Avfs: An On-Access Anti-Virus File System

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

Appears in the proceedings of the 13th USENIX Security Symposium (Security 2004)

Abstract

Viruses and other malicious programs are an ever-increasing threat to current computer systems. They can cause serious damage and consume countless hours of system administrators' time to combat. Most current virus scanners perform scanning only when a file is opened, closed, or executed. Such scanners are inefficient because they scan more data than is needed. Worse, scanning on close may detect a virus after it had already been written to stable storage, opening a window for the virus to spread before detection.

We developed *Avfs*, a true on-access anti-virus file system that incrementally scans files and prevents infected data from being committed to disk. Avfs is a stackable file system and therefore can add virus detection to any other file system: Ext3, NFS, etc. Avfs supports forensic modes that can prevent a virus from reaching the disk or automatically create versions of potentially infected files to allow safe recovery. Avfs can also quarantine infected files on disk and isolate them from user processes. Avfs is based on the open-source ClamAV scan engine, which we significantly enhanced for efficiency and scalability. Whereas ClamAV's performance degrades linearly with the number of signatures, our modified ClamAV scales logarithmically. Our Linux prototype demonstrates an overhead of less than 15% for normal user-like workloads.

1 Introduction

Viruses, worms, and other malicious programs have existed since people started sharing files and using network services [3, 15]. The growth of the Internet in recent years and users' demand for more active content has brought with it an explosion in the number of virus and worm attacks, costing untold hours of lost time. Organizations report more financial losses from viruses than from any other type of attack—reaching well into the millions [16]. Once infected, original file content may not be recoverable. Viruses can transmit confidential data on the network (e.g., passwords) allowing an attacker to gain access to the infected machine. System administrators must clean or reinstall systems that are not adequately protected. A virus's propagation wastes valuable resources such as network bandwidth, disk space, and CPU cycles. Even if a site is not infected with a virus, its servers can be overloaded with probes.

The most common countermeasure to malicious software is a virus scanner. Virus scanners consist of two parts: a scanning engine and a component that feeds the data to the scanning engine. The scanning engine searches for virus *signatures*, or small patterns that uniquely identify a virus. Virus signatures should ideally be kept short so that scanning is more efficient, but at the same time they should be long enough to ensure that there are very few, if any, false positives.

A virus scanner can either scan interactively or transparently. An interactive scanner allows a user to request a scan of a specific file or directory. Since this process is cumbersome, most virus scanners also transparently scan files by intercepting system calls or using other operating-system–specific interception methods. Currently, most transparent scanners only scan files when they are opened, closed, or executed.

Consider the case where a Linux file server exports NFS or CIFS partitions to other machines on the network. Suppose the file server has a virus scanner that scans files when they are closed. Client A could create a file on the server and then write the virus. Suppose that before A closes the file, client B opens this file for reading. In contrast to Windows, Linux does not implement mandatory file locking. There is nothing that prevents B from reading and executing the virus. Even if the file server scans files both when they are opened and closed, B could still execute the virus before it is detected as follows: (1) A writes part of the virus, (2) B opens the file at which point the file is scanned, but no virus is found, (3) A completes writing the virus, (4) B reads the rest of the virus before A closes the file. Virus scanners that scan files when discrete events occur, such as `open`or `close`, leave a window of vulnerability between the time that the virus is written and the time when detection occurs. Additionally, because the entire file must be scanned at once, performance can suffer.

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. Since scanning is performed only when data is read, as opposed to when the file is opened, users are not faced with unexpected delays. We have developed a stackable file system, *Avfs*, that is a true *on-access* virus scanning system. Since Avfs is a stackable file system, it can work with any other unmodified file system (such as Ext2 or NFS), and it requires no operating system changes. For example, an Avfs mounted over SMB can protect Windows clients transparently. In addition to virus detection, Avfs has applications to general pattern matching. For example, an organization might want to track or prevent employees copying files containing the string "Confidential! Do not distribute!".

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. When a process attempts to write a virus, Avfs returns an error to the process without changing the file. The second mode does not immediately return an error to the process. 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 state and allows the administrator to investigate the event.

We have adapted the ClamAV open source virus scanner to work with Avfs. ClamAV includes a virus database that currently contains nearly 20,000 signatures. 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. Whereas ClamAV's performance degrades linearly with the number of virus signatures, Oyster scales logarithmically. Oyster also allows the system administrator to decide what trade-off should be made between memory usage and scanning speed. Since the number of viruses is continuously growing, 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 on the performance of ClamAV by a factor of 4.5.

The rest of the paper is organized as follows. Section 2 outlines the design of our system. Section 3 details the design of our scanner. Section 4 details the design of our file system. Section 5 discusses related work. Section 6 presents an evaluation of our system. We conclude in Section 7 and discuss future directions.

2 Design overview

We begin with an overview of Avfs's components and our four main design goals:

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

Performance: We enhanced the scanning algorithm and avoided repetitive scanning using a state-oriented approach. Our scan engine runs inside the kernel, which improves performance by avoiding message passing and data copying between the kernel and user space.

Flexibility and portability: We designed a flexible system in which the scanning module is separate from the file system module. A stackable file system allows for portability to different environments and works with any other file system.

Transparent: Our system is transparent in that no user intervention is required and existing applications need not be modified to support virus protection.

Stackable file systems are a technique to layer new functionality on existing file systems [19]. A stackable file system is called by the *Virtual File System* (VFS) like other file systems, but in turn calls a lower-level file system instead of performing operations on a backing store such as a disk or an NFS server. Before calling the lower-level file system, stackable file systems can modify an operation or its arguments. The underlying file system could be any file system: Ext2/3, NFS, or even another stackable file system.

Avfs is a stackable file system that provides protection against viruses. Figure 1 shows a high-level view of the Avfs infrastructure. When Avfs is mounted over an existing file system it forms a bridge between the 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 our virus-scanning engine that we 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 *Virtual File System* (*VFS*), `vfs_read()`, translates into `avfs_read()` in the Avfs layer. The lower layer read method (`ext3_read()`) is called and the data received is scanned by Oyster.

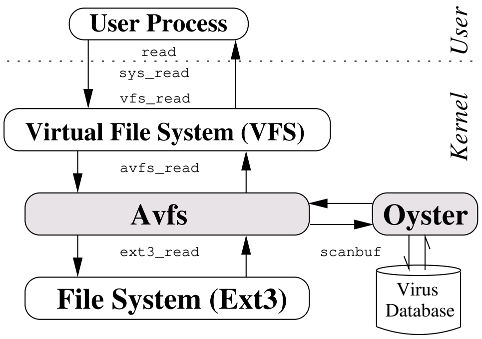


Figure 1: Avfs infrastructure

The relevant file system methods that the stacking infrastructure provides to us are `read`, `write`, `open` and `close`. A page is the fundamental data unit in our file system. Reads and writes occur in pages, and we perform virus scanning during individual page reads and writes. This level of granularity has three key advantages over scanning on `open` and `close`. First, we scan for viruses at the earliest possible time: before the data from a `read` is delivered to the user and before the data from a `write` propagates to the disk. This reduces the window of opportunity for any virus attack significantly. Second, we have an opportunity to maintain the consistency of file data because we scan data for viruses before data gets written to disk. Third, with our state implementation we can scan files partially and incrementally. The state implementation also allows us to mark completely scanned files as *clean* so that would not need to be re-scanned if they are not modified.

In Section 3 we describe Oyster in detail and Section 4 we detail the design of Avfs.

3 Kernel-Based Virus Scanner

In Section 3.1 we describe the internals of ClamAV. In Section 3.2 we describe the enhancements we made to the ClamAV virus scanner.

3.1 ClamAV Overview

We decided to use the freely available *Clam AntiVirus* (ClamAV) [11] scanner as the foundation for our kernel-based virus scanner. ClamAV consists of a core scanner library as well as various command line programs. We modified the core ClamAV scanner library to run inside the kernel, and call this scanner *Oyster*.

ClamAV Virus Database As of December 2003, ClamAV's database had 19,807 viruses. Although this number is smaller than those of major commercial virus scanners, which detect anywhere from 65,000 to 120,000 viruses, the number of viruses recognized by ClamAV has been steadily growing. In the last six months of 2003, over 12,000 new virus signatures were added to the database.

