Approximate Fingerprinting to Accelerate Pattern Matching

Ramaswamy
Ramaswamy
Univ. of Massachusetts
rramaswa@ecs.umass.edu

Lukas Kencl
Intel Research
Cambridge, UK
lukas.kencl@intel.com

Gianluca Iannaccone
Intel Research
Cambridge, UK
gianluca.iannaccone@intel.com

ABSTRACT

Pattern matching and analysis over network data streams is increasingly becoming an essential primitive of network monitoring systems. It is a fundamental part of most intrusion detection systems, worm detecting algorithms and many other anomaly detection mechanisms. It is a processing-intensive task, usually requiring to search for a large number of patterns simultaneously.

We propose the technique of "approximate fingerprinting" to reduce the memory demands and significantly accelerate the pattern matching process. The method computes fingerprints of prefixes of the patterns and matches them against the input stream. It acts as a generic preprocessor to a standard pattern matching engine by "clearing" a large fraction of the input that would not match any of the patterns. The main contribution is the "approximate" characteristic of the fingerprint, which allows to slide the fingerprinting window through the packet at a faster rate, while maintaining a small memory footprint and low number of false positives. An improvement over a Bloom filter solution, a fingerprint can indicate which patterns are the candidate matches. We validate our technique by presenting the performance gain for the popular Snort intrusion detection system with the preprocessor in place.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network monitoring; I.5.2 [**Design Methodology**]: Classifier design and evaluation; C.2.0 [**General**]: Security and protection

General Terms

Performance, Design, Measurement

Keywords

Pattern matching, intrusion detection, fingerprint, deep packet inspection

1. INTRODUCTION

Pattern matching is an extremely common function implemented in many monitoring systems to support a wide range of applications. Most intrusion, virus or worm detection systems [13, 11] operate by comparing an incoming packet stream against a database of patterns known to be present in exploits or worms. These systems have become extremely popular and are being deployed in various operating environments (from end-hosts, to small switches and routers to passive monitoring devices inside the network).

The large-scale deployment is accompanied by rapid growth of the database of content strings. However, pattern matching is a very processing intensive task since patterns have to be compared byte by byte to the input string. Any method to accelerate pattern matching can operate on two variables: the per-pattern processing cost and the per-input-string processing cost. The former has been the subject of extensive past research and several algorithms have been proposed that can compare a large number of patterns at once. They return precise matches at a processing cost independent of the number of patterns searched [1, 4, 21]. Reducing the per-input-string processing cost is instead inherently harder. Exact matches have a cost linear with the length of the string given that there is no alternative to comparing the string against the patterns byte by byte. Research in this area has focused on optimizing the existing multi-pattern matching algorithms, mainly to reduce the memory footprint of those methods [20].

In this paper we propose "approximate fingerprinting" to pre-process the input string and return approximate matches. The method computes a fingerprint over a sliding window of bytes in the input string and compares it against a database of fingerprints derived from the patterns that are searched. The *approximation* comes from the fact that at each step the window slides by more than one byte, reducing this way the number of memory accesses per-packet but introducing additional errors in the matching. Our method returns a subset of the original input strings that are good candidates for a matching pattern and guarantees that this subset will contain *all* the input strings that do match at least one of the patterns. In addition, the candidate packets may carry the information *which* patterns can actually match. The candidates can then be processed by one of the precise pattern matching algorithm proposed in the literature to eliminate any false positives.

The performance gain with the approximate fingerprinting comes from the reduction in the number of strings the complex matching algorithm has to process. Our design ex-

hibits the following properties: (*i*) fast to compute, (*ii*) small memory footprint to be easily implemented in hardware, and (*iii*) small false positive rate and zero false negatives (i.e., no strings that would match a pattern are ever missed). These are desirable properties for implementation on a network system such as a switch or router. The processing environment on such systems is resource limited in terms of both processing and memory with real-time constraints.

The gain can be significant in all cases where the occurrence of strings matching any of the patterns is a relatively rare event. Our method would "clear" a vast majority of the strings, forwarding just a small subset to the more complex precise matching algorithm. In those cases where most strings do match a pattern (e.g., worm infection for network intrusion detection systems) our method introduces a very limited overhead, which can be eliminated by a hardware implementation. Furthermore, the subsequent pattern matching method may still well benefit by exploiting the hints about the location of the candidate matches and the relevant rules as indicated by the preprocessor.

