Questions and Answers
Topics that reviewers found unclear

Arkadiusz Bulski

asked by Thistle (pseudonym)
recorded on 2015-04-10

What control will a user have over wiping of files? Wiping of files is not practical? You mean manual wiping?

There are two different actions, regular deletion and permanent deletion. One is reversible and other is permanent. Please refer to answer to next question.

Practicality is meant here as both performance and assurance. Wiping in general is a synonym to permanent removal but in this context it means a specific implementation, through overwriting of file content. Historically, this is how it was done and is still being done. There are two major deficiencies with this approach. First problem, wiping performance is limited by hard disk performance and file sizes and fragmentation. Hard disks operate usually at maximum of about 120 MB/s. Overriding cannot proceed faster than that. And to be really sure, you should do that 35 times (Gutmann method). Second problem, this method is not safe. Data blocks that are no longer pointed to by inode, due to truncate operation for example, cannot be overridden. In general it is not possible to find out which blocks used to belong to a selected file, much less allocate these blocks back specifically in order to allow overwrite. Filesystems do not keep history of which blocks used to belong to a file. Once a block is deallocated, a track is gone. Also, copy on write mechanism can be applied to data writes, and if so, overwriting has no effect anyway. There is no common interface to find out whether underlying filesystem uses copy on write or not, nor to disable copy on write for a specific file before overwriting.

Will there be an option for non-permanent wiping in case a user accidentally wipes a file? What safety measures will be there to prevent accidental complete wiping of content? Or is that a risk users have to take?

There are two distinct actions. Regular deletion (called deletion) is an action that can be taken by non-privileged user. User processes are able to take this action without any user interaction. Therefore this kind of action is inspectable and undoable. Browsing history reveals any deletion and allows for recovery. Permanent deletion (called wiping) is a different action that requires root privileges. User processes are normally not able to take this action. This kind of action is also inspectable, as revision history reveals that some wiping has taken place but file content is not accessible anymore.

History of changes to files means that all versions of the file remain stored, or at least a history that can be viewed. Isn't that impractical in some cases of confidential files? An option to not save a file history?

History of changes is a list of snapshots, states saved at some points in time, ordered chronologically. History can be browsed and every snapshot can be inspected for details but also every snapshot can be restored back. Confidentiality is not compromised because browsing past versions of files requires same level of access as browsing current version. Anyone having access to past history would necessarily have had access to current file in the first place. When user deletes a file, it will be recoverable as long as it is browsable in history. When user permanently deletes a file (as opposed to regularly deleting), it will not be recoverable even through past versions. Versioning as a feature can also be disabled. If so, regular deletion would make a file immediately inaccessible.

Deleted files are permanently gone, how would that be done? In comparison to saving file history...

Technically there are two ways of ensuring data is permanently gone, either through overriding data itself or overriding encryption keys that were used to encrypt said data. Second approach is much better since keys occupy only few bytes and overriding few bytes takes almost no time. This also erases data blocks no longer assigned to a file, due to truncate operations. Versioning is an independent feature. Different versions are encrypted with same encryption keys, allowing a swift permanent removal of entire history of a given file.

Legal questions, may use of the program encounter difficulties with existing laws in some countries?

Depending on your country of residence, you may be:

- forbidden from using cryptographic products in general
- required to get license or send notification before importing
- required to disclose cryptographic keys to authorities in advance
- asked to enter password and hand over your laptop
- asked if you have any other undisclosed encrypted partitions
- subpoenaed to produce keys or decrypted data itself
- jailed for long time or until your produce encryption keys

Lesson to learn here is that established law can put you in a position where technical solutions do not give you an easy and legal way out. Consider crossing US border with child pornography on your laptop (there was a famous real case). Officer asks you to mount all partitions and then asks if you have any hidden partitions on your laptop. Note that lying to a federal agent is a criminal offense in US. If you mount the hidden partition, jail, if you lie it does not exist, jail if caught, if you deny to answer, perhaps denial of entry into the country. Plausible deniability feature may let you lie your way out of the situation but it will not make it any more legal.

asked by Steven Balderrama
recorded on 2015-08-14

On mission document, page 3, I found top 3 things what lacks today in a filesystem: versioning, secure deletion and disk utilisation.

