Piranha + Cuckoo Snort NIDS

Pornchai Korpraserttaworn* and Surin Kittitornkun*

Abstract

Network Intrusion Detection Systems (NIDSs) provide an important security function to defend network attacks. As network speeds and workloads increase, it is important for NIDSs to be highly efficient. Most NIDSs must check for thousands of known attack patterns in every packet, making pattern matching the most expensive part of signature-based NIDSs in terms of processing and memory resources. Piranha+ Cuckoo Snort combines filtering of Piranha and hashing of Cuckoo together. Especially the matching speed is faster up to 29% than that of Piranha. In this paper, we first introduce the background of Snort NIDS, Piranha, Cuckoo hashing and then present the experimental results. We then discuss the results and finally draw conclusions.

Keyword: Network Intrusion Detection System, Piranha, Cuckoo Hashing, Pattern Matching, Snort

1. Introduction

Network Intrusion Detection Systems (NIDSs) provide a powerful mechanism to defend against wellknown attacks on a computer network or detect network abuse. NIDSs are mainly divided into two major categories: signature-based and anomaly detection. Anomaly-detection NIDSs try to spot abnormal behaviour on network based on statistics like rate of connections, traffic overload or unusual protocol headers. On the contrary, the detection mechanism of a signature-based NIDS is based on a set of signatures, each describing a known attack. As an example, a signature taken from latest Snort, is

alert tcp any any -> HTTP_SERVER 80 (content:"/root.exe"; nocase;)

This signature instructs that if "/root.exe" is found inside the payload of a TCP packet that is originating from any host and any source port and is destined to an HTTP server on port 80, then an attack on the web server is taking place. While this signature requires full packet inspection, there exist simpler signatures that require only header lookups. Pattern (string) matching inflicts a significant cost to the performance of signature-based NIDSs. Previous research results suggest that 30% of total processing time is spent on pattern matching [1], while in some case like Web-intensive traffic, this percentage rises up to 80% [2]. Apart from processing time, memory demands of an NIDS may reach at high levels due to rule-set growth. Although algorithms with lower memory demands have been developed, their performance in comparison with algorithms that consume more memory is still poor. Given the fact that network speed/throughput increases every year, pattern matching evolves to a highly demanding process that needs special considerations. Minimizing the demands of pattern matching leaves headroom for further heuristics to be applied for intrusion detection, like anomaly detection or sophisticated preprocessors.

This paper is organized as follows. Section 2 introduces Snort, an open source NIDS, related backgrounds. The proposed Piranha+Cuckoo Snort is elaborated in Section 3. Results and conclusions are discussed and drawn in Section 4 and 5, respectively.

2. Snort and related backgrounds

2.1 Snort

Snort is a libpcap-based packet sniffer and logger that can be used as a lightweight network intrusion detection system (NIDS). See the data flow of Snort in Figure 1. It features rule-based logging to perform content pattern matching and detect a variety of attacks. In addition, it can probe, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and many more. Snort has real-time alerting capability, with alerts being sent to syslog, Server Message Block (SMB) "WinPopup" messages, or a separate "alert" file. Snort is configured using command line switches and optional Berkeley Packet Filter commands. The detection engine is programmed using a simple language that describes per packet tests and actions. Ease of use simplifies and expedites the development of new detection rules. For example, when the IIS Show code web exploits were revealed on the Bugtraq mailing list, Snort rules to detect the probes were available within a few hours.

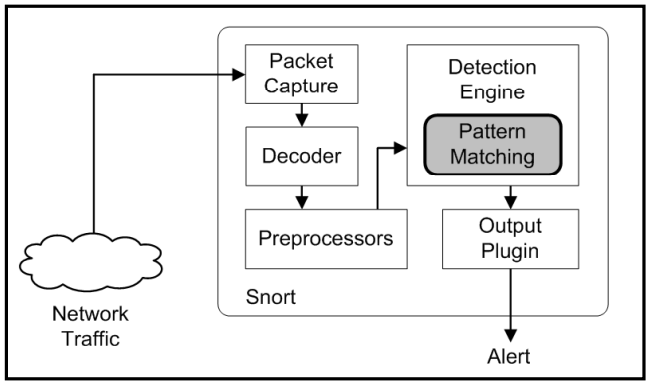

***Figure 1**: Snort's Data Flow and Pattern Matching Engine.*

2.2 Piranha [3]

Piranha Snort consists of four main stages including Preprocessing and Searching.

