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

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

***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 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. Experiments show that Hash-AV improves the performance of the open-source virus scanner Clam-AV by a factor of 2 to 10. The key to Hash-AV's success lies in a set of "bad but cheap" hash functions that are used as initial hashes.**

The speed of Hash-AV makes it well suited for "on-access" virus scanning, providing greater protections to the user. Through intercepting system calls and wrapping glibc libraries, we have implemented an "on-access" version for Hash-AV+Clam-AV. The on-access scanner can examine input data at a throughput of over 200Mb/s, making it suitable for network-based virus scanning.

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. Unfortunately, while the speed of network-based intrusion detection has improved over the years to over 1Gb/s today, the speed of virus scanning has not kept 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 string matching algorithm. The number of virus signatures today is over 100,000 and is growing constantly. Unlike intrusion detection signatures [18], virus signatures cannot be neatly separated into rule sets consisting of a small number of strings. 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 [16].

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.

In this paper, we propose a new virus scanning technique that aims to ride the CPU speed curve. Our solution, called "Hash-AV", combines a set of hash functions and a bloom filter array [2] that fits in the CPU second-level (L2) cache. Hash-AV determines 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 150-200 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.

Using bloom filters to speed up signature matching is not a new idea [3], [19]. For example, research groups have proposed specialized hardware solutions that use parallel bloom filters to scan packets at very high speed [19]. However, the key difference between those hardware solutions and our software solution is that the CPU computation cost of hash functions is not a concern in the former, but a key concern in the latter. If the hash functions are not chosen well, computing the hashes can easily take enough CPU cycles to obliterate the advantage of cache-resident filters.

Hash-AV addresses the problem by using "bad but cheap" hash functions as initial hashes, and relying on serial hash lookups. The "good but expensive" hash functions are only calculated when the cheap initial hashes indicate a match, effectively reducing the CPU computation of the bloom filter probe.

We have applied Hash-AV to Clam-AV [12], the most popular open source anti-virus software. Hash-AV improves Clam-AV's scanning throughput to 29.4 MB/s for executables, 16.6 MB/s for web pages, and 29.5 MB/s for random data, on an Athlon XP 2000+. This represents a speed-up factor of 1.7 to 4.4. The speed improvement doubles as the size of the signature database increases from 20K to 120K. Furthermore, if polymorphic viruses are handled separately using emulation,

[1]Last year's episode of Download.Ject web site infection [15] clearly highlights the importance of scanning viruses at Web proxy gateways.

[2]A bloom filter is a bit array holding results of multiple hash functions on a set of strings. A string can be queried against the filter; the filter generates false positives but never false negatives. Bloom filters are used wherever a compact representation of a set is needed [3].

Clam-AV with Hash-AV can scan executables at 85 MB/s, web pages at 91 MB/s, and random data at 120MB/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.

We have also implemented an "on-access" scanning mode through system call interception and glibc library wrapping. In this mode, Hash-AV+Clam-AV can scan input at over 200Mb/s, making feasible to embed virus scanning in network routers and switches that have application-recognition and reconstruction capabilities [4], [10]. 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 and Clam-AV

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 [11].

a) Clam-AV: Clam-AV is the most widely used open-source anti-virus scanner [12]. It is used by many organizations in their mail servers, and has been incorporated into commercial anti-virus gateway products [10]. As of July 2005, it has a database of over 30,000 viruses, and consists of a core scanner library and various command-line programs. The database includes over 28,000 plain-text strings and over 1,300 strings with wild-card characters embedded. The plain-text strings are for non-polymorphic viruses, and the strings with wild-card characters are for polymorphic viruses.

The current version of Clam-AV uses an optimized version of the Boyer-Moore (BM) algorithm [2] for non-polymorphic signatures, and uses the Aho-Corasick (AC) algorithm [1] for polymorphic ones.

The Boyer-Moore implementation in Clam-AV uses a "shift-table" to reduce the number of times the Boyer-Moore routine is called. At start up, Clam-AV walks over every signature, byte by byte, and hashes the three-byte chunk to initialize a global shift table. Then, at any point in the input stream, Clam-AV can determine if it can skip up to three bytes by performing a quick hash on them. Clam-AV also creates a hash table based on the first three bytes of the signature, and uses this table at run-time when the shift table returns a match. Since this algorithm uses hash functions on all bytes of a signature, it is only applicable for non-polymorphic signatures.