The ClamAV virus definition database contains two types of virus patterns: (1) basic patterns that are a simple sequence of characters that identify a virus, and (2) multi-part patterns that consist of more than one basic sub-pattern. To match a virus, all sub-patterns of a multi-part pattern must match in order. ClamAV virus patterns can also contain wildcard characters. The combination of multi-part patterns and wildcard characters allows ClamAV to detect polymorphic viruses. Polymorphic viruses are more difficult to detect than non-polymorphic viruses, because each instance of a virus has a different footprint from other instances.

Basic patterns tend to be longer than multi-part patterns. Multi-part patterns have multiple pattern to identify a complete virus. The pattern lengths in the database vary from two bytes (for sub-parts of a multi-part pattern) to over 2KB long.

ClamAV Virus Detection Algorithm ClamAV uses a variation of the Aho-Corasick pattern-matching algorithm [1], which is well suited for applications that match a large number of patterns against input text. The algorithm operates in two steps: (1) a pattern matching finite state machine is constructed, and (2) the text string is used as the input to the automaton.

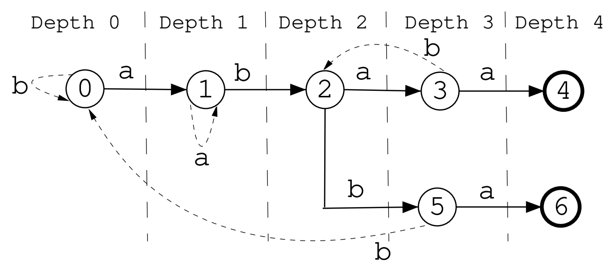


Figure 2: An automaton for keywords "abaa" and "abba" over the alphabet {a,b}. Success transitions are shown with solid lines. Final states are shown with bold circles. Failure transition are shown with dotted lines.

To construct a pattern matching automaton, the Aho-Corasick algorithm first builds a finite state machine for all of the patterns. Figure 2 shows the automaton for the keywords "abaa" and "abba" over the alphabet $\{a, b\}$. State 0 denotes the starting state of the automaton, and the final states are shown with bold circles. First, the pattern "abaa" is added, creating states 0–4. Thereafter, the pattern "abba" is added, creating states 5–6. Only two additional states were required since both patterns share the same prefix "ab." Transitions over the characters of the patterns are called *success* transitions.

Each state in the pattern-matching automaton must have transitions for all letters in the alphabet. If a success transition over a letter does not exist for some state, then a *failure* transition is created. To set up a failure transition, all states are processed in depth order; i.e., we process states of depth n before states of depth $n+1$. A state's depth s is defined as the length of the shortest path from the start state 0 to s. Any failure transition for start state 0 points back to state 0. Suppose that after matching some prefix $P = [1..k]$ of length k the automaton is in state s. Also, suppose that there is no success transition for some character c starting from state s. A failure transition for the character c is determined by following transitions for prefix $P[2..k]c$ starting from state 0.

Failure transitions are set up as follows. First, a missing transition for "b" from state 0 is set up to point back to state 0. State 1 does not have a transition for "a," (there is no pattern that begins with "aa"). To determine this failure transition, the first character is removed from the prefix, and transitions for the remaining characters are followed starting from state 0. So, the failure transition for "a" in state 1 points back to state 1. Similarly, state 3 does not have a transition for "b" (there is no pattern that begins with "abab"). To compute the failure transition for "b" in state 3, transitions for "bab" are followed from state 0. This failure transition points to state 2. Other failure transitions are set up similarly.

To determine if a text string contains any of the patterns, it is applied as the input to the automaton. The automaton follows transitions for each character from the input string until either the end of the string is reached, or the automaton visits one of the final states. To determine which pattern matches when the automaton visits the final state s, we simply follow the shortest path from the start state 0 to s. This automaton is a *trie* data structure. Trie data structures are used for fast pattern matching in input text. In the trie, each node and all of its children have the same prefix. This prefix can be retrieved by traversing the trie from the root node.

To quickly look up each character read from the input, ClamAV constructs a trie structure with a 256-element lookup array for each of the ASCII characters. 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 (1KB for the lookup array plus auxiliary data). Since the Aho-Corasick algorithm builds an automaton that goes as deep as the pattern length, the memory usage of ClamAV's structure would be unacceptably large because some patterns in the database are as long as 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 trie leaf node. ClamAV has the further restriction that all pattern lists must be stored at the same trie level. This restriction significantly simplifies trie construction and pattern matching, but due to this restriction, the shortest pattern length dictates the maximum trie height. Since the shortest pattern is only two bytes long, ClamAV can only build a trie with two levels. Figure 3 shows a fragment of a trie built by the ClamAV algorithm.

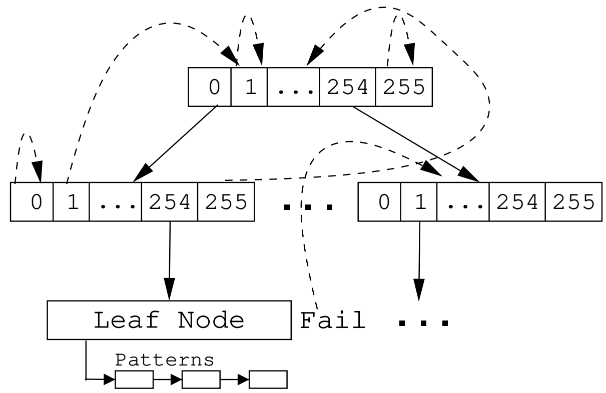


Figure 3: A fragment of the ClamAV trie structure. Success transitions are solid lines. Failure transitions are dashed lines.

ClamAV takes the following steps to construct a trie:

1. Read the next pattern from the virus database.
2. Traverse the trie to find an appropriate node to add the pattern to, creating new levels as needed until the maximum trie height is reached (this step sets up success transitions).
3. Add the pattern to the linked list inside a leaf node.
4. Process all nodes of the trie by depth (*level-order* traversal), and set up all failure transitions.

After the trie is constructed, ClamAV is ready to check whether an input matches any of the patterns in the trie. For each character read, ClamAV follows the trie transition and if a leaf node is encountered, all patterns inside the linked list are checked using sequential string comparisons. This process continues until the last input character is read, or a match is found.

3.2 Oyster Design

Our kernel-based virus scanner module is called by the file system to perform scanning every time files are read for the first time, created, or modified. Since each file contains one or more pages, and there are many files being accessed simultaneously, two of the major requirements for Oyster were speed and efficiency. In addition, since the number of viruses constantly grows, the virus scanner must be scalable. Unfortunately, ClamAV did not prove to be scalable. Its performance gets linearly worse as the number of patterns increase (see Section 6 for a detailed performance comparison). In Section 3.2.1 we explain the scalability problems with ClamAV. In Sections 3.2.2 through 3.2.4 we describe changes we made to the ClamAV data structures and algorithms. In Section 3.2.5 we describe the Oyster API for other kernel modules.

3.2.1 Virus Database and Scalability

The primary issue that limits ClamAV's scalability is the restriction that all pattern lists must be stored at the same trie level. This restriction forces the maximum trie height to be two. With the maximum level of two, and with each node holding 256 transitions, it would appear that this data structure should be scalable for up to $256^2 = 65536$ patterns, but this approximation is correct only if virus signatures consist of uniformly distributed random characters. However, virus signatures are neither random nor uniformly distributed.

Figure 4 shows the distribution of one-character prefixes in the ClamAV's database. Just 25 out of 256 one-character prefixes account for almost 50% of all prefixes. The distribution of two character prefixes is not random either. There are 6,973 unique two-character prefixes. 10% of those prefixes account for 57% of all patterns.

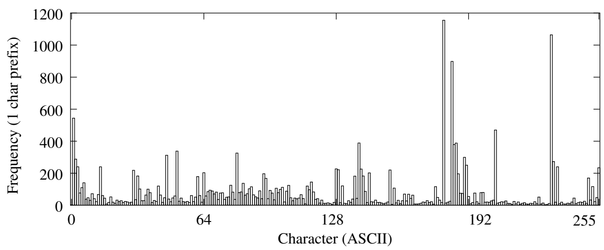


Figure 4: A histogram showing the one-character prefix distribution of ClamAV's database with 19,807 viruses (256 unique prefixes).

This high clustering of patterns means that there are some leaf nodes in the trie that contain linked lists with a large number of patterns. Since all of these patterns are scanned sequentially, performance suffers whenever such a node is traversed during file scanning. To have acceptable performance levels with large databases, the ClamAV data structures and algorithms had to be modified to minimize the number of times that patterns in leaf nodes were scanned and to minimize the number of patterns stored in each list.