We have implemented a prototype of our approximate fingerprinting method as a pre-processor for the Snort network intrusion detection platform. Snort presents a perfect application to our method given the large database of content strings that it searches in the packet stream. The pre-processor is placed inline with other existing pre-processing that Snort performs and reduces the amount of packets that need to be searched by exact pattern matching algorithms in the detection engine of Snort.

We evaluate our implementation using the current database of Snort rules and packet traces with full payloads collected on a residential university access network [9]. We show that our method has an extremely small footprint (in the order of tens of Kbytes for thousands of patterns) and can reduce the load (in terms of processed packets) for the precise matching algorithm to less than 14%.

2. RELATED WORK

String pattern matching is a classical problem in computer science and there exists a large body of work in the literature. The Aho-Corasick [1] and Commentz-Walter [4] algorithm belong to the most commonly used multi-pattern matching algorithms. Both however present an extremely large memory footprint and in their original form are suitable only for software implementation with obvious throughput limitations.

For these reasons several modifications to the Aho-Corasick algorithm have been recently proposed to allow a hardware implementation using FPGAs [18, 19, 6] or network processors [3]. Other proposals attempt to reduce the memory footprint of Aho-Corasick assuming the availability of extremely large memory bus to avoid the performance limitations [20]. Our work complements these efforts by providing a method with an extremely small memory footprint that is able to reduce the number of packets that need to go through the full Aho-Corasick pattern matching.

The idea of filtering packets that are not going to match any of the intrusion detection rules has been explored in the literature using Bloom filters [5, 2, 16]. The limitations of these method lie mainly on the large memory footprint required by the Bloom filter (particularly for a large number of keys and a low false positive rate). This severely limits the applicability of the methods given that a precise pat-

tern matching algorithm still needs to run to eliminate the false positives produced by the Bloom filter. Furthermore, a Bloom filter provides a boolean answer to the suspicious set-membership question but no information on *which* patterns are likely to match. Our proposal is based instead on string fingerprints. Originally introduced by Rabin [12], they have been used to find sets of files that are similar [8], to reduce network traffic by eliminating identical packets [17], and more recently to detect network worm activity by finding commonly occurring patterns in the packet payloads [7, 14, 10].

Sommer and Paxson [15] augment signature based intrusion detection systems such as Snort with additional context. Their goal is to reduce the false positive rate and improve signature matching performance. A direct comparison with their work is not possible due to considerable differences in Snort versions/rulesets and testing platforms.

3. DESIGN CONCEPTS

In the design of any approximate pattern matching method the objective is to find the right balance between the computational complexity, that is mainly in terms of instructions and memory access per packet, the memory footprint required by the data structure and consequently the lookup times, and the false positive rate, i.e. the patterns that are matched erroneously. In the following we address each of this aspect separately and illustrate the key design concepts behind our proposal.

Approximate Fingerprinting. The basic idea is to quickly examine packet payloads and probabilistically determine what patterns, if any, the payload contains. To this end, we utilize Rabin's fingerprints. A fingerprint is a short tag for a longer object (pattern or string). If two fingerprints are different, then the objects that the fingerprints represent are certainly different. The probability that two different objects map to the same fingerprint are small. A Rabin fingerprint is the polynomial representation of the object modulo a predetermined irreducible polynomial.

We build a fingerprint table that contains the Rabin's fingerprint of the first w bytes of each pattern we are trying to match. Fingerprints are then computed over a sliding window of size w bytes on the packet payload and used to index the fingerprint table (see Figure 1a). If a match is found in the table, the packet is annotated to indicate the possible matching patterns. Otherwise, given that no match is found, the packet can entirely bypass the precise pattern matching module[1].

Reducing memory accesses. Sliding forward the window by one byte at each step results in a number of accesses to the fingerprint table almost linear to the packet size. More precisely, $b - w + 1$ table lookup operations will be required for a packet of size b bytes, and fingerprint window size w bytes. This number of memory accesses increases the processing complexity and slows down the approximate fingerprinting step to pre-process the packet stream.