The Aho-Corasick implementation uses a trie to store the automaton generated from the polymorphic signatures. To quickly perform a lookup in this trie, Clam-AV 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. Clam-AV fixes its trie depth to two because its database of polymorphic viruses have signatures with prefixes as short as two bytes.

III. Design of Hash-AV

Hash-AV utilizes the fact that, while virus scanning must be done on network traffic, the vast majority of the data do not contain viruses. Therefore, it aims to determine the no-match cases with *high accuracy*, *minimal main-memory access* and *a small number of CPU instructions*. It achieves the goals by using a filter that fits in CPU caches and acts as a first-pass scan to determine if the data need to go through an exact-match algorithm.

Specifically, Hash-AV 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 [3] pre-constructed from the virus signatures.

A. Basic Mechanisms

Hash-AV constructs a bloom filter from the set of plain-text signatures. The bloom filter is a vector of N bits, initially all set to 0. For each plain-text signature, k hash functions are applied to its first β bytes a, with results $h_1(a), h_2(a), ..., h_k(a)$, all in the range of $1, ..., N$. The bits at positions $h_1(a), h_2(a), ..., h_k(a)$ are then set to 1.

At scanning time, Hash-AV moves over the input data stream one byte at a time. For each β byte block b, the scanning algorithm applies the first hash function, $h_1(b)$, and checks the corresponding bit in the bloom filter. If the bit is 1, it computes the next hash function $h_2(b)$; if not, it immediately goes over to the next byte, and starts applying hash functions over the next β-byte block.

In the case where all k functions have positive bloom filter matches, Hash-AV needs to check for exact match. There are two alternatives here. One is to use Boyer-Moore. Another is to pre-construct a "secondary hash table" using the last hash function h_k, with each entry holding a linked list of signatures which are checked linearly. Hash-AV adopts the latter approach, since the number of signatures in each entry is low.

Several aspects of Hash-AV differentiate it from other approaches. Most commercial scanners use hash-tables to speed up string matching [23], similar to Hash-AV use of the secondary hash table. However, the data structure involved usually does not fit in cache, and the false positive ratio from a single hash function is higher than the bloom filter. Clam-AV uses a cache-resident shift table to reduce the number of times the Boyer-Moore algorithm is called. Unfortunately, since the shift table has to fit in cache, only 3 bytes are used and the resulting false positive ratio is high. In essence, comapred to these schemes, bloom filters are much more compact, and the use of multiple hash functions results in much lower false positives. Other researchers have proposed using hardware bloom filters to perform high-speed network intrusion detection [19]. However, in those designs, all hash functions are calculated at the same time using parallel ASIC hardware. Hash-AV applies hash functions *serially*, in an effort to reduce the number of CPU instructions consumed.

Based on our prior experience in using bloom filters [13], $k = 4$ works 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 M, and the size of the bloom filter be $M * K$ bits. The function h_1 then has a false positive probability of p_1 in the bloom filter. The probability p_1 is determined by both the hash function and the bloom filter's expansion factor K.

Similarly, h_2 has a false positive probability 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 modeled using the above parameters. Note that h_2 is called when h_1 hits in the bloom filter (i.e. h_1's bit is 1), h_3 is called when both h_1 and h_2 hit in the bloom filter, and h_4 is called when all three previous hash functions hit in the bloom filter. Thus, the throughput of the scanning algorithm is:

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

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, might be one or two 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

We evaluate the benefits of Hash-AV for both the current Clam-AV database (about 30,000 signatures) and a database containing 120,000 signatures. The signature database for Clam-AV 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 Clam-AV database, and tries to generate realistic virus signatures.

The generator works as follows. At startup, it reads in the non-polymorphic and polymorphic 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. non-polymorphic 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. Since Clam-AV's polymorphic signatures only use wild-card ASCII characters

* and ?? in between bytes (with * matching any string and ?? matching any single character), this approach generates viruses that are as polymorphic as the ones in the database.

For most experiments, the sample file is an 120MB file created by concatenating 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. We focus on windows executable files since the majority of viruses spread through executables and commercial scanners focus heavily on executable files. In our selection, we pay attention to including only the executable binaries, and avoid setup programs since Hash-AV runs much faster on them.