Our modifications to the ClamAV data structures and algorithms are designed to meet the following three goals: (1) improve scalability and performance, (2) minimize memory usage and support a maximum trie height restriction so that an upper bound on memory usage can be set, and (3) allow the administrator to configure the system to trade-off memory vs. speed.

3.2.2 Variable Height Trie

To improve performance and scalability for large databases, we redesigned the ClamAV data structures to support trie heights greater than two. With each additional level, we add an additional 256-way fan-out for the trie, thus reducing the probability that leaf nodes will be scanned, which in turn improves performance. Patterns that are shorter than the maximum trie height or contain a wildcard character at a position less than the maximum trie height must be added to the linked lists in the intermediate trie nodes. Such nodes can contain both transitions to the lower level as well as patterns. We will use "?" to denote a single wildcard character. Figure 5 shows a trie with a height of four (plus leaf nodes). The trie contains patterns beginning with ASCII characters $\langle 254, 0, 0, 79 \rangle$ (node 8). It also contains patterns that begin with $\langle 0, 0, ? \rangle$ (node 3), as well as patterns beginning with $\langle 0, 0, 123, 255 \rangle$ (node 7).

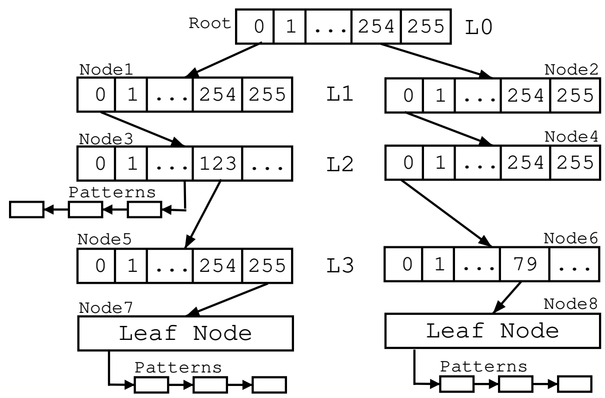
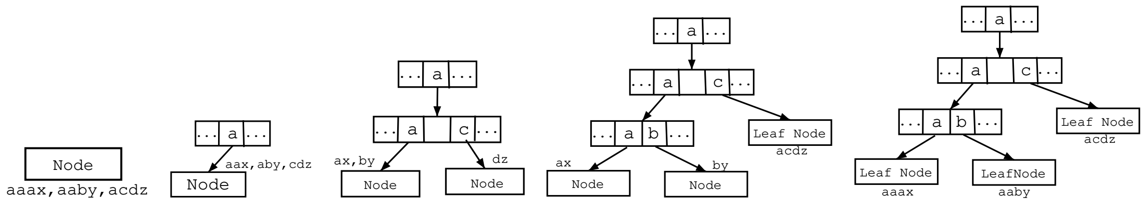


Figure 5: A trie with four levels (only success transitions are shown). Patterns beginning with characters $\langle 0, 0, ? \rangle$ are stored inside node 3, which contains both patterns and transitions.

The trie depicted in Figure 5 has two problems. The first problem is memory usage. If a pattern can be uniquely identified by a two-character prefix, then there is no need to store it at the maximum trie height level since a lot of memory would be used due to the large node size (each node is over 1KB). Our solution stores the pattern at the lowest possible level as soon as a unique prefix for this pattern is found.

The second problem is more involved. Suppose we have two patterns $\langle 0, 0, ?, 1 \rangle$ and $\langle 254, 0, 0, 79, 10 \rangle$. The first pattern is stored inside node 3 in Figure 5. This pattern cannot be stored at a higher level because a transition over the wildcard is not unique. The second pattern is stored inside node 8. Now suppose that we have an input string $\langle 254, 0, 0, 79, 1 \rangle$. The automaton will start transitioning through the right hand side of the trie: root node, node 2, node 4, node 6, and finally node 8. At this point, the last input character "1" will be matched against the last character of the pattern $\langle 254, 0, 0, 79, 10 \rangle$, and the match will fail. However, while traversing the right hand side of the trie, the characters $\langle 0, 0, 79, 1 \rangle$ match the pattern stored inside node 3, but we never visited this node to detect the match. More formally, if we have two patterns with unique prefixes $P_1[1..m]$ and $P_2[1..n], m > n$, and



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4 (e) Step 5

Figure 6: Operation of the `addpatterns` *function.*

$P_1[j..k] = P_2[1..n]$, where $j \geq 2$ and $k \leq m$, then the patterns with prefix P_2 must be scanned as soon as character k is read. We call this situation a *collision*.

3.2.3 Improving Memory Usage

To store patterns at the lowest possible level, we modified the ClamAV trie construction algorithm. Instead of storing the patterns in the trie as soon as they are read from the database, we store them in an array. We sort this array lexicographically by comparing pattern prefixes of length equal to the maximum trie height. After the patterns are sorted, each pattern is assigned an ID that is equal to the pattern's offset in the sorted pattern array. This sorting enables Oyster to conveniently identify all patterns with some unique prefix P by specifying start and end offsets into the sorted array.

We then proceed with the trie construction by calling our `addpatterns(`*`node`*`, `*`start_offset`*`, `*`end_offset`*`)` function, where *node* is the current node in the trie, *start_offset* and *end_offset* are offsets into the sorted pattern array that identify the range of patterns to add to the node. To begin trie construction, we call `addpatterns`, passing it the root node and the entire range of patterns as arguments. The `addpatterns` function operates as follows:

1. If the maximum trie height is reached, add all patterns in the range to the current node and return.
2. If the range contains only one pattern, add this pattern to the current node, and return.
3. Add to the current node all patterns of length equal to the current height and all patterns that have a "?" character at the current height. If there are no more patterns left, return.
4. Otherwise, the range still contains patterns. For each character $0 \leq i \leq 255$, find the range of patterns that have character i in the position equal to the current height, create transitions for i inside the current node, and recursively call the function with the new range and new node. The maximum recursion depth is equal to the maximum trie height. The kernel has a limited stack size, but because our recursive function is bounded by a small maximum trie height, there is no danger of stack overflow.

The Figure 6 shows the trie construction process for the patterns {aaax, aaby, acdz}. In step 1, `addpatterns` is called with a node and the three patterns in the range. Since there is more than one pattern in the range, `addpatterns` creates a transition for the character a, and recursively calls itself with the same range, but using the node at the next level down. In step 2, the next characters from the patterns are compared. Two transitions for characters a and c are set up and the function calls itself recursively twice, once with the range containing the patterns {aaax, aaby}, and once with {acdz} (step 3). In step 4, the pattern "acdz" is added to the current node since the range contains only one pattern, and the remaining patterns get added to the next level in step 5. Notice that the pattern "acdz" was added as soon as the unique prefix "ac" was found for this pattern (step 4).

Since the pattern array was presorted, whenever patterns (delimited by *start_offset* and *end_offset*) get added to a node, they begin with the same prefix, and therefore have sequential pattern IDs. This reduces memory usage. Instead of creating a linked list of patterns, we simply add a pattern-range structure to the node. The pattern-range structure has three members: (1) start offset, (2) end offset, and (3) the level of the trie where the range is stored. The level member of this structure determines how many characters from all of the patterns are already matched by the trie prefix.

The trie construction algorithm described above minimizes memory usage by storing each pattern at the lowest possible level. The algorithm maintains the maximum trie height restriction to enforce an upper bound on memory usage. In addition, we provide a recommended minimum height configuration parameter to allow a trade-off between speed and memory usage. Even if a pattern can be uniquely identified by a single character prefix, it is not added to the trie until the recommended minimum height is reached. Short patterns or patterns with wildcard characters are still stored at levels below the recommended minimum trie height. Increasing the recommended minimum height parameter increases memory usage. This increase, however, could improve performance because leaf nodes of the trie would be scanned less frequently due to the larger trie height (see Section 6). Note that the minimum trie height parameter should not be set too high. In our tests, a minimum height of three proved to be scalable with databases of up to 128K virus definitions. A combination of minimum and maximum heights allows for flexibility in tuning performance and memory usage.

3.2.4 Collision Detection and Avoidance

Collisions are detected using a simple procedure. We start processing every node in the trie in a level-order traversal; i.e., process all nodes on level n before processing nodes on level $n+1$. For every success transition in a node A, we traverse the trie as if it were a failure transition. We look at a node, say node B, pointed to by the failure transition. If node B has pattern ranges stored under it, then there is a collision. Whenever a collision is detected, all pattern ranges from node B are copied to node A. The level member of the pattern range structure, which identifies the number of characters matched so far, is not modified during the copy operation.

