Hash-AV: Fast Virus Signature Matching by Cache-Resident Filters

Ozgun Erdogan and Pei Cao
Department of Computer Science
Stanford University
Stanford, CA 94305

***Abstract*— Fast virus scanning is becoming increasingly important in today's Internet. While Moore's law continues to double CPU cycle speed, virus scanning applications fail to ride on the performance wave due to their frequent random memory accesses. This paper proposes Hash-AV, a virus scanning "booster" technique that aims to take advantage of improvements in CPU performance. Using a set of very cheap hash functions and a bloom filter array that fits in CPU second-level (L2) caches, Hash-AV determines the majority of "no-match" cases without accesses to main memory. We investigate the design choices of Hash-AV, showing that "bad but cheap" hash functions such as the mask operation can work well in serial applications of hash functions.**

Through experimentation, we show that Hash-AV improves the performance of the open-source virus scanner Clam-AV by a factor of 8 to 10 on the current signature database, and by a factor of 26 on databases with 120K signatures. Furthermore, Hash-AV combined with Clam-AV can scan for viruses at over 100Mb/s throughput, making it suitable for network-based virus scanning or on-access scanning of socket send/receive calls.

I. Introduction

In the age of Internet and the Web, viruses proliferate and spread easily. As a result, anti-virus technologies are a must in today's wired world. An effective defense needs virus-scanning performed at every major network traffic stop and at the end-host computers. Today, anti-virus software applications scan traffic at e-mail gateways and corporate gateway proxies[1], and they run on end-hosts such as file servers, desktops and laptops. In other words, virus-scanning is becoming a necessary overhead for almost all data communications over the network. As network throughput and user demand for communication speed increase, the speed of virus-scanning needs to keep pace.

Virtually all virus-scanning programs spend the bulk of their time matching data streams with a set of known virus signatures, and they all utilize some form of multi-pattern matching algorithm. The number of virus signatures today is over 100,000 and is growing constantly. As a result, data structures used by these algorithms cannot fit in the CPU cache, and instead reside in main memory. Traditional matching algorithms require at least one random memory access per scanned byte [15].

The performance of random memory accesses, however, does not improve nearly as much as the CPU speed or even sequential memory access throughput. For example, in the past decade, CPU processing speed has been doubling every 18 months, yet memory speed only improved at a rate of less than 10% per year. As a result, anti-virus software using traditional algorithms only partially benefit from the improvements in microprocessor speeds. Random memory access performance becomes the bottleneck for these scanners.

In this paper, we propose a new virus scanning technique that aims to take advantage of improvements in CPU performance. Our solution, called "Hash-AV", combines a set of very cheap hash functions and a bloom filter array that fits in the CPU second-level (L2) cache. Hash-AV can determine the majority of the "no-match" cases quickly, without incurring main memory accesses. Since the majority of network traffic does not contain viruses, most of the data stream belongs to the "no-match" case.

The intuition behind Hash-AV is the following. Currently, one L2 cache miss costs about 100-200 CPU cycles, and the speed gap keeps increasing [7]. While reading new incoming data, the CPU has to take a cache miss. Once a piece of data has been brought from main memory into the CPU cache, Hash-AV can scan the data using its cache-resident bloom filter while the CPU waits for the next cache miss.

To evaluate the performance of our technique, we applied Hash-AV to Clam-AV [11], the most popular open source anti-virus software. Hash-AV improves Clam-

[1]The recent episode of Download.Ject web site infection [14] clearly highlights the importance of scanning viruses at Web proxy gateways.

AV's scanning throughput to 16.6 MB/s for executables, and to 41 MB/s for web pages, on an Athlon XP 2000+. This represents a speed-up factor of 8 and 10, respectively. If polymorphic viruses are handled using emulations, Clam-AV with Hash-AV can scan executables at 27 MB/s and web pages at 104 MB/s. Given that the memory copy speed is 260 MB/s on the Athlon XP 2000+, the results confirmed our intuition that virus scanning can potentially reach memory copy speeds.

In addition to speeding up anti-virus software on end-host computers, we have implemented a technique that enables network-based virus scanning by storing state between consecutive scans. Using Hash-AV, Clam-AV can now easily scan input at over 100Mb/s. Thus, it is possible to embed virus scanning in network routers and switches that have application-recognition and reconstruction capabilities [5]. Embedding virus scanning in network devices enables much faster updates of virus signature databases, and is a valuable addition to existing security mechanisms in a network.

Finally, Hash-AV's application is not limited to virus scanning. It can benefit any string matching applications that have the following characteristics: a) exact matching is required, and the number of signatures is high (e.g. $>$ 10,000); b) the majority of the cases are "no-match" cases; c) no easy "tokenization" methods exist, so the matching has to be tested byte-by-byte. For any such system, Hash-AV principles and designs apply, though the specific choices of hash functions might differ.

II. Virus Scanning Techniques

Before we delve into the discussion of Hash-AV, we first give a brief introduction of the major techniques used in identifying viruses, and describe Clam-AV in more detail.

A. Basic Techniques

There are three main techniques used in virus-scanning:

- signature-matching: check if a file contains a known virus by searching for a fixed string of bytes (the "signature" of the virus) in the data;
- emulation: check if a file contains a polymorphic virus (those that change from occurrence to occurrence) by executing the instructions of an executable in an emulated environment, and then looking for a fixed string (the "signature") in the memory region of the process [22];
- behavior-checking: check if a file contains an unknown virus by running the file in an emulated environment and observing its behavior;

Behavior-checking is typically run on a specific file to determine if the file contains a new virus. Since it is not routinely run, its performance is usually not a concern. Emulation is typically used only on executables matching certain criteria. Signature-matching is routinely run on all files.

In both signature-matching and emulation, most of the CPU time is spent matching the data against a set of strings. Commercial virus-scanning software has databases containing over 100,000 virus signatures. The signatures typically contain at least 15 bytes [10]. The open-source virus-scanning software, Clam-AV [11], currently contains over 20,000 virus signatures. Clearly, efficient string-matching algorithms are a must in virus scanning software.

The most commonly used multi-string matching algorithm is Aho-Corasick [2]. Aho-Corasick works by first building a finite state machine out of the signatures, and then running the data as input through the automaton. Once the automaton is build, scanning N bytes takes $O(N)$ time. An optimized version of Aho-Corasick makes sure that only one access to the data structure representing the automaton is needed for each byte in the input data. The size of the associated data structure, however, is over 90 MB for 20,000 signatures and grows linearly with the size of the signature database. Hence, the data structure cannot fit in today's CPU caches.

Other multi-string matching algorithms include Boyer-Moore [3] and Wu-Manber [24]. Unfortunately, these algorithms do not work well for virus scanning; we discuss some of the issues in the next section.