* Department of Computer Engineering Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabung.

• Preprocessing

Piranha treats every byte-aligned pattern as a set of 32-bit (4-byte) sub-patterns or sequences. For example, the pattern "/admin.exe" can be considered as the set of its 32-bit byte-aligned sequences as follows, "/adm", "admi", "dmin", "min.", "in.e","n.ex" and ".exe". The 32- bit partitioning is chosen because of faster native 32-bit integer operations of the common 32-bit Intel Architecture. The next step is to select rare sub-patterns to form a filter table (a linear hash table using modulo function) as shown in Figure 3.

• Searching

Piranha combines 4 bytes of the incoming payload as one sequence and hashes each sequence (unsigned 32-bit integer) using modulo function with the table size of 16,384. The hash value points to a linked list of 4-byte sequences. Each 4-byte sequence corresponds to the exact rule pattern for final comparison. Due to the poor hashing (modulo) function, numerous collisions occur resulting in a long chain of up to 25 nodes and consequently long matching time.

2.3 Cuckoo Hashing

Cuckoo Hashing [4] is a dynamic mapping of a static dictionary. The dictionary uses two hash tables, T_1 and T_2, each consisting of r words, and two hash functions $h_1, h_2 : U \rightarrow \{0, \ldots, r-1\}$. Every pattern $x \in S$ is stored in either cell $h_1(x)$ of T_1 or cell $h_2(x)$ of T_2, but never in both. The lookup function is

```
function lookup(x)
    return T1[h1(x)] = x OR T2[h2(x)] = x;
end
```

Each pattern lookup [function lookup(x)] needs two table accesses. In fact, it is optimal among all dictionaries using linear space, except for special cases, see [5]. It is also shown in [5] that if $r \geq (1+\varepsilon)n$ for some constant $\varepsilon > 0$ (i.e., the tables are a bit less than half full), and h_1, h_2 are picked uniformly at random from an (O(1), O(log n))-universal family, the probability that there is no way of arranging the patterns of S according to h_1 and h_2 is O(1/n).

We now consider a simple implementation of the above, still assuming $r \geq (1+\varepsilon)n$ for some constant $\varepsilon > 0$. Deletion is very simple to perform in constant time, not counting the possible cost of shrinking the tables if they are becoming too sparse. As for pattern insertion [procedure insert(x)], it turns out that the "cuckoo approach", kicking other patterns away until every pattern has its own "home", works very well. Specifically, if a pattern x is to be inserted we first see if cell $h_1(x)$ of T_1 is occupied. If not, x is inserted. Otherwise we set $T_1[h_1(x)] \leftarrow x$ anyway, thus making the previous occupant "homeless". This pattern y is then inserted in T_2 in the same way, and so forth iteratively, see Figure 2.

It may happen that this process loops infinitely, as shown in Figure 2. Therefore the number of iterations is bounded by a value MaxLoop. If this MaxLoop is reached, the patterns shall be rehashed in tables T_1 and T_2 using two new hash functions. There is no need to allocate new tables for the rehashing: We may simply run through the tables to delete and perform the usual insertion procedure on all patterns found not to be at their intended position in the table. (Note that kicking away a pattern that is not in its intended position simply corresponds to starting a new insertion of this pattern.)

```
procedure insert(x)
    if lookup(x) then return;
    loop MaxLoop times
      x = T1[h1(x)];
      if x = NULL then return;
      x = T2[h2(x)];
      if x = NULL then return;
    end loop
    rehash();
    insert(x);
    end
```

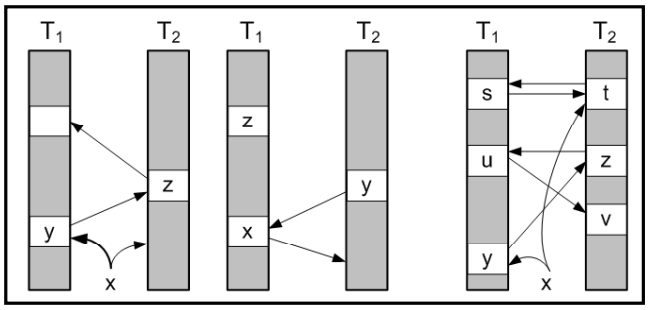

***Figure 2** : Examples of Cuckoo Hashing insertion. Arrows show possibilities for moving patterns.*

(a) Pattern x is successfully inserted by moving patterns y and z from one table to the other.
(b) Pattern x cannot be accommodated and a rehash is necessary.