Preferably, we wish to avoid collisions whenever possible. If too many collisions occur, then instead of having a lot of patterns stored in the linked lists, we will have many pattern ranges stored. To avoid collisions, we exploit two facts: (1) the trie constructed by the `addpatterns` function attempts to add patterns as soon as possible before the maximum trie height is reached, and (2) if the maximum trie height is greater than one, failure transitions from a leaf node can never point to another leaf node. Instead of copying pattern ranges as soon as a collision is detected, we first attempt to push from both nodes A and B down the trie. This reduces the probability of a collision by 256^2 times if ranges from both A and B can be pushed down, or by 256 times if only one of the ranges can be pushed down. The only time pushing down is not possible is if ranges for either A or B contain short patterns or have patterns with a wildcard character in the position equal to the level of the node. If a pattern is already stored on the leaf node, we are guaranteed that this node's pattern ranges will not collide with any other node.

Figure 7 shows a final trie constructed by our algorithm. Patterns beginning with characters $\langle 0, 254 \rangle$ are stored at level two (node A) because either they are short or they have a wildcard character in position two. These patterns are copied by node B due to a collision. The rest of the patterns are stored under leaf nodes.

To summarize, Oyster takes the following steps to construct a trie:

1. Read all patterns from the virus database and store them in a sorted array.
2. Call the `addpatterns` function to build a trie and initialize success transitions.

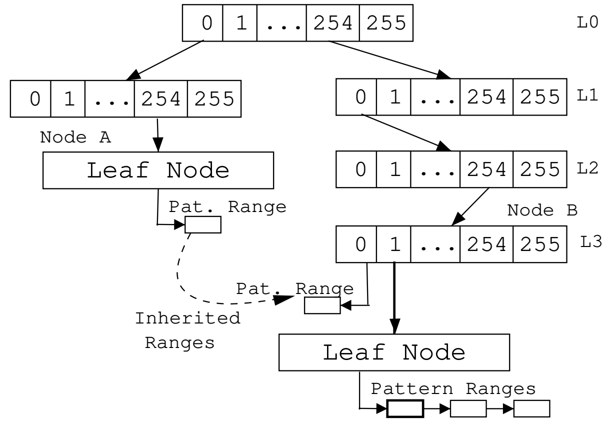


Figure 7: Final trie structure for Oyster. Only success transitions are shown.

3. Execute the pattern-collision detection and avoidance procedure.
4. Set up the failure transitions.

3.2.5 Oyster File System Integration

Oyster provides a simple interface to the file system to perform scanning. It exports a `scanbuf` function which is responsible for scanning a buffer for viruses. The `scanbuf` function supports two modes of scanning: full mode, which scans for all patterns, and regular mode, which scans all regular (non multi-part) patterns. The `scanbuf` function takes the following five parameters: (1) a buffer to scan, (2) the buffer length, (3) the buffer's position in the file, (4) an Oyster state structure, and (5) various flags that determine the scan mode, state handling, and other aspects of the operation. The return code of this function indicates whether the buffer is clean or infected.

The state structure enables Oyster to continue scanning the next buffer right from where the previous call to `scanbuf` left off. The state structure contains the following four members: (1) a linked list of partially-matched patterns represented by the pattern ID and the position of the last character successfully matched, (2) a node ID identifying the trie node where the previous call left off, (3) a structure to keep track of multi-part pattern matches, and (4) a virus database checksum, which we use to check the validity of the state against the currently-loaded database.

We keep only one state structure for each opened inode (file on disk). Multiple processes that read or write to the same file share a single state structure. The size of the state depends on the number of partially-matched viruses, and is usually around 512 bytes. We do not export the state structure to external modules. Instead, we provide functions to allocate and deallocate the structure, as well as functions to serialize and deserialize it so that external modules can store the state persistently.

To load the Oyster module into the kernel, we specify the database files to load as well as the minimum and the maximum trie heights parameters. After the Oyster module is loaded, external file system modules can use Oyster to perform on-access scanning.

3.2.6 Summary of Improvements

Our Oyster scanner improves on ClamAV in two ways: performance and scalability, and kernel integration.

We allow trie heights larger than two, which improves performance logarithmically. Oyster can limit the maximum tree height, to minimize memory usage and improve scalability. We additionally improve performance by allowing pattern scanning to terminate at intermediate trie nodes instead of having to go all the way down to leaf nodes.

ClamAV was designed for scanning whole files in the user level, making assumptions that are unsuitable for running inside kernels. For example, ClamAV scans entire files sequentially, 132KB at a time. Oyster, on the other hand, uses data units that are native to the kernel, scanning one page at a time (4KB on IA-32 hosts). Finally, whereas ClamAV scans whole files sequentially, Oyster scans individual pages as they are being accessed—*regardless* of the order in which they are accessed. This improves performance and guarantees that no infected data is ever leaked. We introduced a state structure to incrementally record the partial scan status of individual pages, and also found that this structure improves performance by up to 68% as compared to ClamAV.

4 The Anti-virus File System

We designed Avfs to achieve the following three goals:

Accuracy and high-security: We achieve this by detecting viruses early and preventing viruses from corrupting the file system.

Performance: We perform partial scanning and avoid repetitive scanning.

Flexibility and portability: Being a stackable file system, Avfs is portable. Moreover, user-oriented features such as forensics and versioning provide flexible options for deployment.

Avfs is a stackable file system for Linux that interfaces with Oyster, as described in Section 3, to provide virus protection. The advantages of being a stackable file system include transparent operation and portability to a variety of other file systems. A state-oriented approach allows Avfs to perform partial and non-repetitive scanning.

4.1 State-Oriented Design

There are two types of state involved in providing on-access virus protection in our system. The first allows safe access to files through the `read` and `write` methods by tracking patterns across page boundaries. This state is computed by Oyster and is maintained by Avfs. The second type of state is used to avoid repetitive scanning and is stored persistently as part of a file by Avfs.

The Oyster scanning module can partially scan files. Oyster can scan one part, say b_1 of a buffer $B = b_1 + b_2$ and compute a state s_1 at the end of this scan. State s_1 and the second part of the buffer, b_2, can be passed to Oyster and the effect of these two scans would be as if buffer B was scanned all at once. Avfs maintains this Oyster state for each file in the file system. When the file is being accessed, this state is kept in memory as part of the in-memory inode structure of the file. We record this state after each page scan, thereby overwriting the previous state.

We do not maintain state for individual pages because the current stackable file system infrastructure has no provision for it. Also, it might be expensive in terms of space utilization. We could store all the state for multiple pages in a single structure, but with increasing file sizes, maintaining this structure becomes expensive.

Our state design divides a file logically into two parts: one scanned and the other unscanned. Along with this state, Avfs also records the page index to which this state corresponds, so that Avfs can provide subsequent pages for scanning in the correct order. When a file is closed, we store this state persistently so that we can resume scanning from where we left off. We use an auxiliary *state file* for each file in a separate hidden directory under the Avfs mount called the *state directory*. Avfs traps the `lookup` and `readdir` operations to prevent access to this directory and its contents by users. The state file's name is a derivative of the inode number of the corresponding file. This facilitates easy access of the state file because the inode number of a file can be easily obtained and thus the state file name can be easily generated. When a file is closed, the entire state (Oyster state + page index) is written into its state file.

In addition to the Oyster state, Avfs has some state of its own which allows it to mark files *clean*, *quarantined*, or *unknown*. These file states are stored as flags in the main file's on-disk inode structure. To quarantine a file we change its permissions to 000, so that non-root users could not access it. Also, if the underlying file system is Ext2/3, we set the *immutable* flag so that even root could not modify the file without changing its attributes.

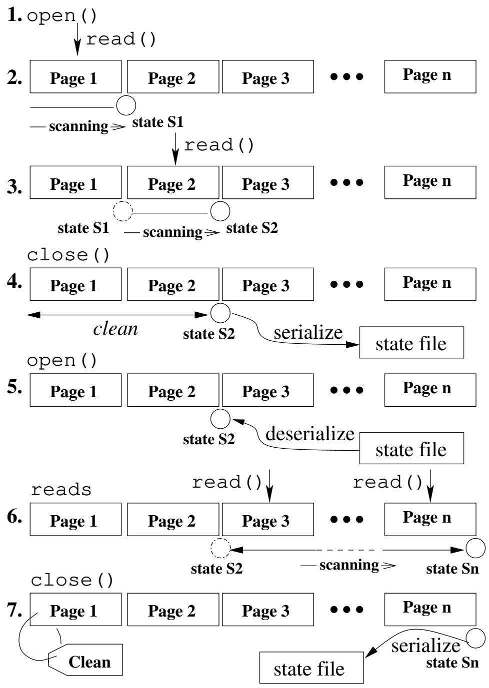


Figure 8: Typical operations on files and their processing in Avfs.