One way to reduce the number of table lookups is to slide the window forward by s bytes ($s > 1$) instead of 1 byte. By

---

[1]Note that bypassing a packet implies that rules are defined on a per packet basis (e.g., independent of the connection status), and that patterns do not traverse packet boundaries. We will address these aspects later in Section 5.
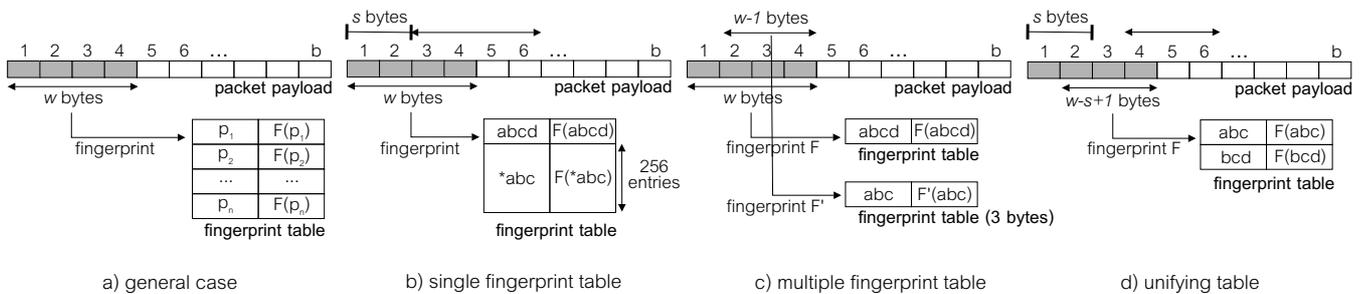
Figure 1: Approximate Fingerprinting

a) general case b) single fingerprint table c) multiple fingerprint table d) unifying table

doing so, we reduce the number of table indexing operations by a factor of s. The number of such operations, L, is given by:

$$L = \left\lceil \frac{b - w + 1}{s} \right\rceil \qquad (1)$$

The problem with sliding the window forward by s bytes is that we would not be able to detect those patterns that cross window boundaries. A solution is then to add extra entries to the fingerprint table that will report all matches which contain any sequence of characters $(s-1)$ byte long followed by the first $(w - s + 1)$ characters of a pattern as a match. This solution will increase the number of false positives for all those patterns that do not cross the window boundary. Indeed, we are trading an *increase* in the fingerprint table size and an *increase* in the false positive rate for a *decrease* in the number of fingerprint table lookups.

Figure 1b shows a detailed example with a window size of 4 bytes and the pattern `abcd`. The window is slid forward by 2 bytes at each step. At the first step, the fingerprint is computed on the substring from byte 1 through byte 4. On the next iteration, the window would move forward by 2 bytes and a new fingerprint would be computed on the substring between byte 3 and byte 6. We will detect the presence of *abcd* in the payload if it starts at byte position 1 or if it starts at byte position 3. We will miss the presence of *abcd* in the packet if it is offset by 1 byte at the beginning. For this reason, we add extra entries to the fingerprint table denoted by *abc* where the * represents a single character from the alphabet (i.e., we add 256 entries to the fingerprint table). The amount of memory required for the pattern table is given by

$$n_p \cdot f \cdot (1 + 256^{s-1}), \qquad (2)$$

where n_p is the total number of original patterns, f is the size of fingerprint in bytes and s is the window step size in bytes.

Reducing the memory footprint. Using a single pattern table as shown in Figure 1b is not very memory efficient. According to (2), the amount of memory required to store all the patterns in the table increases exponentially with the window step size, s. Therefore, this approach is impractical for window step sizes greater than 2 bytes.

