Improving the Resilience of an IDS Against Performance Throttling Attacks

Govind Sreekar Shenoy[1], Jordi Tubella[1], and Antonio González[1,2]

[1] Department of Computer Architecture,
Universitat Politècnica de Catalunya, Barcelona, Spain.
[2] Intel Barcelona Research Center, Barcelona, Spain.
Email: {*govind,jordit*}*@ac.upc.edu, antonio.gonzalez@intel.com*

Abstract. Intrusion Detection Systems (IDS) have emerged as one of the most promising ways to secure systems in the network. To be effective against evasion attempts, the IDS must provide tight bounds on performance. Otherwise an adversary can bypass the IDS by carefully crafting and sending packets that throttle it. This can render the IDS ineffective, thus resulting in the network becoming vulnerable.
We present a performance throttling attack mounted against the computationally intensive string matching algorithm. This algorithm performs string matching by traversing a finite-state-machine (FSM). We observe that there are some input bytes that sequentially traverse a chain of 30 pointers. This chain of traversal drastically degrades performance, and we observe a 22X performance drop in comparison to the average case performance. We investigate hardware and software mechanisms to counter this performance degradation. The software mechanism is targeted for commodity general purpose CPUs. While the hardware-based mechanism uses a parallel traversal suitable for network processor architectures. Our results show that our proposed mechanisms significantly improves (by over 3X magnitude) string matching algorithm's worst performing cases.

1 Introduction

Intrusion Detection Systems (IDS) are emerging as one of the most promising ways of providing protection to systems on the network. By monitoring the traffic in real time, an IDS can detect and also take preventive actions against suspicious activities. To be effective, an IDS must be able to inspect packets at wire speeds. The consequences of not doing so can result either in undetected malicious packets or expensive packet drops. An adversary can also bring the IDS to this state of not being able to process packets at wire speeds. Such attempts are commonly referred to as evasion[6, 9, 18], and stem from weaknesses in some part of IDS processing.

Evasion can come in various flavors. An example of evasion is clever packet fragmentation at "malicious content" boundaries, thus tricking the IDS from inspecting malicious content. Other examples include deliberate packet header corruption and stream re-assembly. The nature and ease of evasion makes it

very appealing for malicious hosts to bypass the IDS. Evasion can also occur by throttling the performance of the IDS. Since the system is unable to keep up with the wire speed, it can lead to the IDS being disabled and attack flood gates opened. For this to occur, an adversary exploits the wide performance gap between the average case and worst-case processing time[6, 15, 21]. This can also be viewed as a class of Denial-of-service (DoS) attack that targets system resource utilization[13]. Earlier works in this direction investigate attack and defense mechanisms for hash tables[6] used by an IDS. Additionally, other works explore weaknesses due to synctatics of signature specifications[21] in the Snort IDS.

In this work we present a performance throttling attack mounted against the string matching algorithm used by an IDS. An IDS like Snort[19] operates by scanning packets for malicious content using a database of >40,000 known attack strings. So Snort uses the Aho-Corasick algorithm[1] to perform a multi-string matching against the packet payload. Since the packet payload needs to be scanned and compared with a database of >40,000 strings, so it is computationally very intensive. Hence, the string matching algorithm can be susceptible to performance throttling attacks. A closer look at the processing time per payload byte of the string matching algorithm reveals a wide performance variation. We observe that there are payload bytes that need 22X processing times in comparison to the average case. Further, the cause of this variance in performance is due to the sequential traversal of a chain of pointers. Our counter-measure focuses on improving the worst-case performance by accelerating this sequential chain traversal. We propose two mechanisms - hardware-based and software-based mechanisms - to counter this performance degradation. The hardware-based mechanism is targeted for a highly parallel architecture like the network processor ([5, 11]). The software-based mechanism is for commodity general purpose CPUs. Our results indicate that our proposed mechanisms significantly improves the worst-case performance.

