

Block Storage Listener for Detecting File-Level Intrusions

Authors:

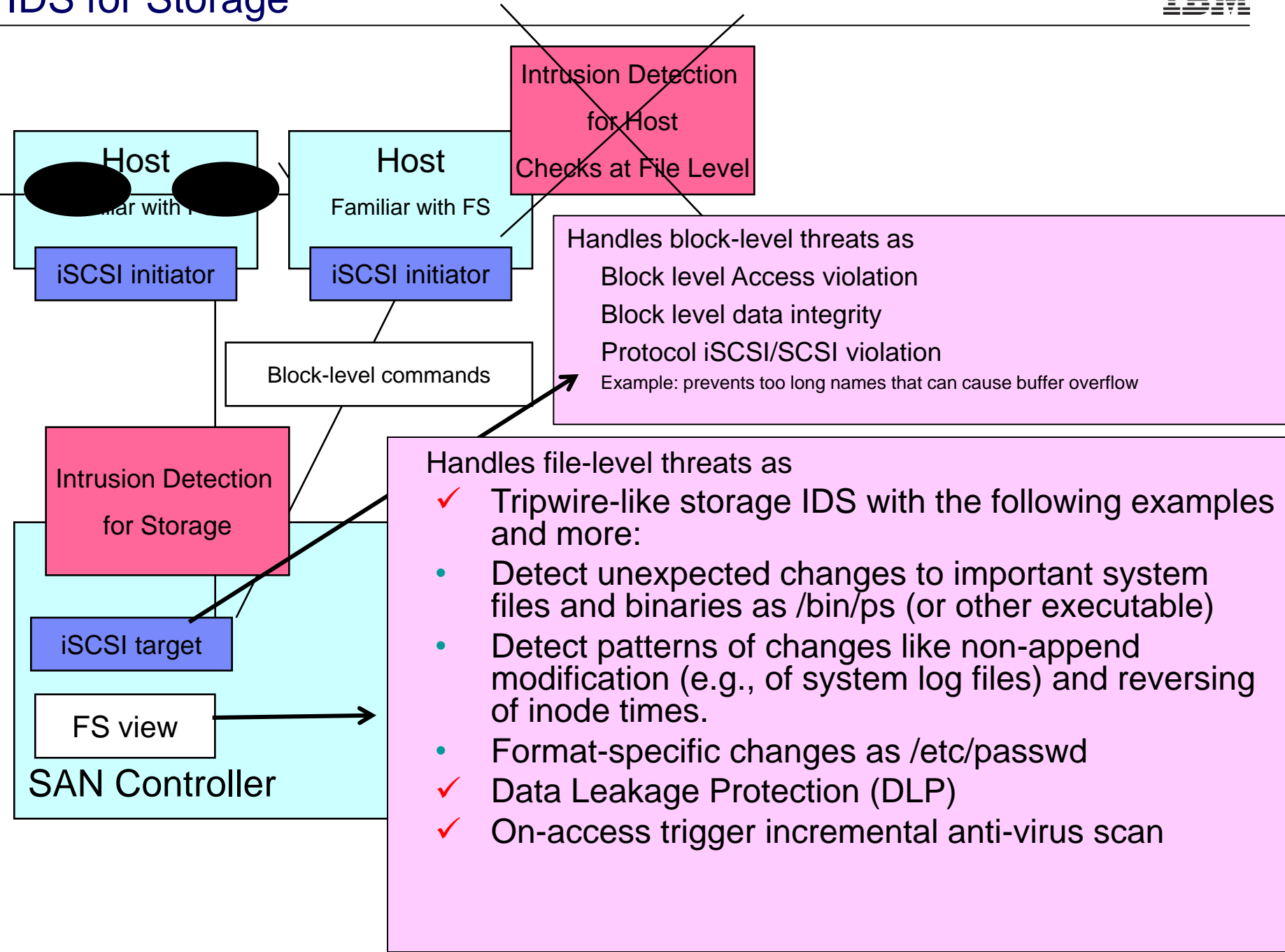
Miriam Allalouf, Itai Segall, Muli Ben-Yehuda and
Julian Satran