Figure 8 illustrates a few operations and their effects on a file under Avfs in a few simple steps:

1. During the first `open` on an unknown file, there is no state associated and the operation proceeds normally.
2. On a `read` of page 1 of the file, the data is scanned and a state $S1$ is computed by Oyster at the end of this scan. This state corresponds to the first page.
3. Reading to the next page of the file makes use of the previous state $S1$ for scanning.
4. When a file is closed, the state needs to be stored persistently. A serialized form of the state is stored into an auxiliary state file.
5. Another `open` on the file causes the state to be deserialized from the state file and brought back into memory for further scanning.
6. Further sequential reads of the file make use of the previous state and ultimately the file gets scanned completely.
7. If the file has been scanned completely, then during its `close`, the latest state is written to the state file and the file is marked *clean*. A clean file is not scanned during subsequent accesses unless it is modified.

This state-oriented design provides a basis for a variety of features, described next.

4.2 Modes of Operation

We designed two scanning modes: a *full* mode that scans for all patterns in Oyster's virus database and a *regular* mode that scans for regular patterns only. We also designed two user-oriented forensic modes for different classes of users and sites of deployment. These two forensic modes are called *Immediate* and *Deferred*. Avfs can be mounted with any combination of scanning and forensic modes using mount-time options. We describe the two scan modes in Section 4.2.1 and the forensic modes in Section 4.2.2

4.2.1 Scan Modes

Regular and multi-part (polymorphic) patterns are explained in detail in Section 3.1. In full mode, Oyster scans input for all the patterns (regular and multi-part) in the database, making use of its full trie structure. Scanning multi-part patterns can be expensive in terms of speed because they can span several pages of a file. Writes to random locations in a file can cause repetitive scanning of the whole file. In regular mode, Oyster scans input only for regular patterns. The full mode is accurate in the sense that it scans the input for all patterns (including multi-part patterns using their virus definitions) in the database. The regular mode trades-off accuracy for speed by scanning only for regular patterns.

The semantics of the full mode are different for unknown and clean files. For unknown files during sequential reads, we always have state at the end of the previous page and as we progress toward the end of the file it gets gradually scanned and finally is marked clean. If we have random reads from different parts of the file, we adopt a different strategy. Random reads ahead of the current scanned page trigger a scan of the intermediate pages. For random reads before the current scanned page, we simply ignore scanning because that part of the file has already been scanned.

Sequential writes are dealt with in the same way as sequential reads, but the case of random writes is slightly different. Multi-part patterns of the form $P = \{p1, p2, p3, \dots, pn\}$ are hard to detect because we could have the following scenario. Consider a multi-part pattern $P = \{p1, p2, p3\}$ and an empty file. The first write could produce $p2$ on page 2. Our scanner would scan the file until the end of page 2 and find that the file is clean. The next write could be $p3$ on page 3. The final write of $p1$ to page 1 would complete the whole virus in the file. To avoid this, the state maintained after the write to page 3 should be invalidated during the write to page 1 and the whole file should be scanned to detect the virus.

We implemented this technique which scans the whole file for multi-part patterns using the multi-part trie structure of Oyster on every random `write`, but it proved to be inefficient because some programs like `ld` write randomly around the two ends of the file and hence cause a number of rescans over the entire length of the file. Our current implementation, therefore, has a *full-*

scan-on-close flag on the file's inode which, when set, causes the file to be scanned completely for multi-part viruses when the file is closed. This flag is set if there are random writes before the current scanned page, and the page is scanned only for regular patterns using a *cushioned* scan implementation. A cushioned scan works by extending the concerned page with sufficiently large buffers of data on either side to guarantee that patterns are detected across page boundaries. The size of a cushion buffer is equal to the length of the longest pattern available in Oyster's database (currently 2,467 characters). A cushioned scan is shown in Figure 9.

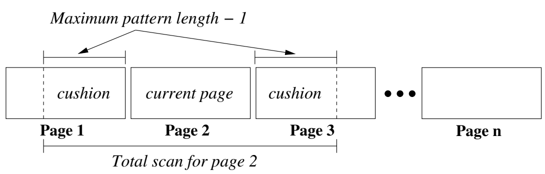


Figure 9: A cushioned scan implementation for viruses spread across page boundaries

We have a configurable parameter *max-jump* that decides when the full-scan-on-close flag should be set. Forward random writes to pages within max-jump from the current scanned page cause intermediate pages to be scanned for all viruses, but forward writes to pages greater than max-jump cause only the regular patterns to be scanned for, and the full-scan-on-close flag is set. If this flag is set, access to the file is denied for all other processes except the current process performing the random writes. When the file is closed, the state is stored persistently in the file's state file. If the state indicates that the last page was scanned, the file is safely marked clean; otherwise, the state represents the last scanned page of the file.

After a file is marked clean, new reads do not result in additional scans. Appends to a clean file are dealt with in the same way as sequential writes for unknown files (files that are not marked clean). Writes to the middle of the file are handled in a similar fashion as writes to unknown files.

The regular mode is almost identical to the full mode except that it scans the input only for regular patterns. Reads to unknown or clean files, as well as sequential writes, are treated in the same way as in full mode. Random writes past the current scanned page force scanning of intermediate pages, and backward writes use cushioned scanning. The value of the maximum pattern length is currently less than a page size, so cushioned scanning adds a maximum overhead of scanning one page on either side. This overhead is less than in full mode because in full mode we have to scan the entire file to detect multi-part viruses. For large files, full mode is slower than the regular mode. Regular mode is useful in cases where random read and write performance is important. Full mode should be used when detection of multi-part patterns is required.

4.2.2 Forensic Modes

If a process writes a virus into the file system, the process should be notified of this behavior as early as possible. Also, if a process is reading from an infected file, then the read must not succeed. These requirements motivated the development of an *immediate* mode that would not let viruses to be written to disk and return an error to the process so that it can take remedial action. Immediate mode is especially suited to a single-user environment where protection from viruses is the prime requirement.

In addition to the immediate mode, we developed another mode called the *deferred* mode which defers the error notification and records malicious activity. Such a mode is suitable for large enterprise servers where several users access data concurrently. In addition to on-access virus protection, this mode provides (1) data consistency by backing up files, and (2) a mechanism to trace processes that attempt to write viruses into the file system. We keep evidence such as process information, time of attack, and infected files, so that the incident can be investigated later.

When Avfs is mounted over an existing file system, it is possible for the underlying file system to already contain some virus-infected files. Such existing virus-infected files are detected during reads from the file. In this case, the file is quarantined, so that no process could access it. Another possibility of a virus infection is through a process that tries to write a virus into the file system. In the immediate mode, these writes are trapped in the Avfs layer itself and are not allowed to propagate to the lower file system. Permission for such a write is denied and the offending process is immediately notified of the corresponding error. The file remains consistent up to the last successful write to the file. The file, however, may have multi-part viruses or viruses that span over page-boundaries. A multi-part virus is not detected until all of its parts are discovered in the correct sequence in a file. At the same time, a file cannot be called clean if it contains even one part of a multi-part virus. Hence, if most parts of a multi-part virus are written and the virus is detected on writing the last part, the file could still be corrupted due to the previous virus parts. If a regular virus spans across page boundaries, only the last write to the page that completes the virus is denied.

Deferred mode operates in the same way as immediate mode for existing virus-infected files. These files are simply quarantined and access to them is denied. Attempts to write viruses, however, are treated differently. Files can have multiple instances open simultaneously.

An `open` on a file causes an instance to be created and a `close` on that instance causes the instance to terminate. We define a *session* to be the duration between the first open of the file and the last close of the file. We back up a file during the first write of a session. We keep this backup in case a virus is created in the course of this session. In such a situation, we revert to the backup and restore the file to a consistent state. Here, we have only one version of the file which prevents us from reverting to versions more than one session old. If some parts of a multi-part virus are written over several sessions, we cannot revert to totally clean versions (without any part of the virus) because we cannot detect such a virus until all its parts are written. With multiple file versions, however, this problem is easily solved.