One solution to this problem is to use *multiple tables* (see Figure 1c) to store fingerprints of substrings of varying lengths starting from w bytes down to $w - s$ bytes. The memory reduction comes from the fact that all the extra patterns we introduced previously differed in the first $s - 1$ characters, but were the same for the remaining $w - s + 1$ characters. Thus, the extra 256^{s-1} patterns stored in the single table case described above can be represented by a single entry in a separate fingerprint tables that contains shorter patterns. The memory growth is now linear with the number of patterns and the window step size and therefore scales well with the increase of the window step size. The number of extra tables needed is the same as the window step size s. Note that since Rabin's fingerprints are computed incrementally, it is trivial to obtain the fingerprint of the last $w - s + 1$ bytes when computing a fingerprint for w bytes.

Consider the same example described previously: window sizes w of 4 bytes, the window step size s is 2 bytes and we look for the pattern *abcd* In the single table case, we would need an extra 256 entries of the form *abc* to be stored in the pattern table. Now, these are represented in another pattern table by just the common portion of the string *abc*.

The steps proceed as follows. A first fingerprint, F, is computed over the first window and a second fingerprint, F', is computed over the shorter window. The first fingerprint table is looked up with F. If a match is found, then *abcd* is present in the first window (there is a small chance of a false positive due to collisions in the fingerprint space). If F is not found in the first fingerprint table, the second fingerprint table is looked up with F'. If a match is found, it is reported as a possible hit. This example is illustrated in Figure 1c. The amount of memory required for these pattern tables is $n_p \cdot f \cdot s$, where n_p is the total number of original patterns, f is the size of fingerprint in bytes and s is the window step size in bytes.

While the multi-table approach is well suited for a hardware implementation, where multiple small tables can easily be accessed in parallel, in a software implementation the multiple sequential memory accesses would significantly lower performance. Thus, alternatively, the multiple tables can be compressed into a single, *unifying table* (see Figure 1d), by storing only *suffixes* of length $w - s + 1$ bytes (the shortest sub-window) of all the entries in the multiple fingerprint tables. In our practical example, to detect *abcd* would thus mean storing the fingerprints F' of strings *abc* (from the original "$w - s + 1$" table) and *bcd* (suffix of entry *abcd* in the $w - s + 2$ table, as in Figures 1c and 1d). This solution further reduces the table size and requires only one fingerprint computation and one lookup, but increases the possible number of false positive matches and fingerprint collisions, due to only a portion of the string being matched and possible overlaps of these shorter substrings.

Total rules	2836
Non-pattern rules	146
Pattern containing rules	2690
`content` containing rules	1930

Table 1: Snort rule database statistics.

4. EVALUATION

We conduct two experiments to evaluate the performance of the approximation pattern matching method. The first experiment models the performance of the preprocessor which uses the method described in Figure 1c. This configuration of the preprocessor is ideal for hardware implementation on an FPGA or network interface card. The evaluation is focused on three aspects: (i) the memory footprint of the fingerprint tables required to store the patterns; (ii) the number of table lookups per packet — that varies with the step size s as given by (1); and (iii) the number of false positives in the packet trace, i.e. the number of packets unnecessarily forwarded to the precise pattern matching algorithm.

The second experiment models the performance of the preprocessor which utilizes the method described in Figure 1d. This configuration is more suited to a software implementation on a general purpose processor. The evaluation is focused on two aspects: (i) the time required to process a stream of packets with and without the preprocessor; and (ii) the fraction of the original data that is forwarded to the precise pattern matching algorithm.

In order to provide a realistic packet stream, we consider two long packet traces (Trace 1 and Trace 2) with full payloads collected in a residential access network [9]. The site where the trace was collected represents a user population estimated in the order of 20,000 with a large number of services and applications. Trace 1 is 20 minutes long and contains 19,018,509 packets. Trace 2 is 70 minutes long and contains 67,875,101 packets.

String Patterns. The set of patterns we search in the payloads are derived from the Snort rules database as of August 25, 2005. The rule statistics for this database are shown in Figure 1. This table shows that there are 2,690 rules in Snort which require pattern matching. Payload pattern matching within Snort is specified by the `content`, `uricontent` or the `pcre` payload detection rule options. The `content` option allows to specify an exact matching pattern. The `uricontent` option results in a search in the normalized request URI field. It only works in conjunction with a HTTP preprocessor. The `pcre` option allows to specify a regular expression over the payload to be sought for. Our system can be placed in line with any other existing pre-processing that Snort performs today (e.g., HTTP inspection, protocol parsing etc.) and label all the packets that could potentially trigger a match. The capability to match on both `uricontent` and `pcre` options exist[2]. In our study, we consider only those rules that present the `content` option (1,930 rules).