The rest of the paper is organized as follows. Section 2 provides a brief background on the Aho-Corasick algorithm. Section 3 presents the motivation of this work. Section 4 details our proposed counter-measure and our architecture. The simulation methodology is discussed in Section 5, and Section 6 presents the performance results. Section 7 discusses the related work in this area. Section 8 provides future directions.

2 Background

An IDS like Snort operates by inspecting packets for prior reported attacks. This database of attack strings are byte patterns that have commonly occurred and detected in attacks. The vast variety of attacks and their constant evolution bloats the attack string database. We observe that there are 42,670 attack strings in the Snort April-2010 ruleset[1] release. So Snort commonly uses a multi-string

[1] A rule in Snort typically contains multiple attack strings. We instrumented Snort to dump all the strings

matching algorithm like the Aho-Corasick algorithm[1] for attack detection. This algorithm works by constructing a FSM using the set of attack strings. Once the FSM is constructed, incoming bytes from packets are used to traverse this FSM. We provide a brief overview of the Aho-Corasick algorithm with an example.

Consider the set of strings: **attacker, tacked, acken, ckeh, ket**. Figure 1 shows the corresponding Aho-Corasick FSM constructed from these strings. The FSM is built in two stages. In the first stage, characters from the strings

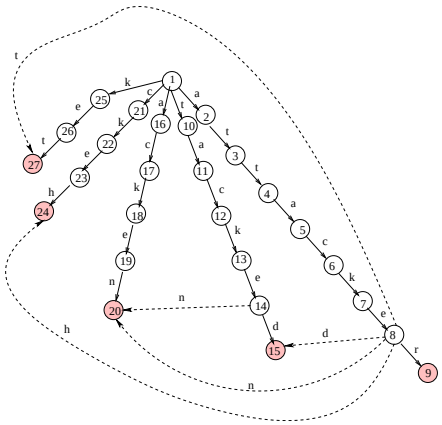


Fig. 1. Example of the Aho-Corasick Finite State Machine.

are added to the FSM. This is done in a way that strings that share a common prefix also share the same set of parents in the FSM. The edges corresponding to this stage are shown as thick lines. The nodes **9, 15, 20, 24, 27** indicate a match for **attacker, tacked, acken, ckeh, ket** respectively. These nodes also store a pointer to the list of matched strings. For example, **node 9** stores a pointer to **attacker**. The second stage in building the FSM consists of inserting failure edges. When a string match is not found, it is possible for the suffix of one string to match the prefix of another. So failure edges need to be inserted. Failure edges are shown with dotted lines. For figure clarity, only a few failure edges are shown. Once this FSM is built, the algorithm traverses it with the payload bytes. In case the payload byte does not match any of the examined edges, then the traversal is restarted from the root-node.

An important issue with this FSM is the large storage space needed. This huge storage size requirement also impacts the performance efficiency of an IDS, due to the large working set size. Some earlier works in reducing the storage space needed for FSM have proposed removing the inherent redundancy in the FSM. A common example of redundancy is due to failure edges. Consider **node 8** and its edge **d**. This is a failure edge and is identical to the edge from **node 14**. Thus **Node 14** is a failure pointer of **node 8**, and this traversal can also be done by jumping to **node 14**. In this way all failure edges are eliminated.

Figure 2(a) shows the FSM built with the failure pointer optimization. We observe that 93% of the edges are failure edges, and hence the failure pointer optimization provides important area benefits. As a consequence, earlier works[2, 12, 20, 22, 23, 25] have optimized the failure edges in this manner. For the rest of the paper, we consider this optimized FSM.

The FSM constructed using the Aho-Corasick algorithm is very similar to a deterministic finite automata (DFA). In fact Snort and other IDSs[16] also use regular expressions to specify attack strings. These regular expressions are again converted to DFAs or NFAs. Hence our work is equally applicable to regular expressions used in an IDS. Note that the optimized FSM thus built is very similar to a NFA. A NFA, unlike a DFA, can have multiple active states. Further these active states need to be traversed sequentially. In order to efficiently traverse the FSM, the Snort[19] IDS uses a backtracking based heuristic for traversing the NFA. This heuristic is very similar to the failure pointer optimization, and so our work can be adapted to accelerate NFA traversal in Snort.