3. Piranha + Cuckoo Snort

As mentioned earlier, sequence filtering and searching of Piranha in Figure 3 can be replaced with Cuckoo hashing to not only solve the collision in Piranha linear/modulo hash table but also reduce the run time complexity. In the searching phase, Piranha+Cuckoo in Figure 4 only hashes two times while Piranha originally hashes once but has to traverse along the linked list up to 25 nodes to find whether it is matched or not matched. Hence, Piranha+Cuckoo Snort can support the increasing number of known attack patterns with almost constant look up time.

4. Experiment and results

We evaluate the performance of Piranha against Piranha-Cuckoo algorithms in Snort 2.4.2 using five packet traces. All Snort preprocessors were disabled.

4.1 Environment

All the experiments are conducted on a 500-MHz Pentium III machine with 32 KB of L1cache, 512 KB of L2 cache, and 512 MB of main memory. The host operating system is Linux (kernel version 2.4.27, Debian 3.1r5 Sarge). We use five full-packet traces Darpa01, Darpa02, Darpa03, which were collected during the DARPA evaluation tests at MIT Lincoln Laboratory in 1999 and Defcon9.1, Defcon9.2 which were collected during the Capture The Flag Contest 2001.

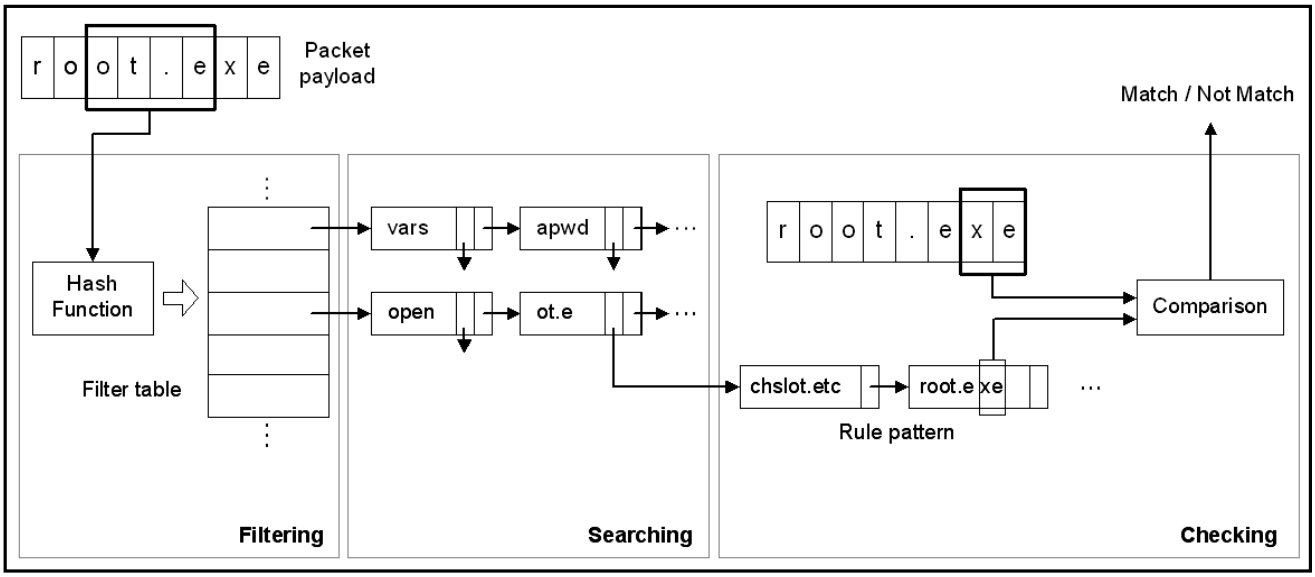

***Figure 3**: Piranha pattern matching algorithm.*

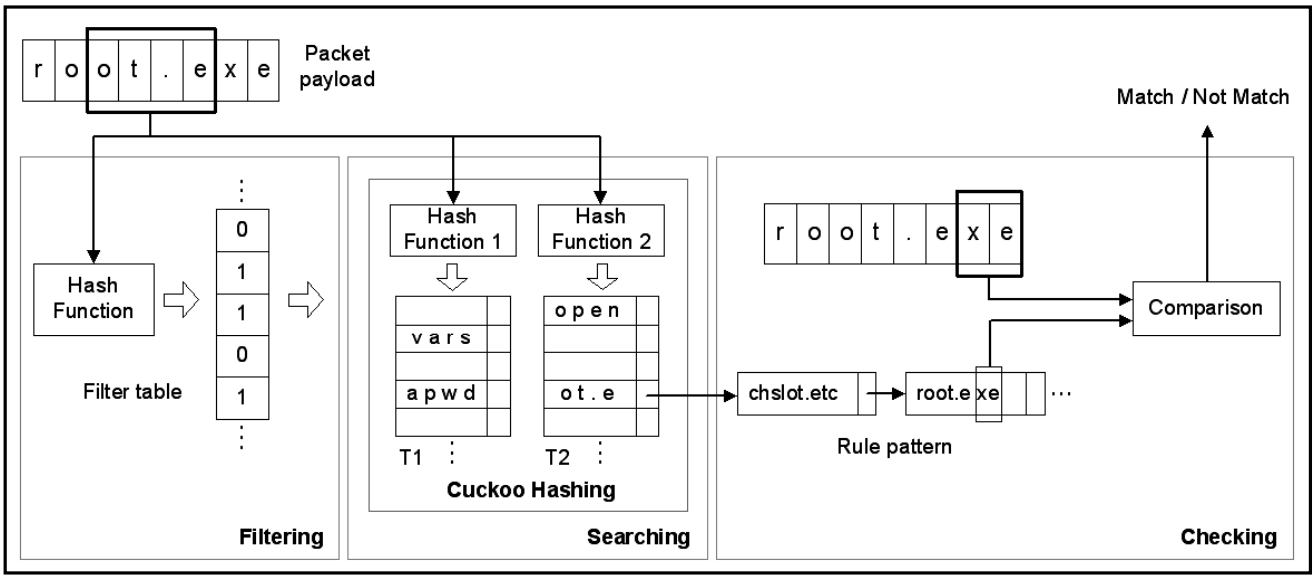

***Figure 4**: Piranha + Cuckoo pattern matching algorithm.*

Each packet trace is read by Snort using –r option in a tcpdump format file as shown in Figure 5. Therefore, Snort can run at almost 100% processor utilization. Each trace is run 10 times to average out the random nature.