When an attempt to write a virus is detected, we record the time of the event and the process identifier (PID) of the offending process. We do not return an error immediately to the process. Instead, we lead it to believe that the write was successful and allow it to proceed writing. However, we prevent read access to the file for the offending process, so that it cannot read the virus it had written. In addition to that, we also deny all access (read, write, open) to the file for all other processes. On close of the session, we rename the infected file to a new name with the recorded PID and time stamp as an evidence of the offense. Then, we rename the saved backup to the original name so that data-consistency is ensured. The saved evidence file can be used to launch an investigation into the incident.

5 Related Work

There are several anti-virus systems available today. Most of these systems are commercial products: Symantec's Norton Antivirus [18], McAfee Virusscan [13], Sophos [17], Anti-Virus by Kaspersky Lab [9], Computer Associates's eTrust [4], and others. To protect trade secrets, little information is released about their internals. Their development is closed, and there is no opportunity for peer review. Although the internals of these products are trade secrets, advertised information and white papers suggest their general structure. Most of the commercial scanners detect viruses using virus signature databases. These scanners boast large virus databases ranging anywhere from 65,000 to 120,000 patterns, which have been built over long periods. They also use heuristic engines for scanning. The heuristic engines eliminate files that cannot contain viruses, and scan only the suspicious ones. Such heuristics typically include identifying executable file types, appropriate file sizes, and scanning only certain regions of files for viruses.

Some commercial scanners offer real-time virus protection. Real-time protection involves scanning files for viruses when they are used. This is done by intercepting the `open`, `close`, or `exec` system calls and scanning entire files when these system calls are invoked. Scanning during an open system call detects a virus only if the file is already infected. If a virus is not present during the open, but is written into the file after the open operation, on-`open` scanning does not detect it. This is the reason most real-time scanners scan for viruses on `close` of the file as well. This scheme has three drawbacks. First, viruses can be detected only after they have been written to the file. If the file cannot be repaired, critical data cannot be restored. Second, multiple processes can access a virus in a file before the file is scanned during `close`. Third, scanning files on both `open` and `close` results in scanning them twice.

ClamAV [11] is a open-source system which forms the basis of our scanning engine. ClamAV maintains an up-to-date virus-definition database; it has been adopted as the primary virus-scanner in many organizations and is the basis for several open-source projects.

Dazuko is a kernel module that provides third-party applications an interface for file access control [6]. Dazuko was originally developed by H+BEDV Datentechnik GmbH, but has been released as free software to encourage development and to enable users to compile the module into their custom kernels. Dazuko intercepts the `open`, `close`, and `exec` system calls. It passes control to virus-scanners during these system calls to perform on-access (on-`open`, on-`close`, on-`exec`) virus scanning. Clamuko [11] (the on-access scanner from ClamAV) and H+BEDV [7] are two virus scanners that use Dazuko. One disadvantage of using systems like Dazuko is that its kernel module has to communicate with user-level virus scanners, slowing performance. Sockets or devices are used for communication, so data also has to traverse protocol layers. Finally, data-copies have to be performed between the kernel and the user-level.

The *Internet Content Adaptation Protocol* (ICAP) [8] is a protocol designed to off-load specific Internet-based content to dedicated servers, thereby freeing up resources and standardizing the way features are implemented. ICAP servers focus on providing specific functionality such as spam filtering or virus scanning. The downside of this scheme is performance: data is transfered over the network to the virus-scanning server.

6 Evaluation

We evaluated the performance of Avfs under a variety of system conditions by comparing it to other commercial and open-source anti-virus systems.

All benchmarks were performed on Red Hat Linux 9 with a vanilla 2.4.22 kernel running on a 1.7GHz Pentium 4 processor with 1GB of RAM. A 20GB 7200 RPM Western Digital Caviar IDE disk was used for all the experiments. To ensure cold-cache results, we unmounted the file systems on which the experiments were conducted between successive runs. To reduce I/O effects due to ZCAV, we located the tests on a partition toward the outside of the disk that was just large enough for the test data [5]. We recorded the elapsed, system, and user times for all tests. We computed the wait time, which is the elapsed time minus the CPU and user times used. Wait time is primarily due to I/O, but other factors such as scheduling can affect it. Each test was run at least 10 times. We used the Student-*t* distribution to compute 95% confidence intervals for the mean elapsed, system, and user times. In each case the half-widths of the confidence intervals were less than 5% of the mean. The user time is not affected by Avfs because only the kernel is changed; therefore we do not discuss user time results.

In Section 6.1 we describe the configurations used for Avfs. Section 6.2 describes the workloads we used to exercise Avfs. We describe the properties of our virus database in Section 6.3. In Section 6.4 we present the results from an Am-Utils compile. Section 6.5 presents the results of Postmark. Finally, in Section 6.6 we compare our scanning engine to others.

6.1 Configurations

We used all the combinations of our scanning modes and forensic modes for evaluating Avfs.

We used two scanning modes:

FULL: Scan for all patterns including multi-part ones.

REGULAR: Scan only for regular patterns.

Each scanning mode was tested with both of our forensic modes:

IMMEDIATE: This mode returns an error to the process immediately and does not allow malicious writes to reach the disk.

DEFERRED: This mode backs up a file so it can be restored to a consistent state after an infection and provide information to trace malicious activity.

We used the default trie minimum height of three and maximum of three for all tests, unless otherwise mentioned. A minimum height of three gave us the best performance for all databases and a maximum height of three gave us the best performance for databases of small sizes like 1K, 2K, 4K and 8K patterns. We show later, in Section 6.4, how the maximum height parameter can be tuned to improve performance for large databases.

For commercial anti-virus products that support on-access scanning, we ran the Am-Utils compile and the Postmark benchmarks. Clamuko and H+BEDV are in this category. Sophos and some other commercial virus scanners do not support on-access scanning trivially, so we compared the performance of our scanning engine to these on large files using command line utilities.

6.2 Workloads

We ran three types of benchmarks on our system: a CPU-intensive benchmark, an I/O intensive one, and one that compares our scanning engine with anti-virus products that do not support on-access scanning.

The first workload was a build of Am-Utils [14]. We used Am-Utils 6.1b3: it contains over 60,000 lines of C code in 430 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories. Though the Am-Utils compile is CPU intensive, it contains a fair mix of file system operations, which result in the creation of several files and random read and write operations on them. For each file, a state file is created and backups of files are created in the deferred forensic mode. These operations of the Am-Utils benchmark sufficiently exercise Avfs. We ran this benchmark with all four combinations of scanning and forensic modes that we support. This workload demonstrates the performance impact a user might see when using Avfs under a normal workload. For this benchmark, 25% of the operations are writes, 22% are `lseek` operations, 20.5% are reads, 10% are `open` operations, 10% are `close` operations, and the remaining operations are a mix of `readdir`, `lookup`, etc.

The second workload we chose was Postmark [10]. Postmark simulates the operation of electronic mail servers. It performs a series of file system operations such as appends, file reads, directory lookups, creations, and deletions. This benchmark uses little CPU, but is I/O intensive. We configured Postmark to create 500 files, each between 4KB and 1MB, and perform 5,000 transactions. We chose 1MB as the file size as it was the average inbox size on our large campus mail server. For this configuration, 45.7% of the operations are writes, 31.7% are reads and the remaining are a mix of operations like `open`, `lookup`, etc. (We used Tracefs [2] to measure the exact distribution of operations in the Am-Utils and Postmark benchmarks.)

The third benchmark compares various user-level command-line scanners available today with our scanner. We scanned two clean 1GB files. The first file had random bytes and the second file was a concatenation of files in `/usr/lib`. The latter represents various executables and hence exercises various parts of the scanning trie under more realistic circumstances than random data. Overall, this workload exercises both scanning for viruses and also loading the virus database. Note that the Oyster module and its user-level counterpart have almost identical code with the exception of memory allocation functions (`kmalloc` vs. `malloc`) and some kernel specific data structures.