Our experiments are run on an Athlon 64 3200+ PC, with 2.0 Ghz CPU, 128KB L1, and 512KB L2 cache. We have also repeated the experiments on an Athlon XP 2000+ and a Pentium-4 2.6Ghz PC, and found matching results.

IV. Hash-AV Components

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 7, 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 list the hash functions, giving their performance measurements over a sample executable of 120 MB, and the percentage of false positives in the filter. For these tests, β is 7 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 [17]	27.44	1.75%
djb2 [26]	43.79	1.77%
hashlib fast-hash [9]	46.22	1.76%
sdbm [25]	36.25	1.75%
ElfHash [20]	25.30	7.46%

We were disappointed with these hash functions. They might be considered fast compared to other hash functions, but not compared to memory-copy speed, which goes at 260MB/s on the desktops.

We then tried two really fast "hash" functions: "mask" and "xor+shift". "Mask" takes the first four bytes, casts them to an integer, and chooses the lowest $log_2(N)$ bits, where N is the size of the bloom filter. "Xor+shift" takes the first six bytes, casts bytes 0-3 into an integer, and xors this word with 0 to get the first hash value. It then repeats the same operation two more times, for bytes 1-4 and 2-5, always xoring with the previous hash value to get the next one. It then picks the lower $log_2(N)$ bits of the final integer to check against the filter. "Mask" and "xor+shift" can be computed at throughputs of 160 MB/s and 120 MB/s consecutively.

On virus signatures, "mask" and "xor+shift" can filter away 88% and 96% of the input bytes. As standalone hash functions their false positives would be too high. However, used as *first level hash functions*, they can effectively cut down the number of times that the "good" hash functions are calculated by an order of magnitude.

Hence, Hash-AV contains the following four hash functions: mask, xor+shift, fast hash from hashlib.c [9] and sdbm [25]. Sdbm is chosen over djb2 because the correlation between fast hash and djb2 is high.

B. Selecting Bloom Filter Sizes

Traditional bloom filter implementations choose filter sizes such that half of the filter bits are 1. In Hash-AV, however, a number of factors impact the choice of the bloom filter size:

- the CPU cache effect: the portion of the filter that fits in the CPU cache, and the cache miss ratio in cases when the filter cannot all fit in the cache.
- the initial hash function effect: the initial hash functions are much faster than the latter ones. However, how much of the input data that the initial hashes filter away depends on the sparsity 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 higher than the cost of a filter probe. However, it would not be acceptable if the cost of the false hit is two orders of magnitude higher.

Clearly, the choices are intertwined, and depend 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 30,000 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 cheap hash functions. Figure 1 shows the results on the AMD desktop.

For 30,000 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 "mask" and "xor+shift", but if the filter does not fit in the cache, the cache miss latency dominates the throughput.

For 120,000 signatures, the performance difference between the 512KB filter and larger filters is not as dramatic. This has two reasons. First, in small filters, the initial hash functions generate too many false positives, leading to more hash calculations that dominate the run-time performance. Second,

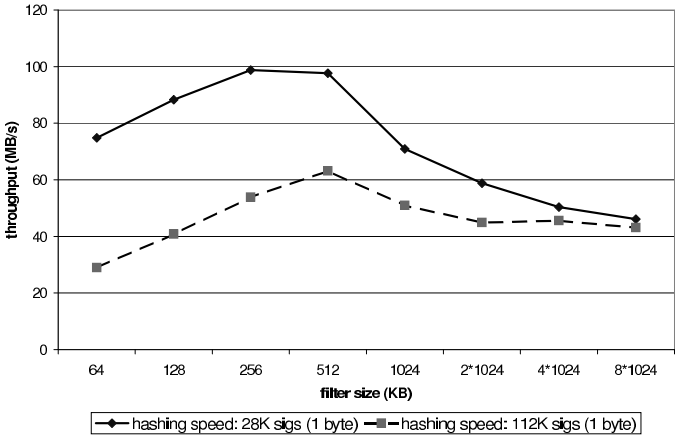

Fig. 1. Performance (in MB/s) of pure hashing for different bloom filter sizes.