IBM – Haifa Research Labs

MSST 2010, May, 2010

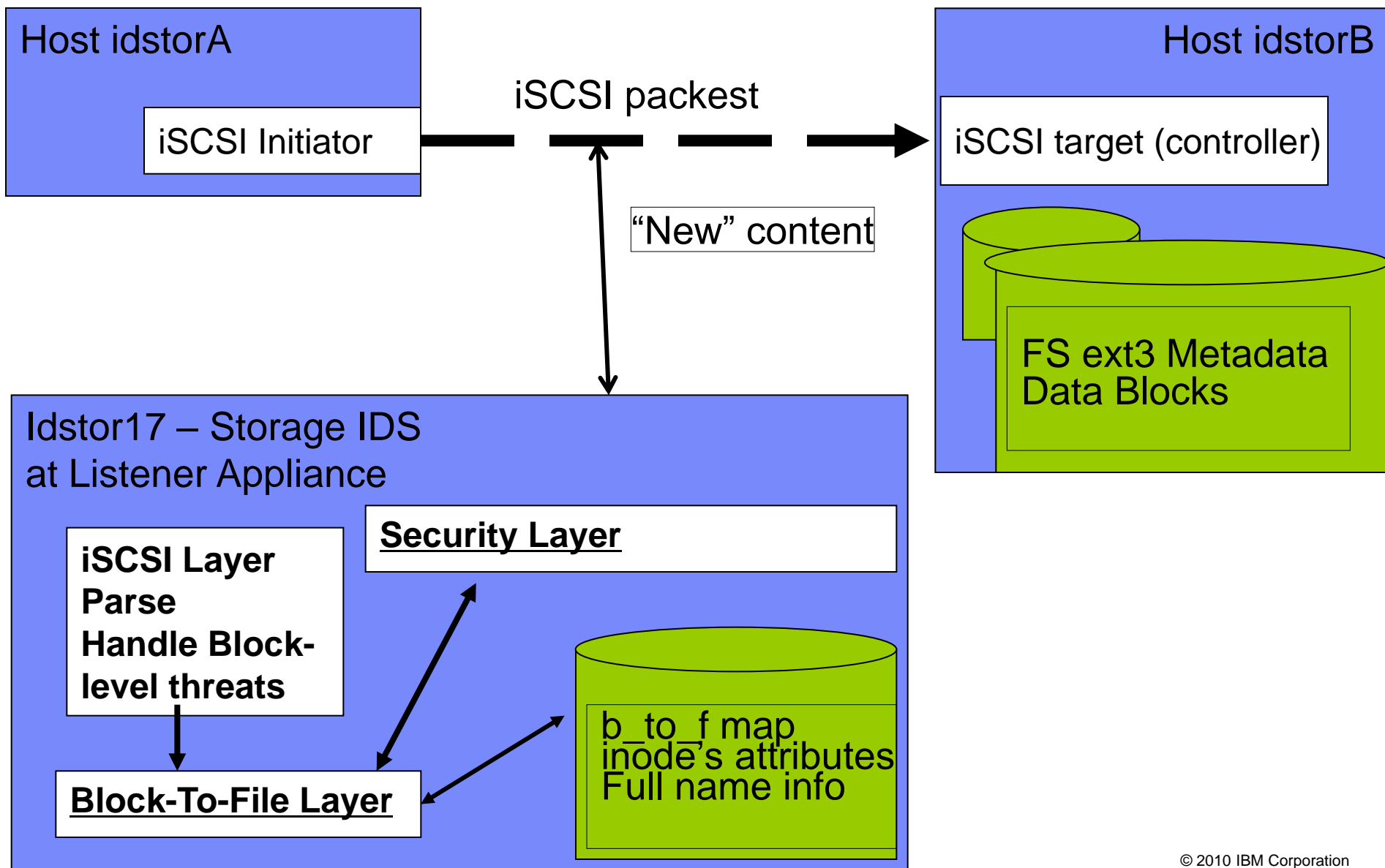
Intrusion Detection System

- **IDS** - Intrusion detection system in 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 IDS** - Sensors are located at points in the network to be monitored.
 - Captures all network traffic and analyzes the content of individual packets in order to detect malicious traffic.
 - **Host-based system IDS** - Sensors consist of a software agent that monitors all activity of the host on which it is installed, including file system, logs and the kernel.
- **Our Goal:** To build **Storage-Base IDS** - Providing security features into the storage controller is essential when hosts are compromised or in case when multiple hosts share an attack that can be detected only by the central storage.
 - There are no storage-based IDS at the SAN block controller that can alert the administrator or the hosts on the appearance of suspicious events.
 - The very few storage systems that do maintain online IDS in storage systems are accessed via file-level protocols, such as CIFS or NFS.