3 Motivation

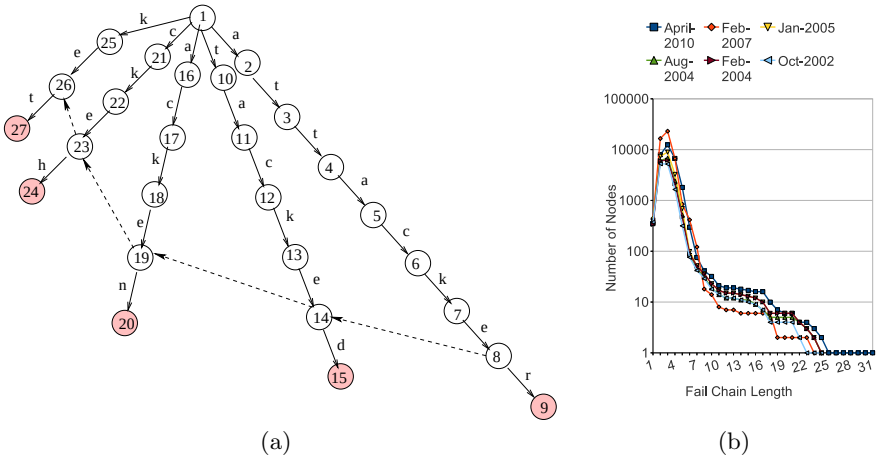
The optimization of failure chains significantly compacts the data structure. However this has a drawback. A node with failure pointers may need additional processing when there are no matching edges. In some cases we observe that this additional processing is a significant overhead.

We illustrate this more clearly with an example. Let the input bytes to the optimized FSM in Figure 2(a) be **a, t, t, a, c, k, e, t**. The first 6 bytes lead up-to **node 8**. For the final byte, **t**, the failure pointer needs to be traversed as there are no matching edges at **node 8**. Hence, **node 14** - the failure pointer of **node 8**- is accessed. Here again there are no matching edges, and so the failure pointer of **node 14** is accessed. This is repeated until a matching edge is found, or the traversal is restarted from the root-node. Note that these chain of failure pointers are accessed sequentially and sometimes wastefully as well. This can lead to significant performance degradation when large chains are visited.

Figure 2(b) shows the failure chain length distribution for various Snort database releases. We define failure chain length of a node as the maximum number of failure pointers that can be traversed starting from that node. The failure chain length of **node 8** in the above example is 4. It is very interesting to observe that there are nodes with failure chain length of up-to 31. Thus for bytes accessing failure edges of these nodes, the processing time can be high. We investigate the performance impact of traversing failure chains.

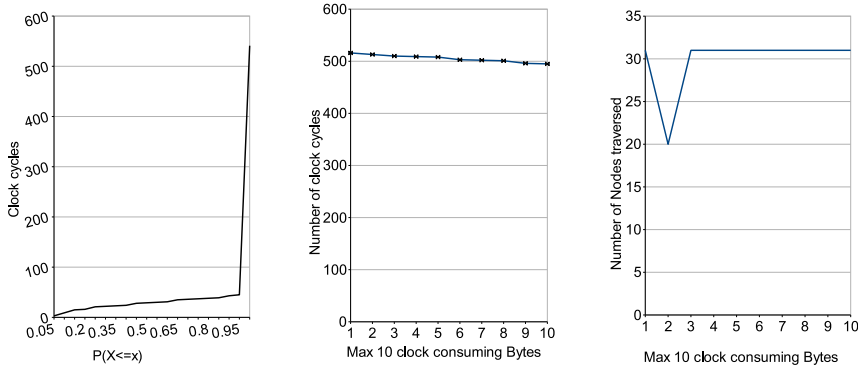
Figure 3(a) shows the CDF of processing time per byte[2]. We see that 95% of input bytes need less than **31 cc**, thus leading to an average processing time of **23.5 cc/B**. However, there are bytes that need up-to **516 cc**. This clearly indicates that there is a wide variation in processing time. We investigate the

[2] Processing time per byte is measured as the total number of clock cycles (cc) needed to complete the processing of a byte.