B. Overview of ClamAV

ClamAV is the most widely used open-source anti-virus scanner available. Currently, it has a database of 20,712 viruses, and consists of a core scanner library and various command-line programs. Besides regular viruses, the database also holds signatures with wild-card characters embedded in them. Signatures with wild-card characters are used to detect polymorphic signatures.

Although a complete description of ClamAV's inner-workings is beyond the scope of this paper, it is important to point out some properties of ClamAV's virus detection algorithm. ClamAV uses the Aho-Corasick algorithm for pattern matching.

ClamAV stores its automaton in a trie data structure. To quickly perform a lookup in this trie, ClamAV uses a 256 element array for each node. It also modifies Aho-Corasick such that the trie has a height of two, and the leaf nodes contain a linked list of possible patterns. ClamAV fixes its trie depth to two because its database

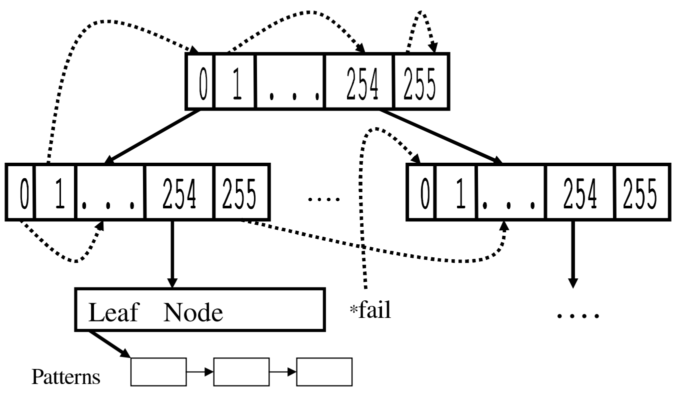

Fig. 1. Part of the trie structure used by ClamAV. Success transitions are shown with solid lines, and failure transitions are represented with dashed lines.

contains polymorphic viruses whose prefixes are as short as two bytes. Figure 1 shows a fragment of this trie structure. As the linked lists get longer, the performance of ClamAV suffers from the cost of traversing the linked list. As a result, ClamAV doesn't scale well with large databases.

Since increasing the depth of the trie improves the scalability of ClamAV, two recent studies explored methods of breaking this restriction. By implementing work-arounds to grow the trie's depth to four, ClamAV developers doubled scanning performance[6]. At the same time, its memory consumption went from 11MB to 90MB. Separately, researchers at Stony Brook also explored techniques of increasing the trie's depth and achieved speeds as high as 3.13 times that of ClamAV[25] on certain files. These approaches, however, have not yet been incorporated into the stable release of ClamAV.

These new approaches are complementary to our Hash-AV technique. Hash-AV still uses the trie structure for exact string matching when necessary, so any improvement in that area improves the overall performance of virus scanning. At the same time, as these approaches increase the trie size, the CPU cache locality becomes poorer, and the exact-matching algorithm becomes increasingly bound by random memory access time. Thus, these new approaches actually have an increased need for techniques such as Hash-AV.

1) Comparing ClamAV and Snort: Snort is a widely used open-source intrusion detection system. It comes with over 1500 signatures. It's interesting to compare ClamAV's choice of algorithms and those of Snort [18].

Unlike virus signatures, a Snort IDS signature is a rule which specifies the port number and type of protocol of the traffic to which the signature should apply. Signatures are grouped into rule sets, each of which applies to a particular protocol or port number. For example, for FTP traffic, Snort only checks signatures that are relevant to FTP. Each rule set varies from 10 to 100 signatures, and the average size is about 40.

Snort uses a string matching engine that performs very well for their rule sets. It uses a modified version of the Wu-Manber algorithm [24]. At start up, the scan engine initializes a bad character shift table and a hash table. Then, at scan time, when the bad character shift fails, the first two bytes of the input are probed into the hash table to find a list of possible matches. Other variations of Snort use Boyer-Moore for rule sets with less than 10 signatures and Wu-Manber for others [23].

The approach taken by Snort, however, cannot be applied to virus scanning. First, the number of virus signatures is much larger. Second, while IDS systems can divide its signatures into rule sets due to the nature of the threats they guard against, virus scanning systems cannot do the same. Even in the relatively small category of macro viruses, there are over 7000 signatures [1]. Thus, virus scanning applications have to use algorithms that scale to a large number of signatures.

C. Handling Polymorphic Viruses

Polymorphic viruses change from instance to instance. It is difficult to construct a fixed "signature" string to identify them. Commercial virus scanning software use both emulation and signature matching to identify polymorphic viruses.

A majority of polymorphic viruses contain two parts: a "decryption" block and a "cipher-text" block. The "decryption" block, usually the header code, decrypts the "cipher-text" block. The decrypted data contains the infection code. The header tries to hide itself from scanners by inserting jumps, no-ops and register name substitutions in its code. Its sole purpose is to decrypt the second block with the key, and execute its code. Writing the header code is hard and tricky, and virus writers put most of the code in the "cipher-text" block.

Commercial scanners emulate the header code to run the decryption, then examine the decrypted data to see if there is a match with a certain signature. When scanning an executable, the scanners emulate the instructions in the beginning of the file, stop at certain points, and look at virtual memory to see if there is a match with any known signatures. To avoid false positives, most commercial products try to extract signatures that have at least 15 consecutive bytes.

Clam-AV currently does not have an emulation engine. Rather, it uses multi-part signatures with wild-cards. A multi-part signature also attempts to map down multiple versions of a virus that originated from the same source. Though not as comprehensive as the emulation approach, the multi-part signature approach can be effective in some environments [11].

As a result, ClamAV's virus database contains signatures that have wild-cards after as few as two bytes. This property of ClamAV doesn't reflect that of commercial products' databases. Thus, when evaluating Hash-AV, we consider both cases: when multi-part signatures are used, and when they are not.

III. Hash-AV Basic Mechanisms

Hash-AV filters act as a first-pass scan on data to determine if the data needs to go through an exact-match algorithm. Specifically, the algorithm moves a sliding window of β bytes down the input stream. For each β bytes under the window, k hash functions are applied to calculate their hashes. The hash results are then used to probe into a bit array of N bits, which is a bloom filter [4] pre-constructed from the virus signatures.

A. Hash-AV Components

The scanning algorithm uses k independent hash functions on strings of β bytes and constructs a bloom filter from the virus signature set. In order to improve throughput, the sliding window moves down the input data stream four bytes at a time. To make sure that all occurrences of viruses are detected, β bytes starting at the first four offsets in each signature are hashed and inserted into the bloom filter (i.e. the hash results are used as indices to set bits in the bloom filter bit array to 1). For example, assume a virus signature "istanbul-turkey", and β=9. During the initialization of the bloom filter, "istanbul-", "stanbul-t", "tanbul-tu" and "anbul-tur" are hashed and inserted into the bloom filter.