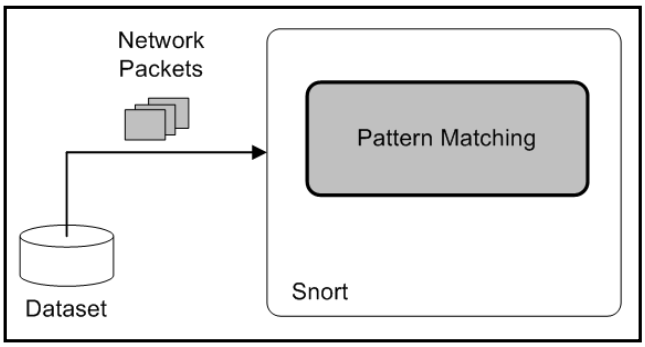

***Figure 5** : Pattern matching of network packets from a tcpdump data trace file.*

4.2 Performance Comparison

We first make sure that Piranha+Cuckoo Snort works correctly as it can detect the same number of alerts as Piranha Snort in Table 1. Then, we can determine the performance of each algorithm by measuring the following parameters: number of matching sequences and amount of matching time. The number of matching sequences describes the run time complexity as shown in Table 2. Piranha+Cuckoo can greatly reduce the number of matched (searched) sequences because Cuckoo hashing requires two lookups to check if it is a match or not. Furthermore, Piranha must traverse along the chain to find out a match or no match.

The amount of matching time can be measured using times, the standard library C function. In each trace, the average matching time of Piranha (T1) is normalized to that of Piranha+Cuckoo (T2) in Table 3 and plotted in Figure 6. As a result of lower number of Matched Sequences in Table 2,

the average matching time of Piranha+Cuckoo is faster than that of Piranha from 2.2% - 29.58%. It can be observed that the %Speed Up in Table 2 corresponds to the Difference of Table 2. For example, %SpeedUp of 2.2% corresponds to the Difference 869,330 Sequences of Defcon 9.1. Accord ingly, %SpeedUp of 29.58% corresponds to the Difference 26,005,446 Sequences of Defcon 9.2.