(a) (b)

Fig. 2. Impact of Failure chain



(a) (b) (c)

Fig. 3. Impact of Failure chains on Performance

cause of this wide variation, by examining the processing of the ten most clock consuming bytes (refer to Figure 3(b)). This is also the tail end of the CDF. As seen in Figure 3(b), we observe that these bytes need at-least **495 cc**. The cause of the enormous processing time is due to the traversal of a chain of failure pointers. In contrast, on examination of the relatively lesser clock consuming bytes (left half of 0.95 probability in the CDF plot), we observe that these bytes traverse at most 3 failure pointers. This clearly shows the significant impact of traversing a large chain of failure pointers.

The dependency of processing time on the failure chain length makes the IDS vulnerable to performance throttling attacks. Hence it is important to accelerate the failure pointer traversal. So we study techniques to do the same.

4 Proposed Counter-measure

Intrusion detection systems are commonly deployed in routers. Routers in turn can use network processors that have a high degree of parallelism. For example, the Intel IXP 2400[11] has a total of 64 threads. We propose a hardware-based mechanism that uses 2 cores and it is suitable for network processor deployment. IDSs can also be deployed in end systems in a non parallel set-up. Hence we also propose a software-based mechanism targeted for such an environment.

4.1 Hardware-based Mechanism

The processing of the failure chain takes a performance hit mainly due to the sequential nature of its traversal. So our proposal performs a parallel traversal. One engine performs the regular FSM traversal, while another engine concurrently finds the candidate failure pointer. We first describe a mechanism to identify the candidate failure pointer and to incorporate it in the traversal algorithm. Later, we present the parallel architecture used for the traversal.

Candidate Failure Pointer Identification The traversal of a chain of failure pointers can be viewed as a comparison of the edges of a node to the input byte. Further, this process is repeated for the chain of failure nodes. So we break it into comparison of a chain of outgoing edges. We illustrate this more clearly with an example. Let the input bytes to the FSM in Figure 2(a) be **a, t, t, a, c, k, e, t**. The first 6 bytes lead up-to **node 8**. For the final byte, **t**, since there are no matching edges the failure pointer is traversed. The failure pointer of **node 8**, **node 14**, is traversed. Since it is a mismatch, the failure chain is followed until **node 26**. So the main operation in the failure pointer traversal is the comparison of the input byte with all outgoing edges of a node. This is checking for membership in a set of outgoing edges, and with each set corresponding to a failure pointer.

Bloom filters[3] offer a convenient and efficient way to check - without incurring any false negatives - for set memberships. We use bloom filters to do the membership check. We create a hash for each failure pointer by using its set of outgoing edges. We term it as a bloom filter signature. We illustrate this with an example. Consider **node 8** (from Figure 2(a)), we create and store bloom filter signatures for all its failure chains, namely, **nodes 14, 19, 23, and 26**. Each of these signatures are generated using outgoing edges of each node.

Figure 4 shows the signature storage of **node 8** generated in this manner. In addition to signatures, we also store offset and fan-out of the corresponding failure pointer. This is done so that when a signature matches, we can directly jump to the matching failure pointer. The traversal using bloom filter signatures

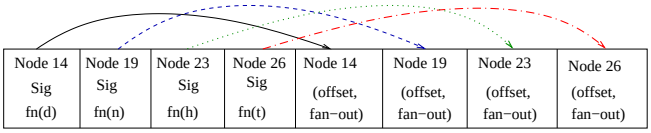


Fig. 4. Node 6 Signature Storage.

is as follows. Consider traversing **node 8** with input byte as **t**. Since there are no matching edges in **node 8**, we check if there are any matching edges in the failure chain. A signature is generated using **t**, and compared against all the failure chain signatures of **node 8**. Since **node 26** has a matching signature, we directly traverse to **node 26**. Note that in case of multiple matches, the matches are traversed sequentially. This preserves traversal correctness, as the signatures are stored in the way they are originally encountered.