At scan time, for each β bytes under the sliding window, the same k hash functions are applied to them, and the results are probed in the bloom filter. If all bits are 1, the exact-match algorithm (for example, Aho-Corasick) is invoked to see if there is an actual match with a virus signature. For example, for input data "xxistanbul-turkey", the algorithm first checks "xxistanbu" against the filter and does not find a hit. Then, skipping four bytes, it probes "tanbul-tu" in the filter. This time it gets a hit, and it goes into the exact-match algorithm to see if there is a match with the virus. The exact-match algorithm requires access to the previous four bytes of input, which are buffered by the Hash-AV algorithm.

The scanning algorithm applies the hash functions one at a time. After one function, the algorithm checks the bit in the bloom filter. If the bit is 1, it goes on to the next hash function; if not, it immediately slides the window down four bytes and goes onto the next β-byte block. Note that this is different from typical hardware implementations of bloom filters [19], which calculate all hash functions at the same time using parallel hardware and then probe the filter.

Based on our prior experience in using bloom filters [12], $k = 4$ works well. Thus, in Hash-AV, we use four hash functions as well.

Therefore, there are three choices left in setting up Hash-AV:

- Choosing four hash functions;
- Choosing the size of the bloom filter;
- Choosing β;

Below, we use a simple model to briefly analyze the impact of each choice.

B. A Simple Performance Model

Assume that the four hash functions are h_1, h_2, h_3, and h_4, applied in that order. Furthermore, assume that the function h_i can be calculated at c_i MB/s. Let the total number of signatures be N, and the size of the bloom filter be $N * K$ bits. The function h_1 then has a false positive ratio of p_1 in the bloom filter. The ratio p_1 is determined by both the hash function and the bloom filter's expansion factor K.

Similarly, h_2 has a false positive ratio of $p_{2,1}$ in h_1's false positive cases. In other words, $p_{2,1}$ is the conditional probability of false positive under h_2 given that h_1 has a false positive. The ratios $p_{3,2,1}$ and $p_{4,3,2,1}$ are defined similarly.

The performance of the scanning algorithm can be characterized in the following formula:

$$c_1 + p_1 * c_2 + p_1 * p_{2,1} * c_3 + p_1 * p_{2,1} * p_{3,2,1} * c_4 + p_1 * p_{2,1} * p_{3,2,1} * p_{4,3,2,1} * C$$

where C is the cost of the exact string matching algorithm. Clearly, since all the probabilities are between 0 and 1, the hash functions should be ordered from the cheapest (computationally) to the most expensive.

The above formula leads to a number of insights. First, it pays to use very fast, but mediocre hash functions for h_1 and h_2. A hashing function which has 15% error rate but takes five CPU cycles to calculate is a poor choice in other circumstances, but serves very well to our purpose. In fact, these cheap functions help us make the theoretical argument that the Hash-AV scanning algorithm can potentially perform at near memory system throughput.

Second, it's important to choose hash functions that are independent. Completely independent hash functions would have the conditional false positive probabilities the same as the unconditional false positive probabilities. On the other hand, non-independent hash functions tend to have the conditional probabilities close to 1, defeating the purpose of multiple hash functions.

Third, the probabilities are affected by the bloom filter's expansion factor K. Since the cost of the exact string matching algorithm, C, is orders of magnitude higher than the cost of the hash functions, it's important that the bloom filter do not contribute significantly to the false positive ratios. In the sections below, we use experiments to determine the appropriate K.

Finally, there is a lower bound on the probabilities $p_1 * p_{2,1} * p_{3,2,1} * p_{4,3,2,1}$, which is determined by the parameter β. In other words, there is a probability that strings that match the first β characters do not match the full signature. In general, longer βs are better. However, a longer β also means that shorter signatures (those of length $< \beta + 3$) must be handled by a different mechanism. Hence, the choice of β also affects the performance.

C. Evaluation Methodology

In our studies, we focus on windows executable files. The majority of viruses spread through executables. Although a considerable number of macro viruses exist, executable files are still the most common means of infection. Commercial companies focus heavily on executable files, so it is logical to analyze the behavior of Hash-AV on these types of files.

In our analysis of Hash-AV, we run performance tests over an 80 MB sample file. To create this file, we concatenate together widely used Windows executables that are over 3 MB in size, including MS Office executables, messenger programs, third-party software used for scientific and entertainment purposes. In our selection, we pay attention to including only the executable binaries, and avoid setup programs since Hash-AV runs much faster on them.

We evaluate the benefits of Hash-AV for both the current ClamAV database (about 20,000 signatures) and a database containing 120,000 signatures. The signature database for ClamAV is growing very rapidly. Hence, it is essential that Hash-AV scales for large signature databases. To generate more signatures, we wrote a synthetic virus generator that examines the properties of the current ClamAV database, and tries to generate realistic virus signatures.

The generator works as follows. At startup, it reads in the regular and multi-part signatures in Clam-AV's database into different arrays in memory. Then it extracts two pieces of information: the distribution of virus signature lengths, and the percentage of polymorphic patterns in the database. Based on these pieces of data, for each "new" virus, the generator first chooses its length and its type (i.e. regular or polymorphic). For byte i in the new virus, the generator randomly picks an existing signature, and copies its byte i. For each byte index, this algorithm statistically favors the most common byte for that index. In ClamAV's signature database, polymorphic signatures are represented with wild-card ASCII characters in between bytes (* and ??), so this approach automatically generates viruses that are as polymorphic as the ones in the database. A more through approach could consider the distribution of byte couplings in the signature database.

	AMD 1.8 Ghz[2]	P4 2.6 Ghz
L1 instr cache	64 KB	12K mic-ops
L1 data cache	64 KB	8 KB
L2 cache	256 KB	512 KB

We run our experiments on two widely used PC desktops, Athlon XP 2000+ and Pentium-4 2.6GHz. The above table gives a summary of architecture specifics for these machines. Most of the results shown in this paper are from experiments on the AMD Athlon desktop, though the results on the Pentium-4 are very similar.

IV. Hash-AV Signature Filters

To actually construct a Hash-AV filter, we need to determine the variables listed in the above section. Below, we use experiments to determine each component of the filter. Since the choices are intertwined, we first fix β to 9, and study the hash functions and bloom filter sizes. We then return to the choice of β near the end.

A. Selecting Hash Functions

The criteria for the hash functions are that they should be cheap and they should produce relatively random distributions. We first chose a set of well-known fast hash functions from the open source community. The functions usually have 0.2% to 1% collision rate on our sample files, and work well on inputs longer than 4 bytes.