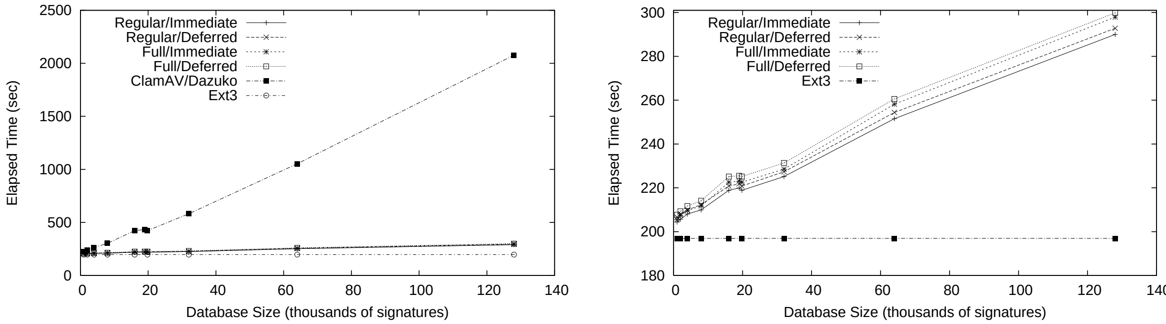


Figure 10: Am-Utils results. The figure on the left shows results for all Avfs modes, ClamAV, and Ext3. The figure on the right shows a detailed view for Avfs and Ext3 (note that the Y axis starts at 180 seconds).

6.3 Test Virus Databases

To evaluate the performance and the scalability of our Oyster virus scanner, we had to generate test virus databases with different numbers of virus signatures. To generate a virus database with fewer than the 19,807 patterns contained in the current ClamAV virus database, we simply picked signatures at random from the original database.

The generation of realistic larger databases was more involved. The most straightforward approach was to simply generate random virus signatures. However, as described in Section 3.2.1, this approach would not yield a representative worst-case virus database. Instead, we obtained the following statistics from the existing ClamAV virus database: (1) the distribution of all unique four character prefixes, D_p; (2) the distribution of virus signature lengths, D_l; (3) the percentage of multi-part patterns in the database, P_m; and (4) the distribution of the number of sub-patterns for each multi-part pattern, D_s. The prefixes of length four were chosen because this number was larger then the minimum trie height parameter in our experiments. To generate one signature, we first determined at random whether the new signature will be a basic or a multi-part signature using the percentage P_m. If the new signature is a multi-part signature, we determined the number of sub-parts based on the distribution D_s, and then generated one basic pattern for each sub-part. To generate a basic signature, we randomly sampled from the distribution D_p to determine the prefix that will be used for this pattern; next, we sampled from the distribution D_l to determine the length of the signature. If the length is greater than four bytes, the remaining characters are generated randomly. The above process was repeated as many times as necessary to generate a database of the desired size. For our evaluation, we generated databases ranging from 2^{10} to 2^{17} (128K) signatures. We verified that the resulting databases had distribution characteristics similar to the current ClamAV database.

6.4 Am-Utils Results

Figure 10 shows the performance for the Am-Utils compile benchmark using various database sizes. The left hand side of this figure shows the results for four Avfs modes, ClamAV, and Ext3. The right hand side of this figure shows a detailed view for Avfs modes and Ext3. Table 1 summarizes the Am-Utils benchmark results.

	Ext3		**Full Deferred**		**ClamAV**	
Size	**Elapsed**	**System**	**Elapsed**	**System**	**Elapsed**	**System**
1K	196.9	42.4	207.7	52.1	225.4	81.5
4K	196.9	42.4	211.7	56.2	262.9	118.3
19.8K	196.9	42.4	225.5	69.7	433.5	289.3
64K	196.9	42.4	260.6	105.4	1052.8	908.8
128K	196.9	42.4	299.9	144.5	2077.4	1933.0
Overhead over Ext3 (%)						
1K	-	-	5.5	22.8	14.5	92.2
4K	-	-	7.5	32.5	33.5	179.0
19.8K	-	-	14.5	64.3	120.2	582.3
64K	-	-	32.3	148.4	434.7	2043.4
128K	-	-	52.3	240.4	955.0	4459.0

Table 1: Am-Utils build times. Elapsed and System times are in seconds. The size of 19.8K corresponds to the current ClamAV database.

The Oyster scanner was configured to use trie heights of three for both minimum and maximum trie height parameters. A minimum height of three gave us the best performance for all databases and a maximum height of three gave us the best performance for databases of small sizes like 1K, 2K, 4K, and 8K patterns. We demonstrate later in this section how the maximum height parameter can be varied to improve performance for large databases.

All of the modes have similar overhead over Ext3, with the slowest mode, FULL/DEFERRED, being 0.5–2.7% slower than the fastest mode, REGULAR/IMMEDIATE. In the FULL/DEFERRED mode, the elapsed time overheads over Ext3 varied from 5.5% for a 1K pattern database to 52.3% for a 128K pattern database, whereas the system time overhead varies from

22.8% to 240.4%. Due to I/O interleaving, a large percentage increase in the system time does not result in the same increase in elapsed time. The increase in the elapsed time is almost entirely due to the higher system time. This increase in system time is due to the larger database sizes. The Oyster module proved to scale well as the database size increased: a 128 times increase in the database size from 1K to 128K patterns resulted in elapsed time increase from 207.7 seconds to 299.9 seconds, a merely 44.4% increase in scan times. For the same set of databases, ClamAV's elapsed time increases from 225.4 seconds to 2,077.4 seconds—a 9.2 factor increase in scan times.

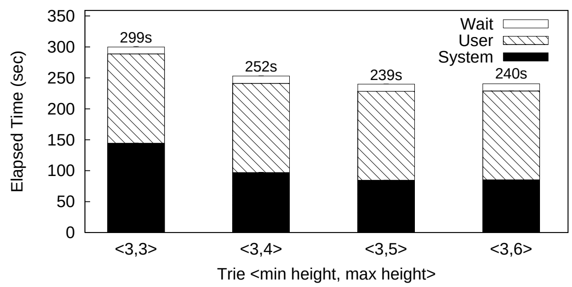


Figure 11: Am-Utils build times using the FULL/DEFERRED *mode with a 128K signature database.*

Max Level	Mem Usage	Δ Time	Speed Gain
3	45MB	0s	0s
4	60MB	−45s	45s
5	182MB	−58s	13s
6	199MB	−57.5s	−0.5s

Table 2: Effect of the maximum trie height parameter on speed and memory usage. The speed gain column shows the speed improvement over the previous maximum trie level.

The scalability for larger databases can be further improved by adjusting the minimum and maximum trie height parameters. We configured Avfs to use the slowest mode, FULL/DEFERRED. The Oyster module was configured to use a database of 128K virus signatures, and a minimum height of three. We repeated the Am-Utils benchmark with various maximum trie height parameters. Figure 11 shows the result of this experiment. Table 2 summarizes improvements in speed and increases in memory usage. A maximum height of five proved to be the fastest, but a height of four provided a reasonable increase in speed while using significantly less memory than a height of five. A system administrator has a lot of flexibility in tuning the performance of the system. If speed is very important, then a maximum trie height of five can be used. However, if the memory availability is tight, then a maximum trie height of four provides a reasonable performance improvement with a smaller memory footprint than a height of five.

6.5 Postmark Results

Figure 12 shows the results of running Postmark with on-access scanners: ClamAV, H+BEDV, and all four modes of Avfs. It also shows the time taken for a Postmark run on Ext3.

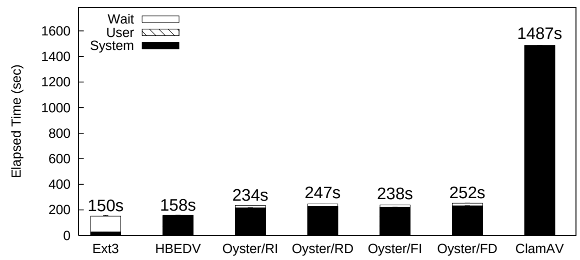


Figure 12: Postmark results. For Avfs, R represents REGULAR *mode, F* FULL, *I* IMMEDIATE, *and D* DEFERRED.

Postmark is an I/O intensive benchmark, which creates a lot of files and performs read and write operations on these files. The files are accessed at random, which results in a file being opened and closed several times during the benchmark.

When a file is created in Postmark, Avfs scans this file and creates a state file for it. Once a state file is present, no additional scanning is required during subsequent reads. Since all the writes are appends, the state in the state file is always valid and only the last page(s) ahead of the current scanned page are scanned. ClamAV's scanner is called several times on the open and close system calls to scan entire files, which contributes to its overhead. The slowest mode of Avfs, FULL/DEFERRED, takes about 252 seconds and Ext3 takes 150 seconds, which is an overhead of 68%. For the same benchmark, ClamAV takes 1,487 seconds—a factor of 9.9 slower.