- **Build block-to-inode inverse map**
 - Current file-system metadata structure enables answering host-level requests:
 - ✓ list the blocks that belong to a certain inode
 - ✓ list the files that belong to a certain directory
 - In order to find file-view having block-level commands, we need to answer questions such as:
 - ✓ given a block number, which inode owns it?
 - ✓ what is the file name and parent directory of this inode?
 - **Online infer file-level commands** from a block-level command sequence
 - **Find file-level access pattern** by tracking block-level access pattern
- → Enables File-level intrusion detection

- **Building Storage IDS at listener appliance**
 - IDS limitations within the controller's I/O path:
 - Adding SW to the I/O path of a controller is a complicated and error-prone task, with heavy development expenses.
 - CPU capacity at the controller is designed to handle the arriving I/O requests and may not be able to perform additional computation tasks
 - Offload to a listener appliance solves the problem
 - As a listener it is easier to market,
 - The solution is not controller-specific and can be used with many different kinds of controllers.
 - A centralized listener appliance per several parallel storage connections

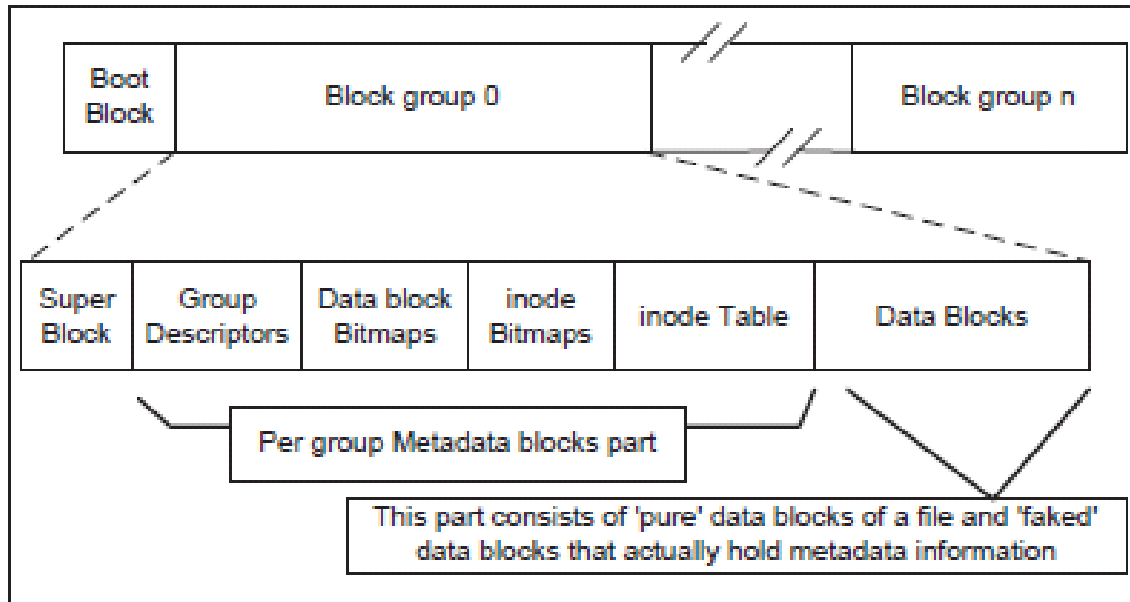


Offline builder: Read the FS (ext3) metadata from the disk

- To find an inode owning a block, one must traverse the entire inode table until an inode that contains the required block is found.
- To find the filename of an inode, and since the directory hierarchy is kept separately from the inode information, additional mapping has to be resolved.

