

# Avfs: An On-Access Anti-Virus File System

Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, and Erez Zadok  
Stony Brook University

## 1 Introduction

Viruses, worms, Trojan horses, and other malicious programs have existed almost since people started sharing files and using network services. The growth of the Internet and users' demands for more active content has brought with it an explosion in the number of virus and worm attacks, costing untold hours of lost time.

Virus scanners consist of two components: a scanning engine and a component that feeds data to the scanning engine. The scanning engine searches for patterns that identify a virus. Data is fed to the engine by intercepting system calls or other methods. Currently, most Anti-virus products only scan files when they are opened, closed, or executed. This leaves a window of vulnerability between when the virus is written and detection occurs. If the virus has already overwritten valuable data, then the data can not always be recovered. On-access scanning is an improvement over `on-open`, `on-close`, and `on-exec` scanning. An on-access scanner looks for viruses when an application reads or writes data, and can prevent a virus from ever being written to disk.

We have developed a file system, *Avfs*, that is a true *on-access* virus scanning system. We have adapted the ClamAV [2] open source virus scanner to work with *Avfs*. Our improved scanning engine, which we call *Oyster*, runs in the kernel and scales significantly better than ClamAV. By running *Oyster* in the kernel we do not incur unnecessary data copies or context switches. The ClamAV scanner does not scale well for large virus databases. Our extensive modifications to ClamAV improved the scalability and scanning speed for large databases. *Oyster* allows the system administrator to decide what trade off should be made between memory usage and scanning speed. Whereas ClamAV's performance degrades linearly with the number of virus signatures, *Oyster* scales logarithmically. Since the number of viruses grows continuously, these scalability improvements will become even more important in the future.

We have evaluated the performance of *Avfs* and *Oyster*. *Avfs* has an overhead of 14.5% for normal user workloads. *Oyster* improves the performance of ClamAV by a factor of 4.5.

## 2 Scanner Design

ClamAV uses a variation of the Aho-Corasick pattern-matching algorithm [1], which is well suited for applications that need to match a large number of patterns against some input text. The algorithm consists of two

parts. In the first part, a pattern matching finite state machine is constructed. In the second part, the text string is used as the input to the automaton.

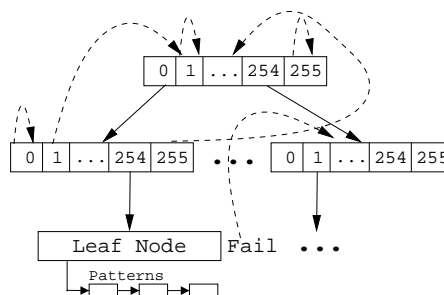


Figure 1: A fragment of the ClamAV trie structure. Solid lines show success transitions; dashed lines show failure transitions.

To quickly look up each character read from the input, ClamAV constructs a 255-way trie structure as shown in Figure 1. The memory usage of ClamAV depends on how deep the trie is. The deeper the trie, the more nodes are created (each node is 1,049 bytes). Since the Aho-Corasick algorithm builds an automaton with a depth equal to the longest pattern, the memory usage would be unacceptably large because some patterns are over 2KB. ClamAV modifies the Aho-Corasick algorithm so that the trie is constructed only to some maximum height, and all patterns beginning with the same prefix are stored in a linked list under the appropriate leaf node. The maximum trie height is restricted by the length of the shortest pattern, which is currently two bytes.

ClamAV's performance suffers whenever a node with a large number of patterns with the same prefix is encountered during input matching. We modified ClamAV's data structures and algorithms to support larger tries, and a maximum trie height restriction to control memory usage. The resulting data structure, which we call a *variable height trie*, enables *Oyster* to scale better for large databases, while maintaining reasonable memory usage.

## 3 File System Design

*Avfs* was designed with three goals in mind:

**Accuracy and high-security** We use a page-based virus scanner that scans on-access as opposed to conventional scanners that scan on `open` and `close`. *Avfs* has support for data-consistency using versioning and support for forensics by recording malicious activity.