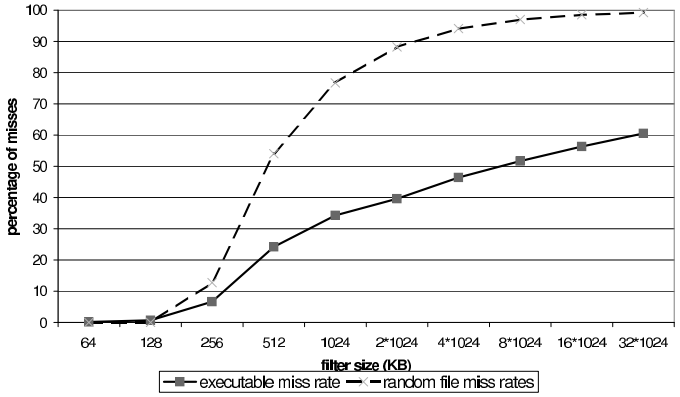

Fig. 2. CPU cache miss rates for the mask operation with different bloom filter sizes.

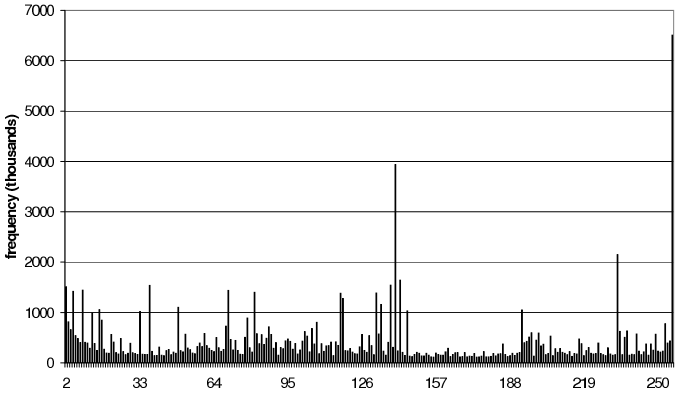

Fig. 3. Distribution of binary data after mask operation (excludes index 0).

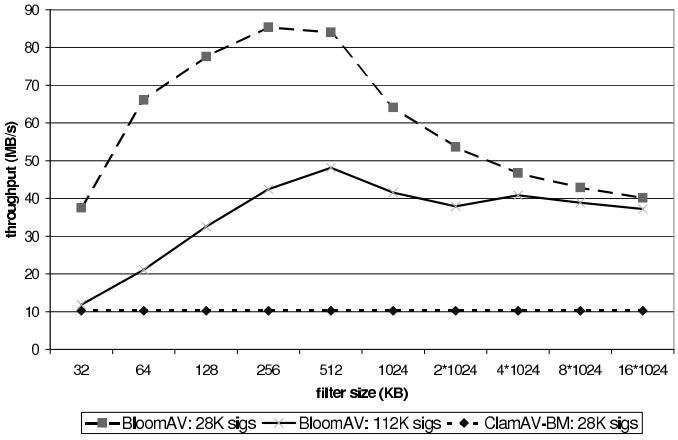

Fig. 4. Performance (in MB/s) of Hash-AV for different bloom filter sizes. β is fixed at 9.

thanks to locality of accesses in the filter, even for large filters, a significant number of signatures can be verified from the CPU cache.

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 2 shows the CPU cache miss rates reported by Cachegrind on the 120 MB sample file and a 100 MB random file. This low miss rate on executables, even for very large bloom filters, means that the input stream is clustered around certain values. 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 of the values of the mask operation. Figure 3 shows the resulting histogram of 256 bins. Also, index 0 in the array is removed from the histogram, as this value appears 3 times more than that of the second highest value. Overall, 37.9% of all mask accesses are contained within 5.47% of the bloom filter.

In addition to the above skewed distribution, we found that most of the words in executable files are closely correlated, and our mask function preserves this correlation in the bloom filter. Since cache architectures rely on locality of access, both temporal and spatial, the cache miss rate for executables can stay at a reasonable level, even for big filter sizes. This helps the performance of large filters since a good portion of heavily accessed data still fits in the CPU cache.