Online Update: Update the block-to-file data structure by analyzing the incoming commands

- Problem is crucial when considering the online model:
 - Captured block level commands, while requiring as little memory as possible, and acting as a listener.
 - Each file-level command is composed of several 'small' iSCSI commands. In order to infer host-level command a few iSCSI commands have to be gathered.



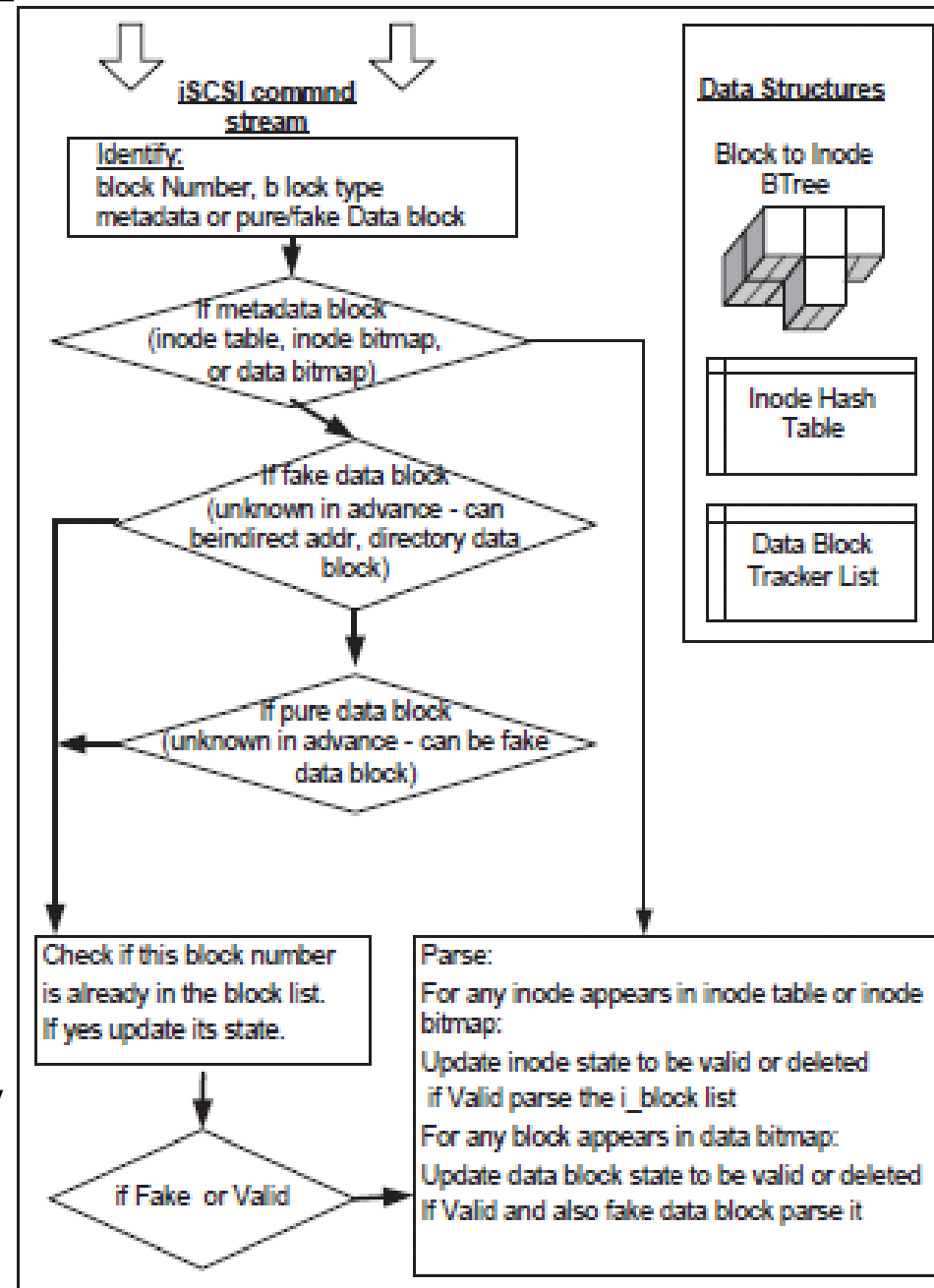
✓ Host-level command that creates a new inode

is translated into block-level commands:

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,
8. Writes the data blocks of this inode.