The approximate fingerprint method can only be applied to content patterns that are longer than the window size

---

[2] In the current Snort implementation, almost every regular expression rule carries a `content` rule and only upon a match of this string the appropriate regular expression evaluation is triggered. The actual `content` string is often a substring of the `pcre` regular expression.
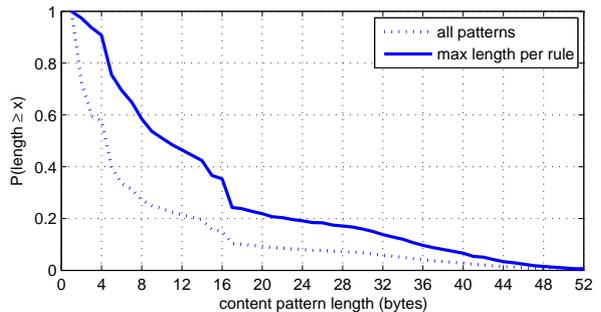

Figure 2: Complementary cumulative distribution of content string lengths

(w) chosen for computing the fingerprint. A large window size would result in more accurate pattern matching and a smaller number of false positives. Also, increasing the window size would enable us to increase the step size (s) and thus reduce the total number of fingerprint table lookups per packet.

To find an appropriate value of w we look at the distribution of the length of the patterns in the Snort ruleset. Figure 2 shows the complementary cumulative distribution of `content` pattern lengths. The dashed line shows the distribution for all patterns, while the solid line shows the length distribution for the longest pattern in each rule. We can see that a window size of 4 bytes would cover 90.78% of all rules, while a size of 8 bytes would cover 58.45% of the rules.

To accommodate rules with patterns shorter than the window size, a different pre-processing step would be needed. A combination of a Bloom filter and the approximate fingerprint would be a possible fast single lookup solution, and is a subject of our ongoing work. Discussion of this technique is beyond the scope of this article, however, relevant references indicate that a feasible solution exists [16].

Performance Results. First, we test the configuration of the pre-processor which is suited for a hardware implementation (described in Figure 1c). We evaluate the performance by running Snort on two traces: the original packet trace and a "reduced" packet trace that contains all packets deemed suspicious by our pre-processor. We measure the run times for Snort on the two traces to get an idea of the performance gain the pre-processor can provide. Trace 1 was used for this experiment.

Table 2 summarizes our experimental results. We used 16 bit wide fingerprints for window sizes of 4 bytes and 32 bit wide fingerprints for window sizes of 8 bytes. For $w = 4$, the subset of the Snort ruleset we considered contains 1637 rules and 543 unique patterns. For $w = 8$, the Snort rule subset contains 1057 rules and 426 unique patterns. Columns 3, 4 and 5 show the size, processing time and alerts returned by Snort for the original trace, while columns 6 to 9 show the same data for the reduced pre-processed trace. The last column shows the memory footprint of that particular pre-processor configuration in Kbytes. This number does not include bookkeeping overhead and represents the amount of space required to store the fingerprint table alone. In a real hardware implementation, the number shown in Table

Window (bytes)	Step (bytes)	Full Trace			Pre-Processed Trace				Memory (kb)
		Packets	Time (sec.)	Alerts	Packets	% Size	Time (sec.)	Alerts	
4	1	19,018,509	401.13	147,588	13,735,302	72.22	391.39	147,588	1.06
4	2	19,018,509	401.13	147,588	13,800,174	72.56	390.15	147,588	2.12
4	3	19,018,509	401.13	147,588	14,808,934	77.86	393.50	147,588	3.18
8	1	19,018,509	402.98	136,730	2,072,319	10.9	78.66	136,730	1.66
8	2	19,018,509	402.98	136,730	2,227,783	11.71	83.76	136,730	3.32
8	3	19,018,509	402.98	136,730	2,356,836	12.39	89.59	136,730	4.99
8	4	19,018,509	402.98	136,730	2,623,751	13.8	95.42	136,730	6.65
8	5	19,018,509	402.98	136,730	3,693,761	19.42	137.52	136,730	8.32
8	6	19,018,509	402.98	136,730	7,332,799	38.55	317.02	136,730	9.98

Table 2: Pre-Processor performance for the packet trace "Trace 1". This particular design is suited for a hardware implementation on an FPGA or NIC. The difference in Snort runtimes indicates the performance gain that can be obtained.