H+BEDV has a heuristics engine that allows it to determine if a file needs to be checked for viruses by looking at the first few bytes in the file. This allows H+BEDV to skip scanning entire files of types that cannot be infected. Although it is possible to suppress the heuristics engine of H+BEDV in the command-line scanner, this option is not available in the on-access scanner. Due to its heuristics engine, H+BEDV shows almost identical performance to Ext3 with a 5% overhead in elapsed time but the system time increases by a factor of 5.9.

6.6 Scan Engine Evaluation

Our test consisted of scanning two large files, one with random data and the other with data that contained executable code from library files. None of the test files contained any viruses so the files were completely scanned.

When a file is given to a command-line scanner for scanning, it needs to set up the scanning trie before scanning can start. The time taken to set up this trie depends on the size of the virus database, so we used the same size database as the other scanner for Oyster. For example, when comparing it to Sophos we used 86,755 patterns, because that is what Sophos reports as their database size.

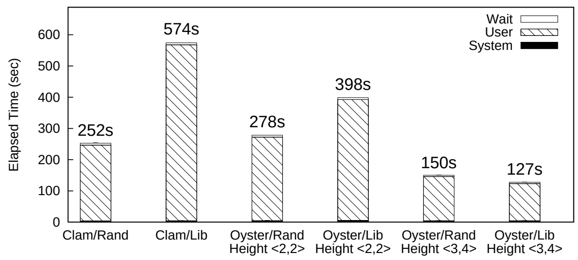


Figure 13: File scan times for Oyster and ClamAV. The database had 19,807 patterns.

Figure 13 compares the performance of ClamAV with Oyster. We ran the benchmark for Oyster once with the minimum and maximum heights set to two, and once with the heights set to 3 and 4, respectively. The $\langle 2,2\rangle$ setting matches the ClamAV trie structure, and $\langle 3,4\rangle$ optimizes Oyster's performance for this benchmark.

For the Oyster scanner configured with $\langle 2,2\rangle$, the random file scan was 26 seconds slower, while the library file scan was 176 seconds faster than ClamAV. The Oyster scanner was faster for the library scan because the internal state structure maintains partially-matched patterns between successive calls to the scanner. The ClamAV scanner does not have such a structure. Instead, it rescans some of the text from the previous buffer so that patterns that span multiple buffers are detected. In a library file scan, ClamAV scanned a total of 6.2 billion patterns, while Oyster scanned 1.8% fewer patterns. In the random file benchmark, ClamAV again scanned 1.8% more patterns. Even though the Oyster scanner scanned six million fewer patterns, the overhead of maintaining the state, which includes additional `malloc` calls and linked list operations, exceeded the savings gained in scanning fewer patterns.

Increasing the trie height parameters to $\langle 3,4\rangle$ significantly reduces the number of patterns scanned by Oyster. For the random file benchmark, Oyster scanned 369 times fewer patterns. For the library benchmark, Oyster scanned 52 times fewer patterns.

Figure 14 compares H+BEDV with Oyster. In this benchmark, we ran the command-line scanner from H+BEDV. We configured H+BEDV so that it scans all input without the heuristics engine. H+BEDV is slower with random input than with the library file. This suggests that the commercial H+BEDV scan engine is optimized to scan executable content, possibly by using a different scanning mechanism. The random file scan of

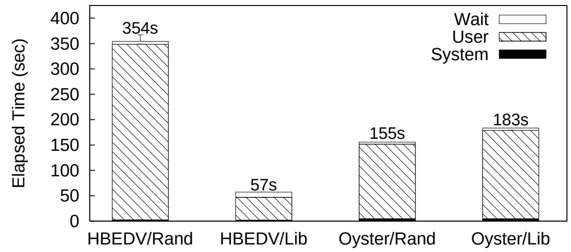


Figure 14: File scan times for Oyster and H+BEDV. The database had 66,393 patterns.

H+BEDV is 6.2 times slower than H+BEDV's scan of the library file. For 1GB of random input, Oyster is 56% faster than H+BEDV. For a 1GB library file however, Oyster is 3.2 times slower than H+BEDV.

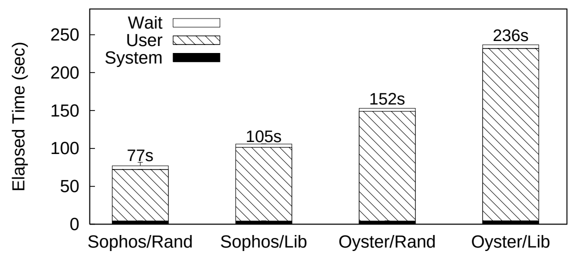


Figure 15: File scan times for Oyster and Sophos. The database had 86,755 patterns.

Figure 15 compares Sophos, an optimized commercial product, with Oyster. Oyster is slower by 97% for random input and by 124% for the library file, suggesting that Oyster can be further optimized in the future.

7 Conclusions

The main contribution of our work is that for the first time, to the best of our knowledge, we have implemented a true 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 reduces the possibility of a virus attack and can trap viruses before they are written to disk. In addition to providing data consistency by backing up files, the forensic modes of operation in Avfs provide means to track malicious activity by recording information about malicious processes.
- Our Oyster scan engine improves the performance of the pattern-matching algorithm using variable trie heights, and scales efficiently for large database sizes. State-based scanning in Oyster allows us to scan a buffer of data in parts. This state-oriented

design reduces the amount of scanning required by performing partial file-scanning.
- By separating the file system (Avfs) from the scan-engine (Oyster), we have made the system flexible and extensible, allowing third-party virus scanners to be integrated into our system.

7.1 Future Work

We plan to improve the Oyster scanning engine in a variety of ways. Oyster scans all files, even those that are not executable. We plan to allow Oyster to scan for viruses within only specific file types. For example, when scanning a Microsoft Office document, Oyster will scan only for macro viruses. Since some viruses can only occur in certain segments of a file, it is not necessary to scan for them in the rest of the file. We plan to integrate positional matching into Oyster, so that only relevant portions of files are scanned. Also, we plan to scan for all patterns on a leaf node of the scanning trie simultaneously instead of scanning for each pattern sequentially, thereby improving the scan engines performance.

We plan to maintain Avfs state for more than one page per file. These states can be used to scan for multi-part patterns efficiently even during random writes. Our investigation will evaluate trade-offs between storing more and less state information. We plan to add more forensic features to the deferred mode, such as terminating the offending processes, and storing the core dump of the process along with other useful evidence of malicious activity. We also plan to integrate a versioning engine [12] with Avfs to support multiple levels of versioning. We can keep track of changes to a file across several versions to provide more accurate forensics.

The Oyster scan engine can also be applied to generic pattern matching. Rather than using a database of viruses, Oyster could use a database of keywords. For example, a brokerage firm could flag files for review by a compliance officer. Other companies could flag keywords related to trade secrets. We plan to investigate what file system policies would be useful for such applications.

8 Acknowledgments

We thank the Usenix Security reviewers for the valuable feedback they provided. We also thank the ClamAV developers for providing an open source virus scanner and for their useful comments during the development of Oyster. This work was partially made possible by an NSF CAREER award EIA-0133589, NSF Trusted Computing award CCR-0310493, and HP/Intel gift numbers 87128 and 88415.1.

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] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004.

[3] F. Cohen. Computer viruses: theory and experiments. *Computers and Security*, 6(1):22–35, 1987.

[4] Computer Associates. eTrust. `www3.ca.com/Solutions`, 2004.

[5] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, June 2003.

[6] H+BEDV Datentechnik GmbH. Dazuko. `www.dazuko.org`, 2004.

[7] H+BEDV Datentechnik GmbH. H+BEDV. `www.hbedv.com`, 2004.

[8] J. Elson and A. Cerpa. Internet Content Adaptation Protocol (ICAP). Technical Report RFC 3507, Network Working Group, April 2003.

[9] Kaspersky Lab. Kaspersky. `www.kaspersky.com`, 2004.

[10] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[11] T. Kojm. ClamAV. `www.clamav.net`, 2004.

[12] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004.

[13] Network Associates Technology, Inc. McAfee. `www.mcafee.com`, 2004.

[14] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[15] J. Reynolds. The Helminthiasis of the Internet. Technical Report RFC 1135, Internet Activities Board, December 1989.

[16] R. Richardson. Computer Crime and Security Survey. *Computer Security Institute*, VIII(1):1–21, 2003. `www.gocsi.com/press/20030528.html`.

[17] Sophos. Sophos Plc. `www.sophos.com`, 2004.

[18] Symantec. Norton Antivirus. `www.symantec.com`, 2004.

[19] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.