State Machine and algorithm assuming **no interleaving**

- IDStor considers an inode to be valid
 - Written in the inode table, $dtime==0$
 - inode bitmap was written, set to true
- ➔ “New File” or “New Directory” command
- An inode is considered *deleted*
 - Written in the inode table, with $dtime<>0$
 - inode bitmap was written, set to zero
- ➔ “Delete File” or “Delete Directory” command
- A data block is *valid*
 - A write command updating this data block.
 - A write command updating the address to this block- direct or indirect
 - The bit in data bitmap was set
- 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.
 - For our inverse map it suffices to keep only the block to inode pointer.



Flow and Data Structure Key points for inferring a single host-level command

- Capture the block level commands
 - sequential layout of device blocks (e.g., 512B) → File: name and logical blocks size (e.g., 4KB)
- Identify the type of the iSCSI command and the type of the block in the command,
- Maintain state machines per each inode and data block -- Insert the relevant information to the kept state machine and add the block-to-inode information to the inverse map.
- ✓ **Each iSCSI command can refer to many inodes and many blocks**
 - ✓ The old content could be stored at the listener, which would require an amount of memory equal to the size of the disk.
 - ✓ We are parsing the arriving metadata blocks to update the new state
- ✓ **inode's status and block's status can be learned** by collecting the information from the iSCS commands
 - ✓ For example inode is Valid only after the arrival of inode data, inode bitmap

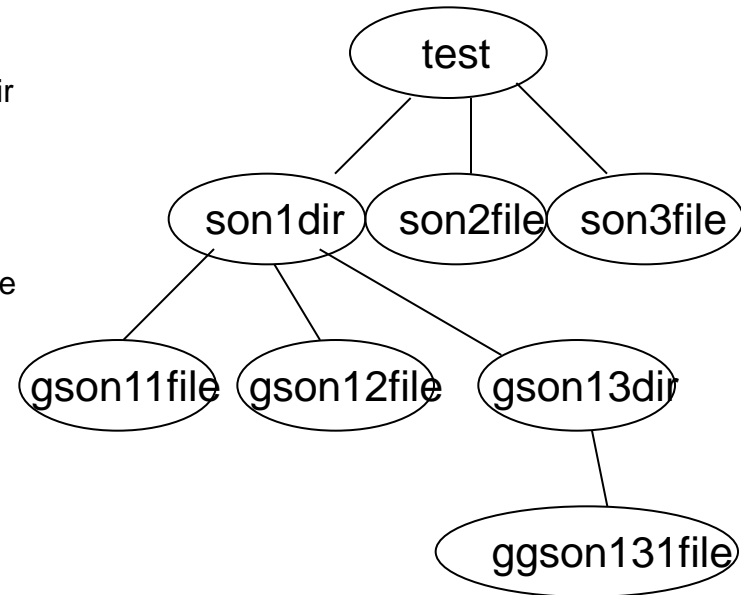
Prototype example - Perform on idstora



```
mount -t ext3 -o data=journal /dev/sda1 /mnt/tmp
```

```
#!/bin/bash
cd /mnt/tmp; mkdir test ; stat -c "test : %i" test
cd test ; echo "Creating files and directories, and writing some data..."
mkdir son1dir ; stat -c "test/son1dir : %i" son1dir
echo " new file " > son2file ; echo " file b " > son3file ;
echo "new file caaa under son1dir" > son1dir/gson11file
echo "new file caaa under son1dir" > son1dir/gson12file
mkdir son1dir/gson13dir ; stat -c "test/son1dir/gson13dir : %i" son1dir/gson13dir
stat -c "test/son2file : %i" son2file ; stat -c "test/son3file : %i" son3file
stat -c "test/son1dir/gson11file : %i" son1dir/gson11file
stat -c "test/son1dir/gson12file : %i" son1dir/gson12file
echo "new file caaa under gson13dir" > son1dir/gson13dir/ggson131file
stat -c "test/son1dir/gson13dir/ggson131file : %i" son1dir/gson13dir/ggson131file
sync
echo "Done creating directories and files. Press enter to continue..."
read
echo "Appending some more data..."
dd if=/dev/urandom of=son3file bs=4096 count=3 oflag=append conv=notrunc
dd if=/dev/urandom of=son1dir/gson11file bs=4096 count=2 oflag=append
conv=notrunc
sync
echo "Done appending data. Press enter to continue..."
read
echo "Truncating some data..."
truncate son3file 4500
sync
echo "Done truncating. Press enter to continue..."
read
cd ..
#sync
rm -rf test
sync
```

