is a known virus or rootkit, detectable by its signature.
- **Hidden "dot" files:** Hidden files have names that are not displayed by normal directory listing interfaces, e.g., ls, and their usage may indicate that an intruder is using the system as a storage repository, perhaps for illicit or pirated content. A large number of empty files or directories may indicate an attempt to exploit a race condition by inducing a time-consuming directory. Another kind of suspicious files are one with names that look like the default '.', but with an additional space '. '
- **Snooping on deleted storage blocks:** In most file systems, storage blocks are allocated to files on demand. When a file is deleted, the storage block contents are not necessarily erased. Rather, most file systems implement file deletion simply by erasing the file from the directory and deleting the file inode. Thus, data contents can be left un-erased in deleted, and now free, storage blocks. By accessing these storage blocks, it is possible for an attacker to gain access to sensitive data.
- **Data leakage prevention** Checks whether someone read data that he is not authorized to. Specifically, it prevents the writing of sensitive data over non secured machines or disks.

In order to detect the above types of threats, the security layer has to be informed of the following file-level operations:

- Modification of the contents of an existing file (or inode). In the block-level, this may manifest in the form of modification of a data block already associated with the inode, an addition of a new block to it, or a removal of a block from an inode.
- Modification of the metadata of a file (or inode), e.g., file deletion, renaming, and addition or deletion of hard links.

The security layer itself also has to maintain certain data for each monitored file. For example, consider a log file, for which an append-only rule is applied, calling for detection of overwrites of existing data (as opposed to appends to the log). In order to detect whether a write to the file indeed modifies the last data block, it needs to know which of the blocks belonging to the file is the last one. Moreover, the data in the last block might also be overwritten, therefore the security layer needs to keep enough information on this specific block in order to infer, given the new data written to it, whether this is an append or an overwrite operation. This may be implemented, for example, by keeping a hash value of the existing data, along with its length. Upon arrival of a write operation to this block, the hash value of the relevant part of its data may be computed and compared with the existing value.

Rules in an IDS are typically pairs of the form $\langle identifier, rule \rangle$, where $identifier$ identifies a certain file or group of files, either by their filenames or by their inode IDs. $rule$ is the rule to be applied to this file or group.

Security administrators clearly want to specify rules using

Shell Script @ Host	IDStor Infers Host-level Commands @ Listener
#! /bin/bash; cd /mnt/tmp	
mkdir test; cd test	14:57:51 INFER: new dir /test (inode 310689)
mkdir son1dir	14:57:51 INFER: block 630784 valid belongs to /test (inode 310689) 14:57:51 INFER: new dir /test/son1dir (inode 310690)
echo "new file" > son2file	14:57:51 INFER: new file /test/son2file (inode 310691)
echo "new file" > son3file	14:57:51 INFER: new file /test/son3file (inode 310692)
echo "new file 1 sub son1dir" > son1dir/gson11file	14:57:51 INFER: block 634880 valid belongs to /test/son1dir (inode 310690) 14:57:51 INFER: new file /test/son1dir/gson11file (inode 310693)
echo "new file 2 sub son1dir" > son1dir/gson12file	14:57:51 INFER: new file /test/son1dir/gson12file (inode 310694)
mkdir son1dir/gson13dir	14:57:51 INFER: new dir /test/son1dir/gson13dir (inode 310695)
echo "new file sub gson13dir" > son1dir/gson13dir/ggson131file	14:57:51 INFER: block 638976 valid belongs to /test/son1dir/gson13dir (inode 310695) 14:57:51 INFER: new file /test/son1dir/gson13dir/ggson131file (inode 310696)
sync echo "Done creating directories and files."	14:57:51 INFER: block 628736 valid, belongs to /test/son2file (inode 310691) 14:57:51 INFER: block 628737 valid, belongs to /test/son3file (inode 310692) 14:57:51 INFER: block 628738 valid, belongs to /test/son1dir/gson11file (inode 310693) 14:57:51 INFER: block 628739 valid, belongs to /test/son1dir/gson12file (inode 310694) 14:57:51 INFER: block 628740 valid, belongs to /test/son1dir/gson13dir/ggson131file (inode 310696)
echo "Appending more data..."	
dd if=/dev/urandom of=son3file bs=4096 count=3 oflag=append conv=notrunc	15:01:07 INFER: block 628741 valid, belongs to /test/son3file (inode 310692) 15:01:07 INFER: block 628742 valid,belongs to /test/son3file (inode 310692) 15:01:07 INFER: block 628743 valid,belongs to /test/son3file (inode 310692)
dd if=/dev/urandom of=son1dir/gson11file bs=4096 count=2 oflag=append conv=notrunc	15:01:07 INFER: block 628744 valid, belongs to /test/son1dir/gson11file (inode 310693) 15:01:07 INFER: block 628745 valid, belongs to /test/son1dir/gson11file (inode 310693)
sync ; echo "Done appending data"	
echo "Truncating some data..."	
truncate son3file 4500	15:01:42 INFER: block 628742 no longer in use 15:01:42 INFER: block 628743 no longer in use
sync ; cd ..	
echo "Deleting everything..." rm -rf test	15:01:59 INFER: block 628736 no longer in use 15:01:59 INFER: block 628737 no longer in use 15:01:59 INFER: block 628738 no longer in use 15:01:59 INFER: block 628739 no longer in use 15:01:59 INFER: block 628740 no longer in use 15:01:59 INFER: block 628741 no longer in use 15:01:59 INFER: block 628744 no longer in use 15:01:59 INFER: block 628745 no longer in use 15:01:59 INFER: block 630784 no longer in use 15:01:59 INFER: block 634880 no longer in use 15:01:59 INFER: block 638976 no longer in use 15:02:00 INFER: file /test/son2file (inode 310691) deleted 15:02:00 INFER: file /test/son3file (inode 310692) deleted 15:02:00 INFER: file /test/son1dir/gson11file (inode 310693) deleted 15:02:00 INFER: file /test/son1dir/gson12file (inode 310694) deleted 15:02:00 INFER: file /test/son1dir/gson13dir/ggson131file (inode 310696) deleted 15:02:00 INFER: dir /test/son1dir/gson13dir (inode 310695) deleted 15:02:00 INFER: dir /test/son1dir (inode 310690) deleted 15:02:00 INFER: dir /test (inode 310689) deleted