In summary, for small number of signatures, the CPU cache size should be chosen as the bloom filter size. For large number of signatures, 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. Figure 4 shows the speed of Hash-AV on the Athlon 64 3200+ desktop over the sample 120MB executable file. 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 is mostly determined by the CPU L2 cache size. For both 30,000 and 120,000 signatures, Hash-AV achieves its best performance with 512KB (i.e. the L2 cache size) bloom filter. The impact of larger filters is less pronounced on the 120,000 signatures, since larger filters

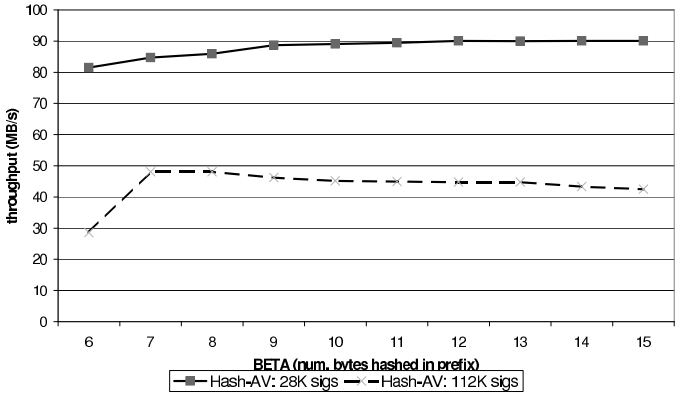

Fig. 5. Performance (in MB/s) of Hash-AV implementation for different βs.

reduce the false positive ratio rapidly for this set of signatures.

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 has to leave out short signatures. The short signatures are then handled together with the polymorphic ones in a separate scan using the Aho-Corasick algorithm. The "xor+shift" function is designed to operate on six bytes of data, and other hash functions can't distinguish input accurately for small input streams. Therefore, a lower limit of six is set on β.

Figure 5 shows the Hash-AV's throughput for different choices of β with 30,000 and 120,000 signatures on an Athlon 64 3200+. 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.

Weighting these effects of β, we decided to choose $\beta = 7$ 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 6-15 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

We compare the throughput of Hash-AV with that of Clam-AV, using both the 30,000 signature database and the 120K signature database. The current implementation of Hash-AV focuses on improving the scanning speed for plain-text signatures. It uses the same Aho-Corasick (AC) implementation as Clam-AV, hence is subject to performance reduction caused by that module[3]. To separate the effect of the AC module, we compare the scanners in two modes, one scanning for plain-text signatures only (i.e. Hash-AV vs. ClamAV-BM), and the other scanning for both plain-text signatures and polymorphic signatures (i.e. Hash-AV+AC vs. ClamAV-BM+AC).

Commercial products are not included in this evaluation for two reasons. First, commercial virus databases are quite different from Clam-AV's database. Second, a number of techniques used by the commercial scanners are not yet implemented in Clam-AV. For example, file-type or virus-type specific information can be used to scan only parts of the file, emulation engines can be used to handle polymorphic viruses, etc. Hash-AV is complimentary to these techniques and can be combined with them to improve the scanning speed further. However, these techniques does make it difficult to compare the performance of commercial scanners with that of Clam-AV.

In the experiments, three different types of inputs are used: the 120MB sample executable file as described in Section III-C, a file of 99 MB containing HTML data crawled from the web, and a 100 MB random file. We choose to use an HTML file since Clam-AV is often used at Web proxy gateways and e-mail servers, which tend to see a lot of HTML text. We include a random file in our tests as they are commonly used in benchmarks for multi-pattern string matching algorithms.

[3]The principles in Hash-AV can be used to speed up the scanning of polymorphic signatures. We have not yet explored this venue because an emulation engine maybe a more effective defense against these viruses.

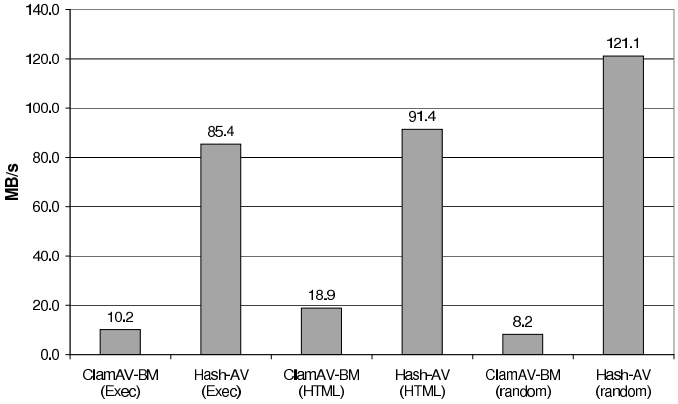

Fig. 6. Throughput of Hash-AV and ClamAV-BM on different types of input files, using the set of plain-text signatures in the existing ClamAV virus database (about 28,000 signatures).