In summary, the %SpeedUp depends on the nature of each packet trace. The 2.20%SpeedUp of Defcon 9.1 trace is due to lower Difference of sequences matched by Piranha+Cuckoo but rather high number of Alerts detected. On the contrary, the 29.58 %SpeedUp of Defcon9.2 trace is due to higher Difference but rather low number of Alerts detected. That means Piranha+Cuckoo is more efficient than Piranha by matching less but still able to detect the same number of Alerts as Piranha.

***Table 1** : Number of Alerts Detected by Piranha+Cuckoo and Piranha.*

Traces	Piranha	Piranha +Cuckoo
Darpa01	3701	3701
Darpa02	3572	3572
Darpa03	2424	2424
Defcon9.1	19910	19910
Defcon9.2	193	193

***Table 2** : Average Matching Time (Seconds) of Piranha and Piranha+Cuckoo.*

Traces	Input Sequences	Matched by Piranha	Matched by Piranha + Cuckoo	Difference
Darpa01	118,867,310	69,966,901	47,738,766	22,228,135
Darpa02	35,344,980	17,671,526	13,030,892	4,640,634
Darpa03	98,831,474	59,096,662	41,367,680	17,728,982
Defcon9.1	591,277,512	5,049,044	4,179,714	869,330
Defcon9.2	202,432,131	44,602,569	18,597,123	26,005,446

***Table 3** : Number of Input 4-byte Sequences, Matched by Piranha, Matched by Piranha+Cuckoo, and Difference.*

Packet	Average Matching Time (seconds)		SpeedUp (%)
	Piranha (T1)	Piranha+Cuckoo (T2)	
Darpa01	15.709	14.841	5.85
Darpa02	4.335	4.125	5.09
Darpa03	13.796	13.223	4.33
Defcon9.1	12.313	12.048	2.20
Defcon9.2	8.201	6.329	29.58

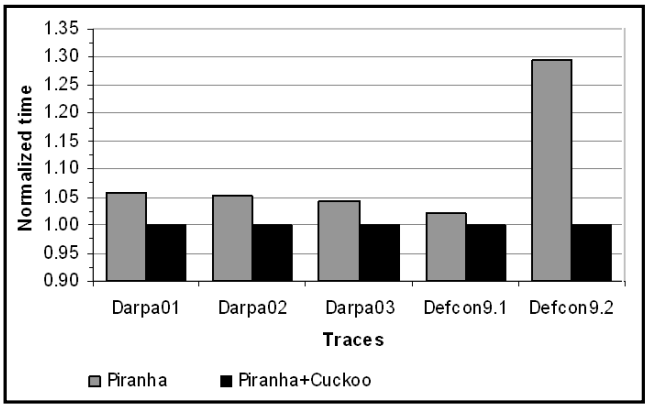

***Figure 6** : Normalized average matching time of Piranha and Piranha+Cuckoo algorithm in 5 packet traces.*

5. Conclusions and Future Works

In this paper, we presented a new pattern matching based on Cuckoo hashing of Snort NIDS. The algorithm is implemented on top of Piranha [3], which is almost optimal in terms of speed and memory resource compared with many famous existing ones [3]. Experiment results show that ours can significantly improve the matching speed of Piranha itself at up to 30% for current and future attack patterns. For future works, we shall examine memory requirements of Piranha+Cuckoo Snort and improve our hashing function to support Gigabit networks.

6. Acknowledgment

I would like to thank Mr. Thinh Tran Ngoc for his support during his Doctoral study at KMITL.

7. References

[1] M. Roesch, "Snort: Lightweight intrusion detection for networks". *In Proceedings of the 1999 USENIX LISA Systems Administration Conference,* November. 1999. http://www.snort.org/.

[2] M. Fisk and G. Varghese, **"**An analysis of fast string matching applied to content-based forwarding and intrusion detection", *Technical Report CS2001-0670 (updated version),* University of California - San Diego, 2002.

[3] S. Antonatos, M. Polychronakis, P. Akritidis, Kostas D. Anagnostakis, and Evangelos P. Markatos: "Piranha: Fast and Memory-efficient Pattern Matching for Intrusion Detection" . *In Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC 2005),* May. 2005.

[4] R. Pagh and F.F. Rodler. "Cuckoo hashing". *In ESA: Annual European Symposium on Algorithms,* volume 2161 of LNCS. Springer, 2001.

[5] Rasmus Pagh. **"**On the Cell Probe Complexity of Membership and Perfect Hashing". *In Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01),* pp. 425–432. ACM Press, 2001.