TABLE II
EXAMPLE

filenames (or directory names), thus when using inode IDs as identifiers, there is a need to translate the filenames into inodes. For such files that already exist in the system, their associated inodes can be fetched easily from the file system. However, for files that do not yet exist, the rule will have to be applied to the inodes as soon as the inodes are created. Therefore, we need to reliably identify filenames and full paths of inodes as they are created or renamed, even if the rules are identified using inode IDs. Also, consider a scenario where a directory is renamed, and thus affects the full path of all files underneath it. To handle such cases, we must maintain a memory-consuming inode-to-filename mapping, as well as keep the whole directory hierarchy, i.e. the list of inodes within a directory, a list that requires even more memory.

VI. RELATED WORK

Several works have suggested to add a file system view to a block-based storage controller in order to handle file-level threats. However, all these works are within the controller's I/O path, and not at a listener. Banikazemi *et al.* [2] provide block to file mapping for a real time intrusion system in the SVC product by translating each file-based rule into a block-level rule that is attached to the appropriate block. This work does not provide file-level inference as we are doing. More than that, they can not handle a rule whenever it can not be translated to block-level commands. They build their mapping inside the I/O path, where write operations of new content can be delayed in case there is a need to read the old content before overwriting it. They use the ext2 file system. Zhang *et al.* [3]

maintain a sector-to-file mapping table at a virtual machine in the host. They use this table to update a sector-base rule set that is located at the storage. The goal of the sector-base rule set is to identify intrusion at the storage level. They are not maintaining file-view at the controller and their IDS will not be effective in case of a compromised host. Zhang *et al.* [9] combine intrusion detection with data recovery. Specifically, they detect the intrusion at the file-level in the host and trigger recovery process at the block-level in the SAN controller without dealing with an inverse map.

Several works have dealt in general with obtaining file-awareness at the block storage controller, not necessarily for intrusion detection. Arpaci-Dusseau *et al.* [10] discuss the past, present and future directions in the design and implementation of smarter storage systems. Sivathanu *et al.* [11] provide important and detailed guidelines for block-to-file mapping in different file system, and with regard to several issues and metrics. They check whether the liveness property of a block, i.e. knowing at any point of time whether this block is free or owned by a certain inode, can be identified at the block-level storage only by an implicit capture of the packet blocks (without extra information from the file system). They are not dealing with the specific implementation at a listener appliance for detecting intrusions. Bairavasundaram *et al.* [12] by observing which files have been accessed through updates to file system meta-data, construct an approximate image of the contents of the file system cache and uses that information to determine the exclusive set of blocks that should be cached by the array. Sivathanu *et al.* [13] study the applicability of semantically smart disk technology underneath database management systems. Li *et al.* [14] develop an algorithm to correlate blocks in storage systems for caching efficiency. Gunawi *et al.* [15] introduce a new reliability infrastructure for file systems called I/O shepherding. I/O shepherding allows a file system developer to craft nuanced reliability policies to detect and recover from a wide range of storage system failures. They incorporate shepherding into the Linux ext3 file system through a set of changes to the consistency management subsystem. Macko *et al.* [16] implement back-references, which is a metadata that maps block numbers to the data objects that use them. Yadwadkar *et al.* [17] analyze traces of file-level traffic in order to understand higher-level semantics of the data.

Finally, [4], [7] and [8] discuss the importance of storage-based intrusion detection and describe the threats that are expressed at the storage but can harm the host components as well. Most of the threats belong to the Tripwire-like threat model [1]. They are usually handled by the Tripwire intrusion detection tool at the host. One of the key factors for a storage-based IDS in order to handle such threats is to have a semantic file view. Factor *et al.* [18] present an approach which leverages the OSD (Object-based Storage Device) security model to provide a logical, cryptographically secured, in-band access control for today's existing block-level devices.