**Performance** We use our improved scan engine, Oyster, and avoid repetitive scanning using state.

**Transparency** User intervention and application modifications are not required to support virus protection.

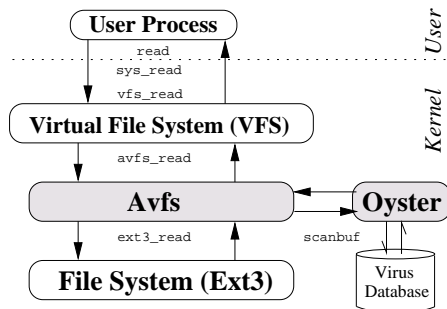


Figure 2: Avfs infrastructure

Figure 2 shows a high-level view of the Avfs infrastructure. When Avfs is mounted over an existing file system it forms a bridge between the *Virtual File System* (VFS) and the underlying file system. The VFS calls various Avfs operations and Avfs in turn calls the corresponding operations of the underlying file system. Avfs performs virus scanning and state updates during these operations. *Oyster* is a virus-scanning engine integrated into the Linux kernel. It exports an API that is used by Avfs for scanning files and buffers of data. For example, a `read` from the VFS, `sys_read`, translates into `vfs_read` in the Avfs layer. The lower layer read method (`ext3_read`) is called and the data received is scanned in the Avfs layer.

To reduce the amount of data scanned, Avfs stores persistent state. Avfs scans one page a time, but a virus may span multiple pages. After scanning one page, Avfs records state. When the next page is scanned, Avfs can resume scanning as if both pages were scanned together. After an entire file is scanned, Avfs marks the file *clean*. Avfs does not scan clean files until they are modified.

Avfs supports two forensic modes. The first mode prevents a virus from ever reaching the disk. As soon as a process attempts to write a virus, Avfs returns an error before the changes are made to the file. The second mode does not immediately return an error. Before the first write to a file is committed, a backup of that file is made. If a virus is detected, then Avfs quarantines the virus (no other process can access a file while it is quarantined), allows the write to go through, records information about the event, and finally reverts to the original file. This leaves the system in a consistent, clean state, and allows the administrator to investigate the event.

## 4 Evaluation

Figure 3 compares the performance of ClamAV with Oyster when scanning two 1GB files. The first file contained random bytes and the second file contained exe-

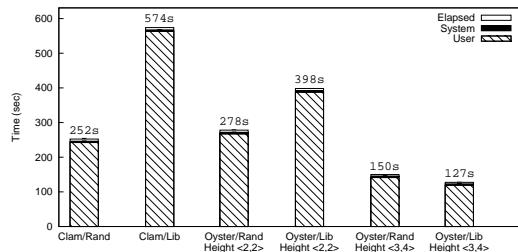


Figure 3: Scan times for Oyster and ClamAV.

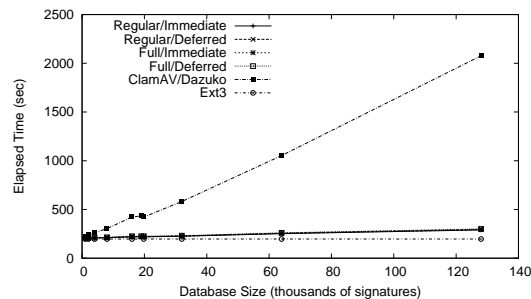


Figure 4: Am-Utils for all Avfs modes, ClamAV, and Ext3.

cutable content from `/usr/lib`. We ran Oyster once with the minimum and maximum heights set to two as in ClamAV. With the heights set to 3 and 4, respectively, Oyster is 4.5 times faster than ClamAV.

Figure 4 shows the performance results for an Am-Utils compile benchmark using various database sizes. This benchmark compares the performance of the Avfs and Oyster scan engines with different forensic modes, against Dazuko, an on open-close-exec scanner that uses ClamAV. Whereas ClamAV’s performance degrades linearly with the number of virus signatures, Oyster scales logarithmically.

## 5 Conclusion

The main contribution of our work is that for the first time, to the best of our knowledge, we have implemented a truly on-access state-oriented anti-virus solution that scans input files for viruses on reads and writes.

Avfs intercepts file access operations (including memory-mapped I/O) at the VFS level unlike other on-access systems that intercept the `open`, `close`, and `exec` system calls. Scanning during read and write operations can trap viruses before they are written to disk. Avfs also provides data consistency with backup files.

Our Oyster scan engine scales efficiently using variable trie heights. The state-based scanning in Oyster allows us to scan a buffer of data incrementally in parts.

## References

- [1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.
- [2] T. Kojm. ClamAV. [www.clamav.net](http://www.clamav.net), 2004.