Script that create the following tree:



Screen snapshot of idstora, idstorb and idstor17

root@idstorA:/home/miriama/idstor/te

```
Main Options  VT Options  VT Fonts
[root@idstorA tests]# :q
-bash: :q: command not found
[root@idstorA tests]# !mou:p
mount -t ext3 -o data=journal /dev/sda1 /mnt/tmp
[root@idstorA tests]#
[root@idstorA tests]#
[root@idstorA tests]# !./test2
./test2-waiting
test : 310689
Creating files and directories, and writing
test/ccc : 310690
test/ccc/cccson : 310695
test/aaa : 310691
test/bbb : 310692
test/ccc/aaainccc : 310693
test/ccc/bbbinccc : 310694
test/ccc/cccson/aaason : 310696
Done creating directories and files. Press enter to continue...

Appending some more data...
3+0 records in
3+0 records out
12288 bytes (12 kB) copied, 0.00431031 seconds
2+0 records in
2+0 records out
8192 bytes (8.2 kB) copied, 0.00263198 seconds
Done appending data. Press enter to continue...

Truncating some data...
Done truncating. Press enter to continue...

[root@idstorA tests]#
[root@idstorA tests]#
[root@idstorA tests]#
[root@idstorA tests]#
[root@idstorA tests]#
[root@idstorA tests]#
[root@idstorA tests]#
[root@idstorA tests]#
[root@idstorA tests]#
[root@idstorA tests]#
```

Main Options VT Options VT Fonts

```
[root@idstorB ~]# cd ~/miriama/idstor/iSCSI_target/isd_target/src/
[root@idstorB src]# !./ip
./isd_target -ip 9.148.42.121 -dev /dev/sg1
[root@idstorB src]# ./isd_target -ip 9.148.42.121 -dev /dev/sg1
Target Name = idstorb
starting iSCSI target, target name = iqn.2004-11.com.ibm:0:idstorb
binding to (IP=9.148.42.121 port=3260)
Exporting sg device /dev/sg1

initiator 9.148.41.228 logged in - iSCSI normal session started
```

root@idstor17:/home/miriama/idstor/online

Main Options VT Options VT Fonts

```
14:57:51 INFER: block 638976 valid, belongs to /test/ccc/cccson (inode 310695)
14:57:51 INFER: new file /test/ccc/cccson/aaason (inode 310696)
14:57:51 INFER: block 628736 valid, belongs to /test/aaa (inode 310691)
14:57:51 INFER: block 628737 valid, belongs to /test/bbb (inode 310692)
14:57:51 INFER: block 628738 valid, belongs to /test/ccc/aaainccc (inode 310693)
14:57:51 INFER: block 628739 valid, belongs to /test/ccc/bbbinccc (inode 310694)
14:57:51 INFER: block 628740 valid, belongs to /test/ccc/cccson/aaason (inode 310696)
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
15:01:07 INFER: block 628741 valid, belongs to /test/bbb (inode 310692)
15:01:07 INFER: block 628742 valid, belongs to /test/bbb (inode 310692)
15:01:07 INFER: block 628743 valid, belongs to /test/bbb (inode 310692)
15:01:07 INFER: block 628744 valid, belongs to /test/ccc/aaainccc (inode 310693)
15:01:07 INFER: block 628745 valid, belongs to /test/ccc/aaainccc (inode 310693)
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
15:01:42 INFER: block 628742 no longer in use
15:01:42 INFER: block 628743 no longer in use
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
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/aaa (inode 310691) deleted
15:02:00 INFER: file /test/bbb (inode 310692) deleted
15:02:00 INFER: file /test/ccc/aaainccc (inode 310693) deleted
15:02:00 INFER: file /test/ccc/bbbinccc (inode 310694) deleted
```

```
Shell Script @ idstorA
#! /bin/bash ; cd /mnt/tmp
mkdir test ; cd test
    mkdir son1dir →→

echo " new file " > son2file →→→
echo " file b " > son3file

echo "new file caaa under son1dir" >
    son1dir/gson11file →→
echo "new file caaa under son1dir" >
    son1dir/gson12file → →
mkdir son1dir/gson13dir
    →→
echo "new file caaa under gson13dir"
>
    son1dir/gson13dir/ggson131file
    → →

sync

echo "Done creating directories and
files.
Press enter to continue..."
read
```

```
IDStor infers Host-level command @ listener
14:57:51 INFER: new dir /test (inode 310689)

14:57:51 INFER: block 630784 valid, belongs to /test (inode
310689)
14:57:51 INFER: new dir /test/son1dir (inode 310690)

14:57:51 INFER: new file /test/son2file (inode 310691)
14:57:51 INFER: new file /test/son3file (inode 310692)

14:57:51 INFER: block 634880 valid, belongs to /test/son1dir
(inode 310690)
14:57:51 INFER: new file /test/son1dir/gson11file (inode 310693)
14:57:51 INFER: new file /test/son1dir/gson12file (inode 310694)
14:57:51 INFER: new dir /test/son1dir/gson13dir (inode 310695)
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)
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)
```