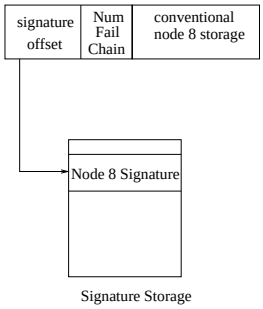


Fig. 5. Node 8 FSM Storage and Signature Access.

The failure chain signature matching can be performed independently and in parallel with the conventional node processing. In the above example, the failure chain traversal is done sequentially after checking for matching edges in a node. We accelerate it by performing the failure pointer identification concurrently with the conventional node processing. If there is no need to traverse the failure pointer, then the failure pointer identification is discarded. So our proposed architecture consists of two engines: a regular FSM traversal engine and an engine to identify the candidate failure pointer. We further decouple the memory by storing the bloom filter signatures in a separate memory bank. Our memory architecture consists of two memory banks, with one containing the FSM and the other containing signatures. This helps us in decoupling the FSM traversal from the failure chain computation. Additionally, we store a pointer to the node signature in the FSM data structure. So every node also stores a pointer to the signature database and its failure chain length. Figure 5 shows the storage for **node 8**.

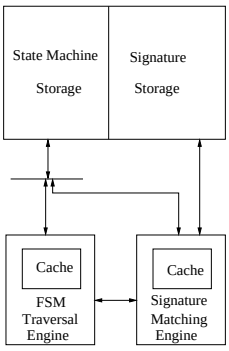


Fig. 6. Hardware Architecture.

Hardware Architecture Figure 6 shows our proposed hardware architecture. The hardware consists of a FSM traversal engine and a signature processing engine. The FSM traversal engine performs the regular state-machine traversal. We have used the FSM traversal engine as proposed in [20] and we provide here a brief summary. The traversal operations essentially consist of two steps. In the first step, all the edges of a node are scanned, and then the matching edge information is read. So we split this engine into these operations (refer to Figure 7(a), 7(b), 7(c)). In edge scanning, the set of edges are read and compared with the input byte. This is iterated over all edges until a matching edge is obtained. If a matching edge exists, then the associated edge information is read. Otherwise, the traversal is restarted from the root-node.

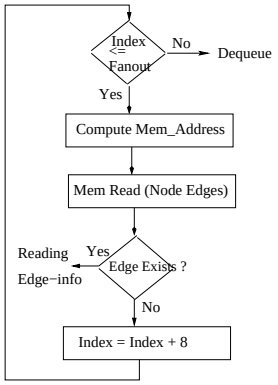
The signature matching engine performs the following functionalities. It generates the bloom filter signature using the input byte, and then compares it with the stored signatures. Signatures are of length 4 B and are generated using two hash functions[3]. Since the signature comparison is an AND operation, so we use 16 B AND operators for signature comparison. This allows us to compare four signatures at a time. If a signature matches, then the matched failure pointer is traversed. Figure 7(c) shows the flow-chart for the signature matching engine.

Our architecture concurrently perform signature comparison and the regular FSM traversal. Hence if the input byte matches an edge, the signature processing is flushed. However, if there are no matching edges, then the candidate failure pointer is obtained from the signature matching engine. Subsequently, this node is traversed by the FSM traversal engine.

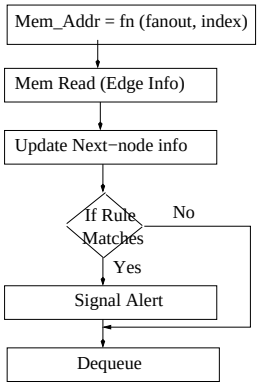
4.2 Software-based Mechanism

In this mechanism, the Aho-Corasick FSM is constructed so that there is an upper-bound on the failure chain length. This upper-bound can also be viewed a threshold value. In this mechanism, failure edges are inserted for nodes with failure chain lengths a multiple of this threshold value.