Window (bytes)	Step (bytes)	Snort standalone			Snort + preprocessor			Memory (kb)
		Packets	Time (sec.)	Alerts	Packets	Time (sec.)	Alerts	
8	1	67,875,101	1683.84	577,388	7,367,837	1458.9	577,234	67.3
8	2	67,875,101	1683.84	577,388	7,890,043	1368.74	577,234	70.5
8	3	67,875,101	1683.84	577,388	9,354,317	1440.63	577,234	73.3

Table 3: Pre-Processor performance when integrated with Snort for the packet trace "Trace 2". TCP stream reassembly is performed by Snort. This design is suited for software implementation on a general purpose processor.

2 would increase by a few kilobytes depending on the hash table scheme used to access the table.

The tradeoffs involved in varying w and s can be seen in Table 2. For a window size of 4 bytes, the size of the pre-processed trace is almost 75% that of the original trace. This is due to the inclusion of a large number of packets as false positives and is mainly caused by the collisions in the fingerprint space (just 16 bit wide). Increasing the step size s for this window size results in a very small increase in the size of the pre-processed trace. This is caused by another small increase in the number of false positives due to the fact that we move the window forward by more than one byte. In general, increasing the step size s, results in: (i) increasing the memory footprint for the pre-processor, (ii) reducing the per packet table lookups (see Equation 1), and (iii) increasing the number of false positives generated by the pre-processor. An increase in the number of false positives results in an increase in the size of the pre-processed trace. For a window size of 8 bytes, false positives due to collisions in the fingerprint space are no longer dominant (32 bit wide fingerprints) and the effects of varying the step size are more pronounced in this case.

In general, desirable configurations of the pre-processor should have a low memory footprint, generate a low number of false positives (i.e. the size of the pre-processed trace should be small), and have a reduced number of table lookups per packet. Setting w to 8 and s to a value between 2 and 5 provides these characteristics. On the average, the size of the pre-processed trace is between 10% to 20% of the original trace which translates in a corresponding improvement in the runtime of Snort. Note that the number of alerts returned by processing the full and reduced traces are the same for all cases.

The second experiment evaluates the configuration of the preprocessor when implemented in software on a general purpose processor. In order to measure processing time, the approximate pattern matching technique described in Figure 1d is implemented as a Snort preprocessor. The number of packets processed by Snort and the time required to process these packets are measured while executing Snort with and without the preprocessor.

Table 3 summarizes the results of this experiment. Snort was run with TCP stream reassembly enabled so that patterns are searched for in individual packets as well as reassembled stream buffers. Window size w was set at 8 bytes, 32 bit fingerprints were used, and step size s was varied from 1 byte to 3 bytes. When Snort is run with the preprocessor, the amount of packets processed varies from 10% to 14% of the total number of packets in Trace 2. This translates to a 14% to 19% improvement in the runtime of Snort when the preprocessor is enabled. The performance gain is limited given that in this software solution, the processing time is inherently dominated by moving the packets to and from memory.

Column 9 shows the memory footprint of the preprocessor configuration and includes the overhead for storing the hash table unlike the memory footprint shown in Table 2. A hash table with 16,384 entries is used. The top level of the hash table contains pointers to bucket chains. This hash table organization offers an ideal tradeoff between speed of hash function computation/table lookup and memory footprint. In comparison, the Snort data structure for the Aho-Corasick pattern matching algorithm requires 57.02 MBytes for the same set of patterns that the preprocessor takes as input.