VII. Concluding Remarks and Future Works

This research deals with how to provide a better data protection at the block storage controller by enabling intrusion detection at a listener appliance. To handle a wider range of threats, we deploy an online mechanism that infers file-level commands. In order to answer questions as "given a block number, which inode owns it?" and "what is the file name of this inode?", we explored the file system layout, implemented the translation mechanism and discussed its limitations . Our implementation and discussion provide a proof-of-concept and open the door for additional research directions in the storage-based IDS field.

This work deals with detecting intrusions and sending relevant alerts. We do not deal with the questions of who actually receives this alert and how. The question of who the alert should be sent to is related to the question of what exactly the system protects. For example, if the goal is to protect the storage system itself, the alert should be sent to a storage administrator. If, on the other hand, the host is the concern, then some host administrator should be alerted.

We also do not consider prevention mechanisms. In network IDS, dropping suspicious packets usually suffices. Host-based IDS triggers anti-virus or file system rollback mechanism upon violation. In storage IDS, however, this seems not to be the case. For example. dropping packets might cause unwanted file system inconsistency. One possible way of dealing with prevention is by using snapshots mechanisms. Consistent snapshots may be kept over time, such that the last known snapshot can be reverted in case of an intrusion. Thus, another possible and very interesting research work is how to prevent intrusions to storage in addition to their detections.

Given today's high distribution of web-based threats that inject malicious content into database tables, dealing with the detection at the block-level storage is essential, though very challenging task. One way to extend this work, is providing an additional level of mapping where each file name is mapped to its corresponding table in the database to detect database-level intrusions, assuming the tables are mapped to files, or map the blocks to tables in case the database tables are mapped directly to raw blocks.

The block-to-file mechanism can be used by applications other than security, applications that can benefit from the file-view at block storage controller such as performing file-level replication or providing file-level monitoring at the storage controller. Moreover, this research focuses on the ext3 file system as a representing journaling filesystem. An extended research over other filesystems, can make it more robust and effective.

References

[1] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in *In Proceedings of the 2nd ACM Conference on Computer and Communications Security*. ACM Press, 1993, pp. 18–29.

[2] M. Banikazemi, D. Poff, and B. Abali, "Storage-based intrusion detection for storage area networks (sans)," in *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 118–127.

[3] Y. Zhang, Y. Gu, H. Wang, and D. Wang, "Virtual-machine-based intrusion detection on file-aware block level storage," in *SBAC-PAD '06: Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing*, 2006, pp. 185–192.

[4] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. N. Soules, G. R. Goodson, and G. R. Ganger, "Storage-based intrusion detection: Watching storage activity for suspicious behavior," in *In Proceedings of the 12th USENIX Security Symposium*, 2003.

[5] D. Bovet and M. Cesati, *Understanding the Linux Kernel, Third Edition*, 3rd ed. O'Reilly Media, Inc., 2005.

[6] S. C. Tweedie, "Journalling the ext2fs filesystem," in *Proceedings of the 4th Annual LinuxExpo*, 1998.

[7] R. Hasan, S. Myagmar, A. J. Lee, and W. Yurcik, "Toward a threat model for storage systems," 2005.

[8] J. L. Griffin, A. Pennington, J. S. Bucy, D. Choundappan, N. Muralidharan, and G. R. Ganger, "On the feasibility of intrusion detection inside workstation disks," 2003.

[9] Y. Zhang, H. Wang, Y. Gu, and D. Wang, "Idrs: Combining file-level intrusion detection with block-level data recovery based on iscsi," in *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, 2008, pp. 630–635.

[10] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, L. N. Bairavasundaram, T. E. Denehy, F. I. Popovici, V. Prabhakaran, and M. Sivathanu, "Semantically-Smart Disk Systems: Past, Present, and Future," *Sigmetrics Performance Evaluation Review (PER)*, vol. 33, no. 4, pp. 29–35, March 2006.

[11] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Life or death at block-level," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 26–26.

[12] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Munich, Germany, June 2004.

[13] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Database-Aware Semantically-Smart Storage," in *Proceedings of the Fourth USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[14] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-miner: Mining block correlations in storage systems," in *In Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST 04)*, 2004.

[15] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Improving File System Reliability with I/O Shepherding," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007, pp. 283–296.

[16] P. Macko, M. Seltzer, and K. A. Smith, "Tracking back references in a write-anywhere file system," in *In Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST 10)*, 2010.

[17] N. J. Yadwadkar, C. Bhattacharyya, K. Gopinath, T. Niranjan, and S. Susarla, "Discovery of application workloads from network file traces," in *In Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST 10)*, 2010.

[18] M. Factor, D. Naor, E. Rom, J. Satran, and S. Tal, "Capability based secure access control to networked storage devices," in *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, 2007.