Table 1's first five rows list the hash functions, giving their performance measurements over a sample executable of 80 MB, and the percentage of false positives in the filter. For these tests, β is 9 and the bloom filter size is 256 KB. The throughput measurement contains the cost of hashing each block and the overhead of probing the bloom filter to see if there is a match.

Hash Name	Hash Perf (MB/s)	% of unfiltered input
fnv-32-prime [16]	51.52	6.16%
djb2 [27]	83.73	6.23%
hashlib fast-hash [9]	81.04	6.17%
sdbm [26]	70.66	6.22%
ElfHash [20]	50.45	7.60%
simple mask	233.87	27.35%
simple mod	102.84	13.26%

We were disappointed with these hash functions. Although the functions are said to be very fast, they still perform too slow for our purposes, especially compared to memory-copy speed, which goes at 260MB/s on the desktops.

We then proceeded to add two really fast "hash" functions: "mask" and "mod". Strictly speaking they are not hash functions at all; but they did work in the virus scanning cases. These functions are used as *first level hash functions*. They take the first four bytes, cast it to an integer pointer, and perform their operation on it. They are independent of the size of β, and always act on first four bytes of a block.

The definitions of both functions depend on the size of the bloom filter. For example, for a filter size of 64 Kbits (2^{16}), masking simply **and**s the last sixteen bits of the input block with 1s. That is, with β=5, the block "seatt" is first cast to an integer pointer, changing the data to "seat" for this hash function. Then, a bitwise AND is performed on this data, taking the first two bytes on little-endian machines. As a result, "se" is checked against the bloom filter. For modding, a prime that is known to be good for the range 2^{15}-2^{16} is chosen, and the mod operation is performed on the input with that prime.

Using these two functions as first-level hash functions leads to a big performance gain in Hash-AV. The throughput is approximately doubled in almost all cases.

Hence, Hash-AV contains the following four hash functions: simple mask, simple mod, fast hash from hashlib.c [9] and sdbm [26]. We chose sdbm over djb2 because the correlation between fast hash and djb2 is high.

B. Selecting Bloom Filter Sizes

Traditional bloom filter implementations choose a filter size such that it has about half of the filter populated. Since four variations of the signature are inserted into the filter, for a virus signature database of N signatures, approximately 16*N bits (four hash functions on four variations) are set. Therefore a bloom filter that is 32*N bits, or 4*N bytes, is considered big enough. Based on the conventional wisdom, an 80KB filter should be used for 20,000 signatures, and a 480KB filter would be ideal for 120,000 signatures.

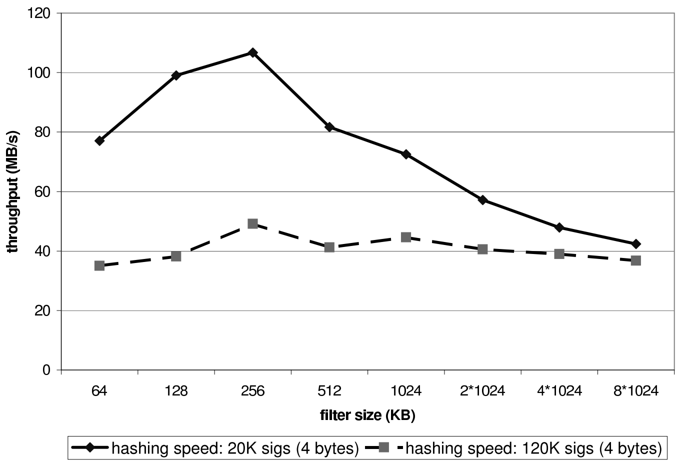

Fig. 2. Performance (in MB/s) of pure hashing for different bloom filter sizes.

In Hash-AV, however, a number of factors impact the choice of the bloom filter size:

- the CPU cache effect: the amount of the filter that fits in the CPU cache, and the cache miss ratio in cases where the filter can't all fit in the cache.
- the mask& mod effect: the mask and mod operations are much faster than other more general hash functions. However, how much of the input these two operations can filter away depends on the sparseness of the bloom filter.
- the false hit ratio: A 3% false hit ratio in a bloom filter might be acceptable if the cost of the false hit is only an order of magnitude greater than the cost of a filter scan. However, it would not be acceptable if the cost of the false hit is two orders higher.

Clearly, the choices are intertwined, and depends on the relative ratio of the cache size and the size of the filter. Below, we use a variety of experiments to examine the factors one by one.

1) Pure Hashing Speed: Given our choice of hash functions, we first look at the pure "hash and lookup" speed under different filter sizes, for databases of both 20,000 and 120,000 signatures. This "pure hashing" speed factors out the effect of false hit ratios, and instead reflects the impact of the CPU cache and the "mask&mod" operations. Figure 2 shows the results on the AMD desktop.

For 20,000 signatures, the results peak at AMD's second-level cache size. This is not surprising since, as long as the filter fits in the cache, large filters lead to more input eliminated by the mask and mod operations,

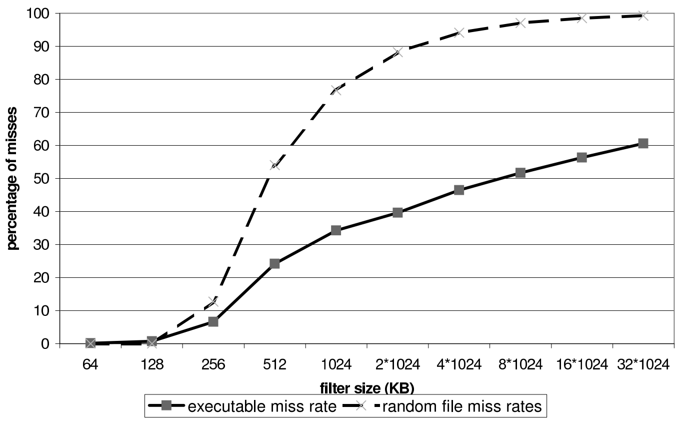

Fig. 3. CPU cache miss rates for the mask operation with different bloom filter sizes.

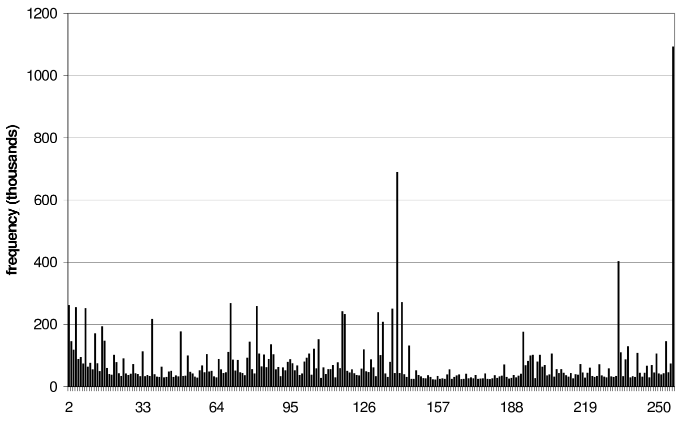

Fig. 4. Distribution of binary data after mask operation (excludes index 0).

but if the filter doesn't fit in the cache, the cache miss latency dominates the throughput.