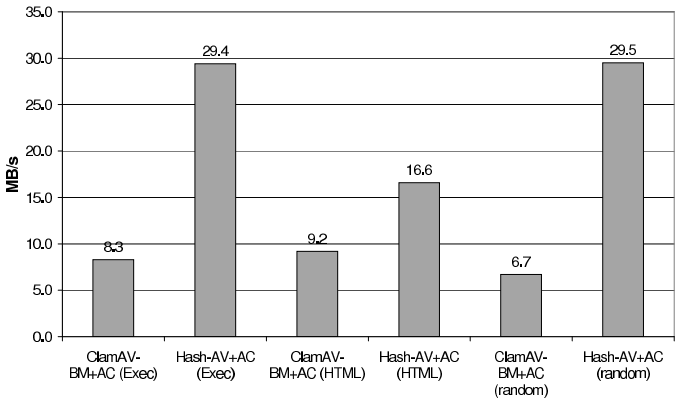

Fig. 7. File scan throughput of Hash-AV+AC and ClamAV-BM+AC, using the existing Clam-AV virus database (about 28,000 plain-text signatures and 2,000 polymorphic signatures).

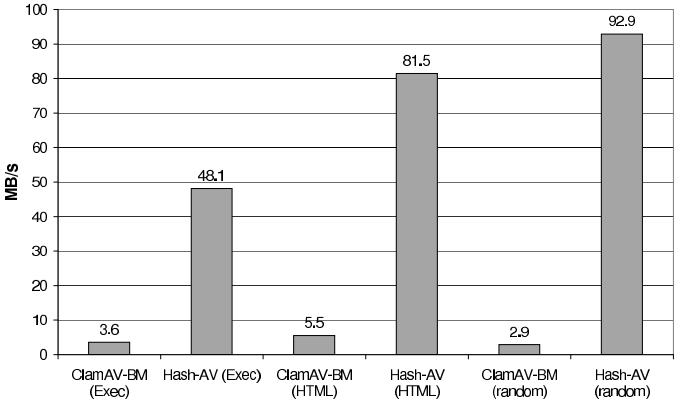

Fig. 8. Throughput of Hash-AV and ClamAV-BM scanning for 112,000 plain-text signatures. (The method of signature generation is described in Section III-C.)

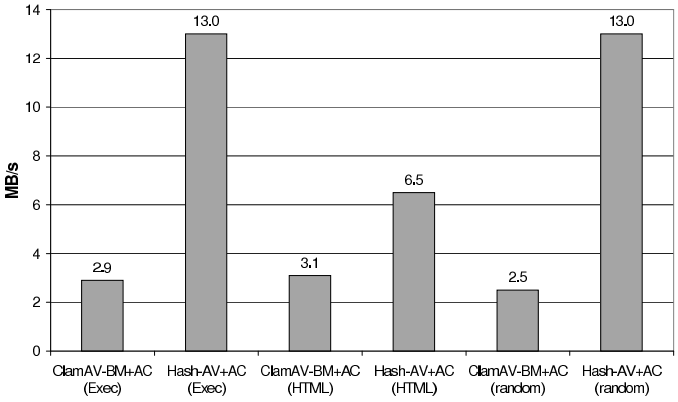

Fig. 9. Throughput of Hash-AV and ClamAV-BM scanning for 120,000 signatures, including both plain-text and polymorphic signatures. (The method of signature generation is described in Section III-C.)

Figure 6 shows the throughput of Hash-AV and ClamAV-BM scanning for 28,000 plain-text signatures from the current virus database of ClamAV. Figure 7 shows the throughput of Hash-AV+AC and ClamAV-BM+AC, scanning for all signatures in the current ClamAV database. Figure 6 demonstrates the speedup offered by the approaches in Hash-AV; the scanning speed is improved by a factor of 8 to 15. Even though scanning for polymorphic signatures (i.e. the AC module) caused a slowdown, Figure 7 shows that Hash-AV still outperforms Clam-AV by a factor of 1.8 to 4.4 on the full set of signatures.