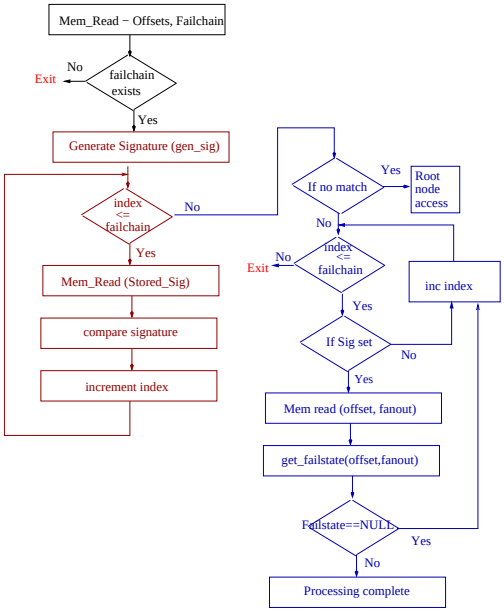
[3] A design space exploration was done to obtain these parameter values.



(a) Edge Scanning



(b) Reading Edge Information



(c) Signature Matching Engine

Fig. 7. Functionality of Signature Matching Engine and FSM Traversal Engine.

We illustrate it more clearly with an example. Consider the FSM shown in Figure 2(a). If we use a threshold value of **3**, then failure edges are inserted for nodes with failure chain length of **3**. Hence, failure edges are inserted for **node 14**. In this way we limit the failure chain traversal to a fixed upper bound. This also enables in efficiently storing the FSM as failure edges are only inserted for selective nodes and not all the nodes in the FSM. In our simulations, we explore different values of the threshold in order to find an optimal point.

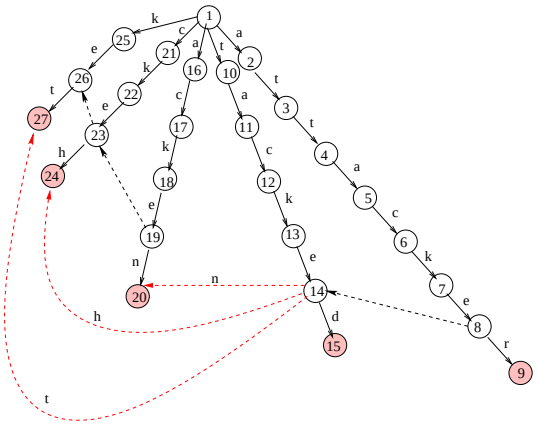


Fig. 8. Software-based Mechanism.

5 Simulation Methodology

We evaluate the performance of our proposed mechanisms, and compare it with the conventional method of sequentially traversing the failure pointer. We have used three public traces, a synthetically generated trace, and a Honeypot trace.

The public traces are from the Lincoln labs [14] and Defcon[7]. For the Lincoln labs we have used two weeks of traces (referred to by their respective week) from 1999. In the Defcon trace, we use the trace captured for the Capture the flag (CTF) game[7]. CTF is a hacking contest in the Defcon conference. The objective of this contest is to break into computers of other teams, while at the same time preventing others from do so. We have also deployed a low-interaction Honeypot running in collaboration with the Leurrecom project[17]. This Honeypot has been running for 3 months, and the logs indicate that there has been an interaction with the outside world for at-least 61 days. We have used the traces collected from this Honeypot. We also include a synthetically generated trace. The synthetic trace was generated by randomly selecting strings from the Snort rule database and further combining multiple strings. This was done to ensure minimum-sized packet (64 B).