For 120,000 signatures, the performance difference between the 256KB filter and larger filters is not as dramatic. This has two reasons. First, the mask and mod hash functions generate too many false positives with smaller filters which result in more h_3 and h_4 calculations that dominate run-time. Second, thanks to locality of accesses in the filter, a significant number of signatures can still be verified from the cache even with larger filters.

We used Cachegrind [21], a cache miss profiler, to examine the cache miss ratio of Hash-AV for 120,000 signatures. When executed, Cachegrind runs the target program on a simulated x86 CPU, and reports the number of misses. Cachegrind implements the "inclusive L2 cache" semantics, the standard on Pentium machines. Therefore, the results derived from Cachegrind traces are accurate enough, but are not exact representations of AMD Athlon's cache behavior.

Figure 3 shows the CPU cache miss rates reported by Cachegrind on a 80 MB sample executable and a 50 MB random file. The sample executable is a merge of various programs used in different fields. [3]

This low miss rate on executables, even for very large bloom filters means that the input stream is clustered around certain values. This helps as the filter sizes get bigger since a good portion of heavily accessed data still fits in the CPU cache.

To further analyze the sample executable file, we implemented a program that goes over the input four bytes at a time, and constructs a histogram over 256 values. Figure 4 shows the resulting histogram where the input blocks are also masked. Also, index 0 in the array is removed from the histogram, as this value appears 10 times more than that of the second highest value. Overall, 34.9% of all mask accesses are contained within 5.47% of the bloom filter.

This value however, still doesn't completely explain the low miss rates for very big bloom filters. It is our understanding that most of the words in executable files are closely correlated, and our mask and mod functions preserve this correlation in the bloom filter. Since cache architectures rely on locality of access, both temporal and spatial, the cache miss rate for executables stay at a reasonable level, even for big filter sizes.

In summary, for small number of signatures, the CPU cache size should be chosen as the bloom filter size. For larger number of signatures, the cache effects are not as important and other factors play a bigger role.

2) Scanning Speed and Bloom Filter Sizes: While the above experiments look into the pure hashing and probing speed, the actual performance of Hash-AV also depends on the overhead of the exact match algorithm in Clam-AV. Since the current implementation of Clam-AV does not scale well with the number of signatures, the relative cost of the Clam-AV's exact match algorithm increases as the number of signatures increases. As a result, the best filter sizes for 20,000 signatures are quite different from those for 120,000 signatures.

Figure 5 shows the test results for an Athlon XP 2000+ over a sample executable of 80 MBs. The file is first mapped into memory, and the tests are run on a warm cache to eliminate the disk access overhead.

As the results show, the best filter size varies by the number of signatures and the CPU cache size. On an Athlon XP 2000+, for 20,000 signatures, Hash-AV achieves its peak performance with a 256 KB bloom filter. For 120,000 signatures, however, 4MB filter works well.

The main reason for the difference is the relative cost

[3]The low rate of cache misses on our sample file lead us to repeat the same set of tests on another group of executables merged from /usr/bin, and no significant differences were found.

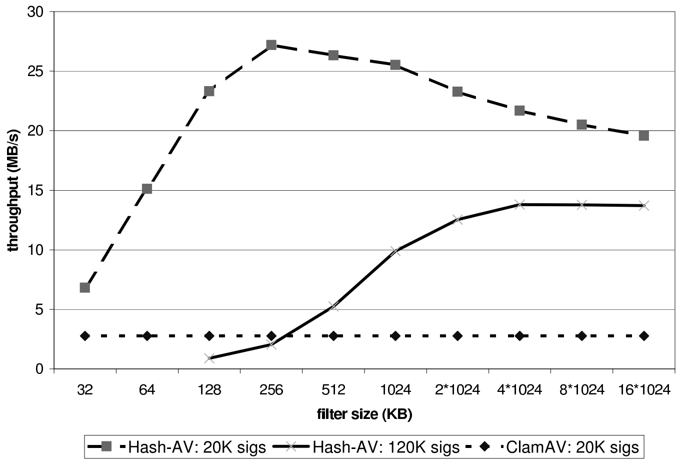

Fig. 5. Performance (in MB/s) of Hash-AV for different bloom filter sizes. β is fixed at 9.

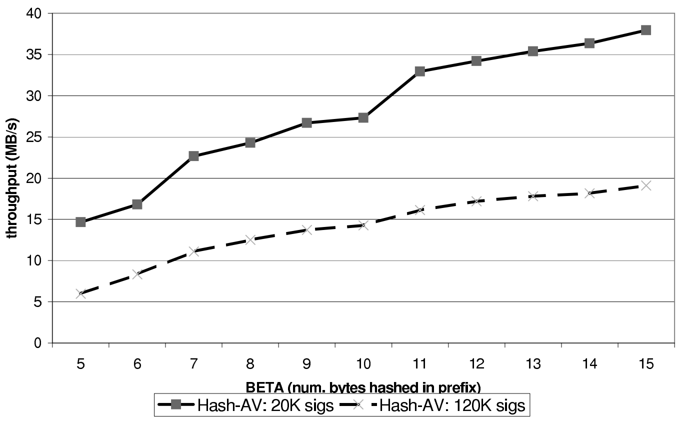

Fig. 6. Performance (in MB/s) of Hash-AV implementation for different βs.

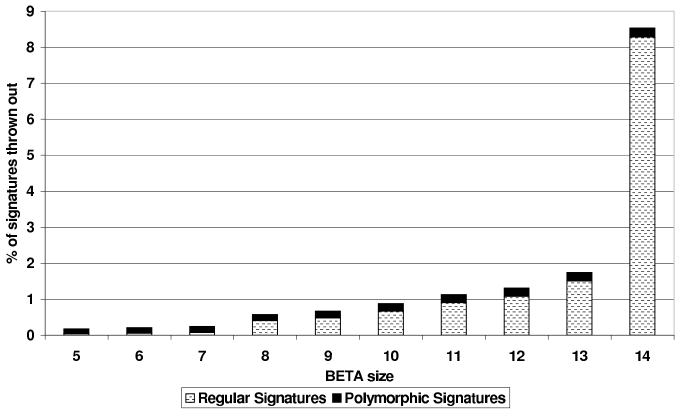

Fig. 7. Percentage of signatures that are thrown out and require special handling for various βs.

of calling ClamAV. As we discussed in Section II-B, the current ClamAV implementation doesn't scale to a large number of signatures. As a result, for 20,000 and 120,000 signatures, the cost of calling ClamAV is 35 and 90 times higher, respectively, than that of pure hashing. Therefore, at 120,000 signatures, it's imperative to get the false hit ratio as low as possible, even if it means that the filter doesn't fit in the cache.