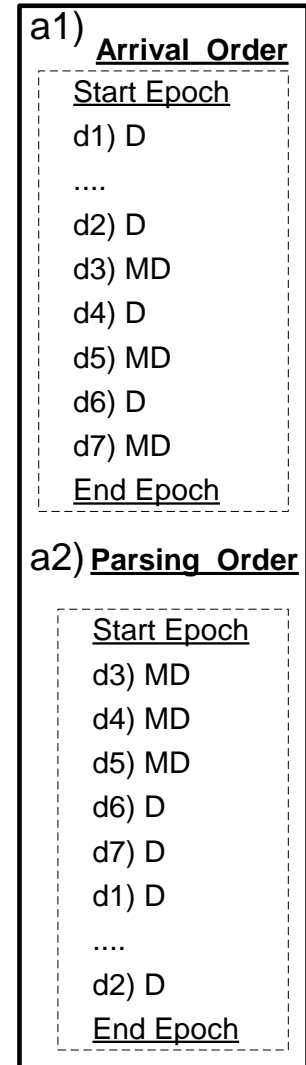
Shell Script @ idstorA

```
echo "Appending more data..."
dd if=/dev/urandom of=son3file
  bs=4096 count=3
  oflag=append
  conv=notrunc
dd if=/dev/urandom
  of=son1dir/gson11file
  bs=4096 count=2
  oflag=append
  conv=notrunc
truncate son3file 4500
  → → →
Sync ; cd ..
rm -rf test → → →
sync
```

IDStor infers Host-level command @ listener

```
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)
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)
15:01:42 INFER: block 628742 no longer in use
15:01:42 INFER: block 628743 no longer in use
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
```

- Information can be delayed in cache at the host and flushed to the disk or storage at any order, and usually after some delay. →
 - an ambiguous inode identification by our online parser.
- For example,
 - ✓ Assume block 1000 is currently assigned to inode x. Inode a is truncated, thus block 1000 is freed but was not flushed to disk yet.
 - ✓ Now a large amount of data is added to inode y, 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,
 - ✓ b) the inode of y has to be written to update that 1000 is an indirect block belonging to it.
 - ✓ If b) happens before a), It is OK!!
 - ✓ if a) happens before b), our inverse map still holds block 1000 as a valid pure data block belonging to inode x, 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).
- Parsing order is different than arrival order!! We must parse the data block after parsing the relevant metadata blocks