There are also other features but some are so specialized that clearly there will not be many people that will have a use for them. This project aims to deliver a versatile filesystem that will be useful to everybody.

Selective compressioning is great, but some files should not be compressed for performance.

Some files are almost impossible to compress, audio video formats are a good example. However, performance does not necessarily have to be negatively impacted. Methods like Gzip are meant to achieve high utilisation with little regard for performance but there are also other, less efficient, methods that are many times faster than a hard disk. Snappy developed by Google can compress ~250 MB/s and decompress ~500 MB/s per core at it's slowest, with worst case input data, and much more with more compressible data, while a hard disk can sustain only about ~120 MB/s transfer. Fragmentation lowers disk side of inequality even further. Direct Memory Access (DMA) allows to transfer data to disk in parallel to code execution so compression and disk operations can be done in parallel. The minimum of two throughputs then becomes a bottleneck. Compression is therefore CPU bound, and not disk bound. Processor load should be fractional and not impact user experience. Also, adaptive compression can be used. If during sequential writing compression gain is close to none, then further writes can skip

compression in anticipation of no gain.

We sorta talked about performance. Since there is compression and encryption, how will that not hinder performance?

Compare throughput of modern hard disks to throughput of modern processors. Hard disks can sustain ~120 MB/s. Compression like Snappy can handle ~250 MB/s per core easily. Encryption using Salsa20 can handle ~400 MB/s per core. Lightweight hashing using SipHash can handle ~530 MB/s per core. All of the above can be achieved at a fraction of CPU load, and you can always disable some features to lower the CPU load. DMA allows to transfer data to disk in parallel to computation. In the end, throughput becomes the minimum of the two, which is equal to disk throughput. Refer to previous question.

And yes, versioning. How many times I have seen this, especially programmers who do not utilize software versioning. I have seen it even at my work now. Version control, is that what you mean?

There are two features that may be used to revert changes. Snapshots is a feature that saves state at explicitly chosen points in time, allowing to revert only to specific states that have been recognized in advance. This is the model of how version control software works. Commits are then the equivalent of creating snapshots. So this was the first feature. Continuous versioning is similar to snapshots but has an important difference. Every change is automatically creating a revision, like a temporary lightweight snapshot, that gets stacked on top of previous changes. As time progresses, revisions from too old past become automatically removed or compacted away. So in contrast, snapshots are manually created while versioning is automatic, and snapshots are kept forever while versions are only available for some period of time unless manually retained.

Lastly, you said about having it's own type of recovery. Explain in detail how it will recover or roll back.

Early filesystems like Fast File System and it's descendant Ext2 depended on running a program called *fsck* that scanned all inodes after an interruption. Later filesystems like Ext3 were using journaling to reapply a set of changed blocks. Later filesystems like Btrfs and ZFS started using copy on write and checksums. The state of the art solution would perhaps comprise of copy on write, checksums, and intents.

From user's perspective one thing should be noticeable: recovery should happen lazily. Mounting an interrupted filesystem should be achievable in sub-second time. Files that were not closed properly require maintenance and will be scrutinized either at opening or during routine scrubbing, whichever comes first. Maintenance should be carried out on individual files and be postponed preferably until after booting is complete. Further recovery can run in background. Unaffected files become accessible immediately.

This can be achieved by using copy on write, checksums, and intents. For example, when a write operation occurs the data is stored into a newly allocated extent and both its address and checksum are stored in an intent, which itself is stored at a known location. At mount, only intents need to be processed. Intents integrity are checked through their checksums and if they are good, extents they point to become reserved as if they were properly allocated. At this point filesystem is mounted and ready to go. When a file gets opened any remaining unverified intents referring to it must be processed. Intent holds a checksum to verify the data, if data is good then the intent is reapplied to inode, if not then data is deallocated and operation gets reverted.