C. Selecting β

The choice of β is mainly affected by the distribution of signature lengths in the signature database. Generally, larger βs are preferred since strings that match the first β bytes in a signature are more likely to match the actual signature. On the other hand, dramatically increasing β has two side-effects. First, the hash functions take more time to compute the result, which in turn slows the algorithm down. Second, with a bigger β, Hash-AV needs to throw out more signatures, and that means there are more signatures that need to be handled separately. Since the second level hash functions can't distinguish input accurately for less than 5 bytes, a lower limit of 5 is set on β.

Figure 6 shows the Hash-AV's throughput for different choices of β with 20,000 and 120,000 signatures on an Athlon XP 2000+. Increasing β leads to a significant performance increase at first; this is because of the high number of false positives that are eliminated. Then the graph converges to a maximum, this is the stage where the trade-off between eliminating more false positives and spending more time in hash functions brings the performance to an equilibrium state.

Figure 7 shows the percentage of virus signatures that cannot be handled by Hash-AV for different sizes of β. These signatures would have to be handled by a string-matching algorithm such as Aho-Corasick, as we discuss in Section V.

Weighting these effects of β, we decided to choose $\beta = 9$ in our algorithms.

D. Helper Tools for Different CPUs

The proper choices of the various parameters in Hash-AV depend heavily on the characteristics of the hardware on the system. Hash-AV tries to push the scanning performance up to a maximum by using both memory and the CPU very efficiently. Computers built today use a wide variety of hardware components, with varying CPU speeds, CPU cache sizes, memory bus bandwidths, and memory access speeds. To help tuning Hash-AV for different systems, we constructed two tools.

The first tool, called the Hash Performance Tester, attempts to determine the best hash functions on a CPU architecture. The functions that we chose are known to perform well on x86 architectures. However, different

CPU architectures have different characteristics. For example, shift operations are slower on Pentium 4 based architectures, and multiplication is slower on Sun based systems. Hence, the tool contains implementations of seven hash functions, and uses a script to compare their speeds on the target system. The tool then recommends the four fastest hash functions.

The second tool chooses bloom filter size and β. The script generates, compiles and executes code for filter sizes between 64 KB - 128 MB (variables may be set by the user). It chooses the filter size that leads to the fastest execution speed. β chooser acts in a similar way, probing β sizes between 5-15 bytes. It picks the smallest value from the equilibrium state in the graph.

These tools helped us determine appropriate parameter settings for Pentium 4 2.6 Ghz. Though the tools have no knowledge of the CPU architecture, they did determine that the appropriate bloom filter size for the Pentium architecture should be 512KB.

V. Performance Benefits of Hash-AV

Hash-AV is not a replacement for current AV scanning algorithms, instead we claim that this technique offers the fastest way to compare the input stream against tens of thousands of signatures. Our implementation of Hash-AV makes calls to ClamAV when the bloom filter indicates that there might be a match.

In this section, we report the comparison of Hash-AV+Clam-AV versus Clam-AV. We did not include any commercial products in this evaluation, because their virus databases are very different from Clam-AV's and it's impossible to do an apples-to-apples comparison.

While we believe ClamAV's handling of polymorphic viruses is not indicative of the state of art commercial virus scanning software, it is important to make a one-to-one comparison with ClamAV, and scan for all the signatures in its database. This can easily be achieved by first running Hash-AV with ClamAV on signatures over $\beta + 3$ bytes, and then running ClamAV again for the left-out signatures. Clever techniques and different optimizations could then be used to increase the performance of this "two-scan" approach.

Figure 8 compares the performance of ClamAV, Hash-AV+Clam-AV but ignoring the shorter signatures, and Hash-AV+Clam-AV doing "two-scans". The tests are run on both the existing Clam-AV database (indicated as the "20K sigs" runs) and the generated 120,000 signature database (indicated as the "120K sigs" runs).

These tests were first performed on a sample executable of size 80 MBs. The second set of tests (Figure 9) was applied on a file of 99 MB containing HTML data that has been crawled off the web, and on a 100 MB random file. We chose to do an HTML benchmark since ClamAV is used at Web proxy gateways and e-mail servers, which tend to see a lot of HTML text. We included a random file in our tests as they are commonly used in benchmarks for multi-pattern string matching algorithms.

In all three cases, Hash-AV performs much better than ClamAV.

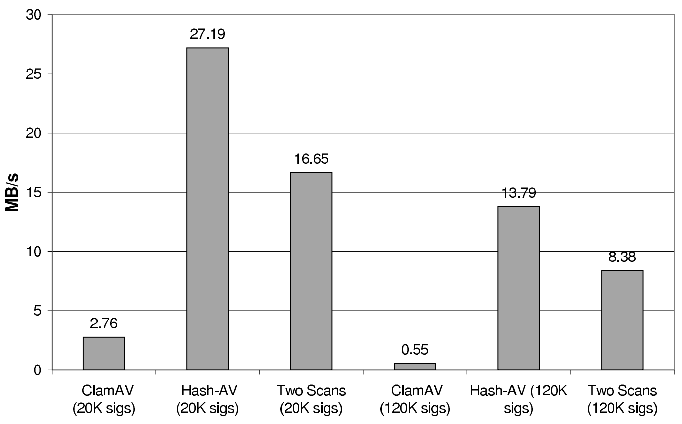

Fig. 8. File scan times for ClamAV, Hash-AV and Two Scans on a sample executable. (one scan with Hash-AV and another with ClamAV for very short signatures Hash-AV cannot handle)

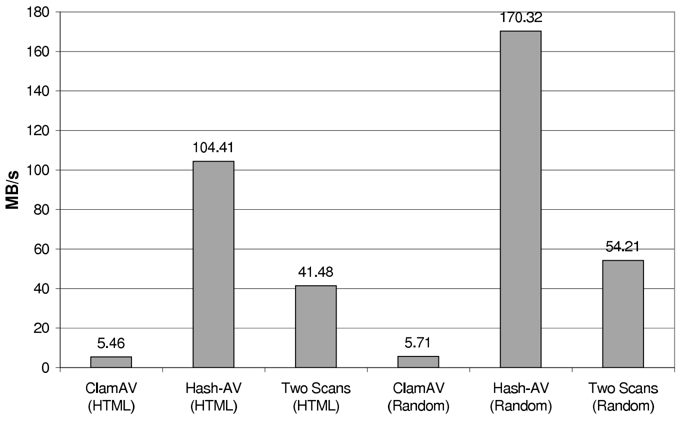

Fig. 9. File scan times for ClamAV, Hash-AV and Two Scans on an HTML and a random file.