In Table 3, a point of diminishing return is reached when the step size s is increased to 3 bytes. This value causes an increase in the runtime of Snort when compared to a preprocessor with step size 2 bytes. This is due to the increase in the number of packets that the preprocessor sends to Snort (column 6 in Table 3). This is a direct effect of an increase in the false positive rate caused by incrementing the step

size from 2 bytes to 3 bytes. This effect can also be seen in Table 2. The number of alerts returned by Snort when run with and without the preprocessor (shown in column 5 and column 8) differ. This is a side effect of the way that Snort reports alerts for a reassembled stream which causes multiple duplicated alerts to be raised for each packet that belongs to the stream.

The results in Table 3 show that the processing cost of the preprocessor is not prohibitively large for a software implementation. A hardware implementation can further improve performance by exploiting parallelism during fingerprint table lookups. The fingerprint approach is computationally lightweight and successive fingerprints over a sliding window can be incrementally computed in constant time using the technique presented in [8]. Considering all these factors into account, we argue that a hardware implementation of the preprocessor can process data at high line rates in real time.

5. DISCUSSION AND FUTURE WORK

We have presented a method to accelerate pattern matching using approximate fingerprints. Our preliminary results show the feasibility of this approach, which can significantly reduce the load on precise pattern matching methods used in current network intrusion detection systems. The approach has two other favorable properties: an extremely small memory footprint and low number of memory accesses per packet. This enables a potential hardware implementation of the method, necessary for high-speed links.

This first study of approximate fingerprinting has raised several issues that are currently under investigation. Our results indicate a fingerprint window size of 8 bytes to be the most appropriate in terms of number of memory accesses, as well as to reduce the number of false positives. Although this does not cover all the rules present in Snort, relevant literature and our ongoing work indicate that a single-lookup solution for the short patterns is feasible and could well be integrated with the approximate fingerprinting solution.

Finally, a fundamental advantage of a fingerprint-based solution is the ability to indicate candidate matches (as opposed to e.g. a Bloom filter). This is a yet-unexplored aspect of our solution. The candidate match may indicate to Snort at what offset to start the precise pattern matching algorithm, and what patterns to focus on. It may also facilitate other pattern matching applications, such as rapid worm detection, by gathering approximate statistics of frequent content blocks.

6. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[2] K. Anagnostakis, E. Markatos, S. Antonatos, and M. Poluchronakis. E^{2}xB: A domain-specific string matching algorithm for intrusion detection. In *Proceedings of IFIP Information Security Conference*, May 2003.

[3] H. Bos and K. Huang. Towards software-based signature detection for intrusion prevention on the network card. In *Proceedings of Symposium on Recent Advances in Intrusion Detection*, Sept. 2005.

[4] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of ICALP*, pages 118–132, July 1979.

[5] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *Proceedings of Symposium on High Performance Interconnects (HotI)*, pages 44–51, Aug. 2003.

[6] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for content filtering. In *Proceedings of ANCS*, Oct. 2005.

[7] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of Usenix Security*, Aug. 2004.

[8] U. Manber. Finding similar files in a large file system. In *Proceedings of Usenix Conference*, 1994.

[9] A. Moore, J. Hall, E. Harris, C. Kreibich, and I. Pratt. Architecture of a network monitor. In *Proceedings of Passive and Active Measurement Workshop*, Apr. 2003.

[10] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[11] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31, 1999.

[12] M. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, Department of Computer Science, 1981.

[13] M. Roesch. Snort: Lightweigth intrusion detection. In *Proceedings of Usenix LISA*, Nov. 1999.

[14] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *OSDI*, Dec. 2004.

[15] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[16] H. Song, T. Sproull, M. Attig, and J. Lockwood. Snort offloader: A reconfigurable hardware NIDS filter. In *Proceedings of 15th International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finland, Aug. 2005.

[17] N. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, 2000.

[18] Y. Sugawara, M. Inaba, and K. Hiraki. Over 10 Gbps string matching mechanisms for multi-stream packet scanning systems. In *Proceedings of Field Programmable Logic and Application*, Apr. 2004.

[19] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proceedings of the International Symposium on Computer Architecture*, June 2005.

[20] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of IEEE Infocom*, Mar. 2004.

[21] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings of Usenix Conference*, pages 153–162, 1992.