To examine the scalability of Hash-AV and Clam-AV, we repeated the experiments on the 120,000 signature set. Figure 8 and Figure 9 show the results. Hash-AV scales better than Clam-AV; as the database size increases, the throughput reduction in Hash-AV is much less than that of Clam-AV, particularly on executable inputs and random inputs.

a) Worst-Case Performance of Hash-AV: All string-matching algorithms have varying performances depending upon the input. Hash-AV is no exception. An attacker could construct inputs so that the scanning speed of Hash-AV is reduced to that of exact string matching algorithm. This usually does not present a problem for desktops, but would be a problem for application gateways such as mail servers and web proxies. However, application gateways typically have other means to mitigate these attacks. For example, they can scan different mailboxes or different HTTP streams in separate processes, and use CPU scheduling to ensure that processes that are scanning slowly do not impact other processes.

VI. Embedding AV Scanning

Due to its high performance, Hash-AV is well suited for "on-access" virus scanning. In the "on-access" mode, a file is automatically scanned whenever it is used, and data from the network are scanned when they are received by the application (i.e. socket reads). "On-access" virus scanning typically provides greater protection to the user.

We have 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. The second approach implements a wrapper around the glibc read(), write(), send() and recv() code to pass the data to Hash-AV for scanning. The implementation is packaged as a dynamically linked library libcav.so, and applications link with this library instead of the regular libc.so to enable virus-scanning. The first approach is suited for file accesses, and the second approach

is suited for socket accesses.

When used in on-access scanning mode, Hash-AV keeps state across invocations. After Hash-AV scans the buffer, if the 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.

We compare the performance overhead of the different approaches to embedded scanning by issuing reads of the sample 120MB executable file used in previous tests. The existing Clam-AV virus database is used in these experiments. The table below 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 via Dazuko	26.85
Wrapping glibc calls:	20.98

The first row is how fast Hash-AV+AC 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 an extra memory copy in our implementation.

Note that in on-access mode, the throughput of Hash-AV+AC is nearly 200Mb/s. Since most desktops' connections to the Internet are well below 200Mb/s, attaching virus scanning to socket reads and writes would not affect applications receiving data from the Internet.

We confirm this expectation by linking the *wget* application with libcav.so, and measuring the speed of fetching files between machines. When transferring the 120MB executable file between two computers connected by a 100 baseT switch, wget without virus scanning takes 8704 milli-seconds, and wget with virus scanning takes 10453 milli-seconds. Approximately 470 ms of the overhead comes from the initialization time of Clam-AV 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 shows better results primarily because of the decrease in partial matches and associated state saving. The transfer time for a sample 99 MB HTML document is 10021 milli-seconds for raw data transfer, and 10660 milli-seconds for transfer with AV scanning.

In summary, Hash-AV+AC is well suited for on-access scanning of network transfers for most desktops, and can be used as a component in a file system on-access scanning implementation, for example, avfs [24].

VII. Related Work

Multi-string pattern matching algorithms is a well-studied topic with applications in many domains [14]. 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 [16]. However, these studies have not looked into virus scanning applications, which are quite different from IDS systems [6].

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 [5], [24]. In addition, the Avfs paper [24] 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 significantly improve the performance of the open-source virus scanner Clam-AV. 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 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.

Key to Hash-AV's success is the use of very cheap functions such as "mask" and "xor+shift". While in ordinary

circumstances they are not good hash functions, they are very effective as initial functions in a serial application of a set of hash functions. Though our particular choices might be considered 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.

For future work, we plan to improve polymorphic virus detection. Currently, both Clam-AV and Hash-AV rely on multi-part signatures for polymorphic viruses, which are handled by the Aho-Corasick algorithm. 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 plan to apply 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] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

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

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

[4] 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.

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

[6] O. Erdogan and P. Cao. Hash-av: Fast virus signature scanning by cache-resident filters. In *http://crypto.stanford.edu/c̃ao/hash-av/*, 2005.

[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] W. Inc. Watchguard announces gateway antivirus for e-mail for the firebox x security appliance. In *http://www.watchguard.com/press/releases/wg294.asp*, Nov. 2004.

[11] 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.

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

[13] 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.

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

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

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

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

[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 filters. 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 profiler. In *http://developer.kde.org/šewardj/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] P. Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2004.

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

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

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