Worst-Case Performance of Hash-AV

Today, anti-virus products are mostly used on end nodes, where virus scans are performed at regular intervals (once a week) or when the file is accessed by the end user. Therefore, unlike intrusion detection systems, worst-case performance leads to some latency instead of a server overload. On the other hand, a large number of mail servers provide virus scanning features for their users today, and it may become possible to kill these servers by sending in files that target the weaknesses of the scanner program.

Hash-AV uses hashing for imprecise string matching to achieve very high speeds. It makes calls to ClamAV when all the four hash functions in the bloom filter give positives. Then, in the worst-case, Hash-AV's performance will be ClamAV's execution speed plus the cost of four fast hash functions.

In addition, mail servers could adopt techniques where, if a file seems to match in the bloom filter all the time, the file is marked suspicious and the delivery is delayed until the file is examined at an off-peak time.

VI. Embedding AV Scanning

Modern anti-virus scanners provide an on-access mode where a file is automatically scanned for viruses when it is used. To implement the capability, the scanners typically install their own device drivers that intercept open()/close() and exec() system calls. ClamAV also provides on-access scan capability by using the Dazuko [8] module to intercept system calls in Linux.

An argument can be made that on-access scanning offers more protection than periodic scanning or manual-triggered scanning. However, on-access scanning places an even higher demand on scanner speed. Hash-AV is particularly well suited for on-access scanning mode, since its overhead on "clean" data is quite low.

We implemented two approaches for on-access scanning of Hash-AV on Linux. The first approach uses Dazuko [8] to pass open/close/exec system calls to Hash-AV+Clam-AV. The second approach implements a wrapper around the glibc read(), write(), send() and recv() code to pass the data to Hash-AV+Clam-AV for scanning. Certain applications, like programs that use network interfaces, often need scanning on the fly, that is, when reading from or writing to a socket. The second approach can easily provide scanning for these applications.

Below, we first describe the implementation of the second approach, then present performance results comparing the overhead of all approaches.

A. AV Scanning By Intercepting glibc Calls

Our implementation of read/write/send/recv glibc calls feed the data to the scanner first before calling the actual glibc implementation of these system calls. Currently, the implementation is focused on network stream data, and does not yet handle read/write at arbitrary offsets in a file. Reading/writing at arbitrary offsets are best handled by a whole-file scan at file open/close time. The implementation is packaged as a dynamically linked library libcav.so, and applications can simply link with this library [4] to use the virus-scanning version of the glibc calls.

When used in on-access scanning mode, Hash-AV+Clam-AV keeps state across invocations. After Hash-AV scans the buffer, if the ClamAV exact-match code is invoked and reaches the end of buffer with a partially scanned pattern, that pattern is saved along with the offset. Upon the next invocation, the Hash-AV code first checks if the saved patterns now have a complete match or an extended partial match. If an extended partial match occurs, the pattern is saved around. With large enough reads, the cost of saving and processing the state is negligible.

Concerned about the complexity and reliability of Hash-AV+Clam-AV code, we also implemented a variation where libcav.so is separated from the scanner. A user-level daemon runs the scanner code, and applications linked with libcav.so pass the data to the scanner daemon through shared memory. If the scanner crashes for some reason (for example, a bug, unexpected input, etc.), libcav.so will give up waiting for the scan after a while, and will continue on with normal read/write calls. The cost of this approach is the extra memory copy of the buffer.

B. Performance Results

We compare the performance overhead of the different approaches to embedded scanning by issuing reads of the sample 80MB executable file used in previous tests. Table 3 shows the performance cost associated with each one of these approaches.

Applied Method	Performance (MB/s)
Command Line Scanning	27.19
Intercepting open/close/exec at the kernel	26.85
Wrapping around read/write glibc calls	21.53
Wrapping glibc calls: talking to scanner via shared mem.	20.98

[4]In Linux systems, when a program with a reference to a shared library is compiled, only the name of the function is recorded. At load time, the dynamic loader fixes up the references to these functions. Since C allows multiple definitions of a function, to intercept read/write calls, it suffices to write a new shared library with the same declarations. The calls to glibc's read/write can be made by either using each function's secondary definition in glibc or by making calls to dlsym() when a secondary definition doesn't exist. Finally, by setting the LD_PRELOAD environment variable to libcav.so, we make sure that the loader first looks for the Hash-AV wrapped versions of these read/write functions. If the user decides to stop AV scanning, he just needs to unset LD_PRELOAD [17].

The first row is how fast pure scanning performs on the file. The second row is the throughput of intercepting open() via Dazuko and scanning the file following the call. The throughput change is due to the overhead of intercepting the call and passing it to a user level program. The third row shows the results of reading the executable in chunks of 4 MBs, with each read call issued to libcav.so. The throughput change is due to the repeated calls of the scanner code, and state processing code upon each call. The last row is the approach where libcav.so communicates with the scanner code via shared memory.

In comparison, file reads without scanning can perform over 200MB/s if the file data reside in the buffer cache (i.e. memory-resident), but would be limited to disk throughput if the file data are not memory-resident. While on-access scanning is much slower than memory-resident file reads or writes, it can be combined with file system implementations, for example, avfs [25], to avoid impacting application performance.

On the other hand, the above scanning speed is faster than 100baseT Ethernet, therefore we expect that attaching virus scanning to socket reads and writes would not affect application performances on desktops using 100baseT NICs. We confirm our expectation by linking wget application with libcav.so, and measuring the speed of fetching files between machines.

When using wget to transfer the 80MB merged executable file between two computers connected by a 100 Mbit switch, wget without virus scanning took 8704 milli-seconds, and wget with virus scanning took 10453 milli-seconds. Approximately 470 ms of the overhead comes from the initialization time of ClamAV and HashAV's data structures. The rest of the slow down is mainly due to state saving and restoring between consecutive reads that happen in chunks of 8 KB in wget.

A similar performance test for transferring HTML documents lead to much better results primarily because of the decrease in state saving. The transfer time for a sample 99 MB HTML document was 10021 milli-seconds for raw data transfer, and 10660 milli-seconds for transfer with AV scanning.

In summary, Hash-AV+Clam-AV is well suited for on-access scanning of network transfers in today's 100baseT LAN environment, and can be used as a component in a file system on-access scanning implementation. Implementing the scanner as a daemon to communicate with applications appears to work well.

VII. Related Work

Multi-string pattern matching algorithms is a well-studied topic with applications in many domains [13]. In the networking area, the two prominent applications are IDS (Intrusion Detection Systems) and virus scanners. Recently, several innovations have been proposed for pattern matching in IDS, for example, hardware-based parallel bloom filters [19], and novel compression techniques to reduce memory requirements of IDS and improve hardware implementation performance [15]. However, these studies have not looked into virus scanning applications, which are quite different from IDS systems, as discussed in Section II-B.1.