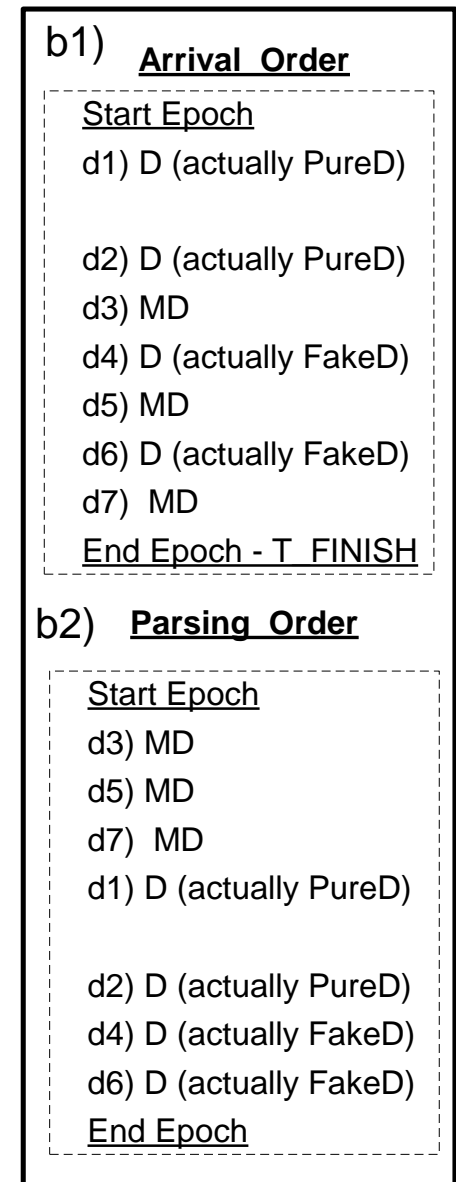
- How is it different for ext3?
 - How to identify the epoch? By using ext3 Journaling - The Journal and Ordered modes can guarantee the start of the transaction epoch by capturing the T_FINISH command
 - How to distinguish between Pure Data and Fake data?
- Algorithm:
 - Parse the Metadata and delay the data (pure and faked)

Use each parsed metadata for the state machine calculation

Maintain a list of the unhandled data blocks that are referred
by the metadata

Maintain a list of the delayed data blocks

- Go over the unhandled data block list and fetch them from the delayed list until



- Naïve solution: store amount of memory equal to the size of the disk.
- *Inodes hash table* holds all the inodes that exist in the system, valid and semi valid.
 - For each inode: inode structure that reflects the inode state,
 - The keys are the inode numbers and the values are pointing to the appropriate inode structures.
- *Block-to-inode ranged 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.
 - Contains only valid blocks, pointing to valid inodes.
- *Data block tracker 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.
 - Kept temporarily - state and its content - until its ownership and type are verified.
 - Content can be deleted once it is parsed.
- Our data structure is mostly independent of the file system.
- The data structures described above are first initialized and then updated online.

Memory Consumption – for the block-to-inode mapping

- 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).
 - Our inode structure holds only those fields in the file system data structure that are necessary for the block-to-inode inference.
- No need to keep data blocks, both pure and fake ones,
 - Enough to maintain a list of block numbers, each with a pointer to the owning inode → we keep at most 8 bytes per each data block.
 - For example, for a logical block size of 4KB, 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.
- Overestimation due to the range-aware BTree data structure since the keys represent a series of consecutive blocks rather than a single block number.
- Additional memory is kept for the data block tracker list.
 - Each block is kept for a short interval as the time it takes for the journal mechanism to be flushed to disk.
 - 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 $4000 * 5 * 2500 = 50\text{Mbytes}$ for temporary data.



inode-to-filename mapping

Do we need to answer the question: “What was the exact file name that is related to a certain inferred host-level command, given an inode?”.

- When using hard links – several filenames per inode
- Requires a lot of memory since
 - should holds the directory hierarchy information with all the filenames.
 - Otherwise It is hard to infer the addition or the deletion of another file name to an inode

List of Rules – pairs of the form *<identifier; rule>*,

- For existing files, their associated inodes can be fetched easily from the file system.
- For files that do not yet exist, → the rule requires the filenames and full paths of inodes identification as they are created or renamed, even if the rules are identified using inode IDs.
- a directory is renamed: affects the full path of all files underneath it. → keep the whole directory hierarchy.

Current Research

Online infer of file-level information by listening to block protocol

Future Works

- Extend the security layer work
- Storage-based Intrusion Prevention
- Storage-based IDS In the Databases world
- Use the file system view algorithm for other application as:
 - Data replication
 - Backup
- Integration with other devices that use block interface – as the hypervisor

Thank You!