Our focus on virus scanning applications and software implementation distinguishes our study from the above efforts. Virus scanning applications are commonly host-based, as opposed to IDS systems which are commonly network-based. As a result, software implementations running on generic processors are more appropriate for virus scanners than hardware implementations. Software implementation is different from hardware implementation due to serial applications of hash functions, stringent requirements on the CPU cost of a hash function, and the performance impact of good cache locality. As a result, design choices for software implementation are quite different from those of hardware implementations.

Recently, there have been renewed focus on improving the scalability of Clam-AV by increasing the trie depth [6], [25]. In addition, the Avfs paper [25] provides an excellent study of the issues involved in integrating virus scanners in file system implementation. The techniques described in these studies are complementary to Hash-AV, and the techniques should be combined together to further improve Clam-AV performance.

Because of their importance, there have been constant improvements on multi-string matching algorithms and their variations. Hash-AV is a "booster" technique that is independent of the underlying string matching algorithm, and can be combined with any improved matching algorithm. The benefit of Hash-AV is in quickly determining no-match cases in a CPU cache-friendly manner, and Hash-AV is beneficial to any systems where the no-match cases are the vast majority.

VIII. Conclusions and Future Work

In this paper, we have shown that, through judicious use of the CPU cache, Hash-AV can improve the performance of the open-source virus scanner Clam-AV by an order of magnitude. By using cache-resident bloom filters, Hash-AV determines the vast majority of the "no-match" cases with no main memory accesses. By using cheap hash functions whose computational costs are easily hidden by memory access delays, Hash-AV can potentially scan inputs at a third of memory copy speeds. Since the speed gap between CPU computations and random memory accesses continues to increase, we expect Hash-AV to become more critical for virus scanning performance.

Our study of Hash-AV's design choices show the surprising effectiveness of very cheap functions such as "mask". While in ordinary circumstances "mask" would not be considered as a "hash" function, it can be used as a first hash function in situations where the "hash and lookup" operations are performed serially. Though our choice of the "mask" function might be specific to virus signatures, we believe that in any application where the "no-match" cases are the majority, one can find a very cheap operation that eliminates a significant portion of the input data. This operation can then be used as a first hash function in the Hash-AV technique.

For future work, we plan to improve polymorphic virus detection in Clam-AV. Currently, Clam-AV relies on multi-part signatures for polymorphic viruses, which are not handled by Hash-AV. We plan to look into cache-friendly techniques to speed up matching of those multi-part signatures. We also plan to investigate efficient emulation engines for polymorphic virus detection. For example, one possibility for efficient emulation engine would be through uses of virtual machine technologies.

Finally, we will look into applications of Hash-AV to other large-signature-set pattern matching applications, for example, certain information retrieval and anti-spam applications. The configurations of Hash-AV are likely to be different in those applications, and new capabilities such as handling "do-not-care" characters might be needed as well.

References

[1] AGN. Agn anti-virus test center antivirus scanner tests april 2003. In *http://agn-www.informatik.uni-hamburg.de/vtc/*, Apr. 2003.

[2] A. V. Aho and M. J. Corasick. Effi cient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[3] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10), 1977.

[4] A. Broder and M. Mitzenmacher. Network applications of bloom fi lters: A survey. In *Allerton 2002*, page http://www.eecs.harvard.edu/m̃ichaelm/NEWWORK/papers.html, 2002.

[5] Cisco. Network-based application recognition and distributed network-based application recognition. In *http://www.cisco.com/univercd/cc/td/doc/product/software/ ios122/122newft/122t/122t8/dtnbarad.htm*, 2004.

[6] M. Dounin. Clamav developer forum. In *http://sourceforge.net/mailarchive/forum.php?forum=clamav-devel*, June 2004.

[7] J. S. Gardner. Pc motherboard technology. In *http://www.extremetech.com/article2/0,1558,1148755,00.asp*, June 2001.

[8] H. D. GmbH. Dazuko. In *http://www.dazuko.org*, 2004.

[9] GNU. hashlib.c – functions to manage and access hash tables for bash. In *http://www.opensource.apple.com/darwinsource/10.3/bash-29/bash/hashlib.c*, 1991.

[10] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th Virus Bulletin International Conference*, pages 178–184, 1994.

[11] T. Kojm. Clamav. In *http://www.clamav.net*, 2004.

[12] J. A. L. Fan, P. Cao and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of the 1998 ACM SIGCOMM Conference*, Sept. 1998.

[13] S. Lonardi. Pattern matching pointers. In *http://www.cs.ucr.edu/ stelo/pattern.html*, 2004.

[14] Microsoft. What you should know about download.ject. In *http://www.microsoft.com/security/incident/download_ject.mspx*, June 2004.

[15] B. C. Nathan Tuck, Timothy Sherwood and G. Varghese. Deterministic memory-effi cient string matching algorithms for intrusion detection. In *Proceedings of the 2004 IEEE Infocom Conference*, Mar. 2004.

[16] L. C. Noll. Fowler/noll/vo (fnv) hash. In *http://www.isthe.com/chongo/tech/comp/fnv/*, 2004.

[17] N. S. P. Broadwell and J. Traupman. Fig: A prototype tool for online verifi cation of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and Self-MANaged Systems (SHAMAN)*, June 2002.

[18] M. Roesch. Snort: Network intrusion detection system. In *http://www.snort.org*, 2004.

[19] T. S. Sarang Dharmapurikar, Praveen Krishnamurthy and J. Lockwood. Deep packet inspection using parallel bloom fi lters. In *Proceedings of the 11th Symposium on High Performance Internconnects*, Aug. 2003.

[20] Scalabium. Elf hash algorithm. In *http://www.scalabium.com/faq/dct0136.htm*, 2004.

[21] J. Seward. Cachegrind: A cache miss profi ler. In *http://developer.kde.org/s̃ewardj/docs-2.0.0/cg_main.html*, 2004.

[22] F. Skulason. The evolution of polymorphic viruses. In *http://vx.netlux.org/lib/static/vdat/polyevol.htm*, 2004.

[23] SourceFire. Snort 2.0 high-performance multi-rule inspection engine. In *http://www.sourcefire.com/whitepapers/sf_snort20_HPMRIE.pdf*, Apr. 2004.

[24] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona, 1994.

[25] C. P. W. Y. Miretskiy, A. Das and E. Zadok. Avfs: An on-access anti-virus fi le system. In *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.

[26] O. Yigit. sdbm - substitute dbm. In *http://search.cpan.org/src/NWCLARK/perl-5.8.4/ext/SDBM_File/sdbm*, 1990.

[27] O. Yigit. Hash functions. In *http://www.cs.yorku.ca/ oz/hash.html*, 2004.