Table 1 summarizes the traces used. Note that we have inspected TCP, ICMP and UDP packets from these traces. We have used the Snort database released on April 2010 and containing 40,678 strings. We use **average number of clock cycles per incoming byte** as the metric for performance comparison. This is computed by dividing the total number of clock-cycles by the total number of bytes. Total number of clock-cycles is the sum of **total processing time** and **total memory access time**. The **total processing time** comprises of: edge-scanning, reading edge-information, signature comparison, and signature offset computation. These processing times are obtained by assuming each of the arithmetic processing blocks need 1 cycle and branches need 2 cycles (refer

Data-sets	Mean Packet Size (B)	Num Packets (M)
Defcon	71.9	15.64
synthetic	73.64	0.120
Week 2	160.51	13.18
Week 3	200.01	14.91
Honeypot	205	0.46

Table 1. Summary of Traces used in the Evaluation.

to Figure 7(a), 7(b), 7(c)). With this assumption, edge scanning needs 6 cc plus the memory access latency.

The **total memory access time** is obtained from the trace-driven cache simulator [8], which was modified to model cache access times and processing times. The cache miss penalty is obtained from CACTI [24] by plugging into the SRAM model of CACTI the FSM memory sizes. We have used a 16k direct-mapped cache-configuration for the caches. Note that in case of the hardware-based mechanism, there are two caches each of 16k size. The cache hit time of 2 cc is used (also obtained from CACTI). The core frequency is assumed to be 3 GHz.

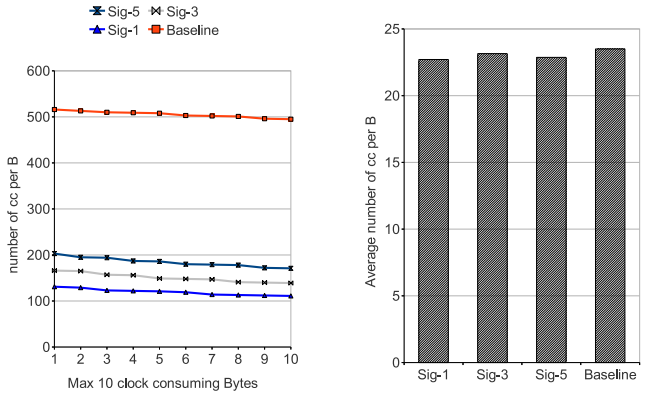
6 Results

We compare the performance of our proposed architecture with the **Baseline**. Note that the **Baseline** performs traversal using the conventional way of sequentially following failure pointers.

For the hardware-based mechanism, we have varied the minimal failure chain length. Hence signatures are kept only for those nodes with a failure chain length greater than the threshold. We have used threshold values of **1, 3, 5**. A threshold value of **1** indicates that nodes with failure chain lengths >= 2 have stored signatures. For the software-based mechanism, we have similarly varied the failure chain length threshold. So in this scheme, nodes with a given threshold failure chain length will have all its failure edges in place. We have used threshold values of **3, 5, 7**.

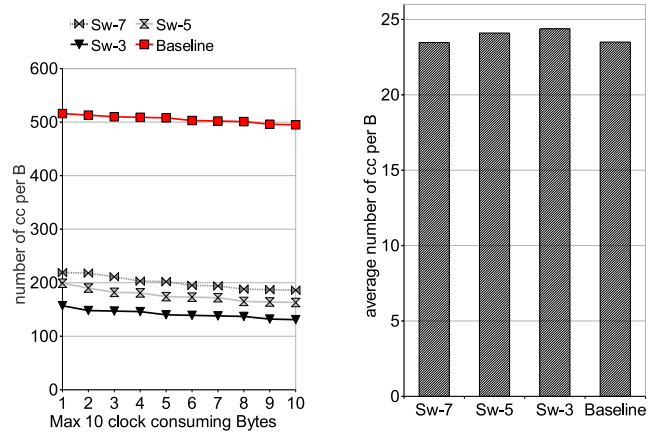
In order to evaluate the worst performance cases, we compare the processing clock cycles (cc) needed for the 10 most clock consuming bytes. Note that a byte that performs badly in one scheme may not do so in another scheme. We also compare the average-case performance. We initially report results for the synthetic trace to determine the optimal points for the hardware and software-based mechanism. Later we report results for the remaining traces.

A few terminology clarifications. **Sig-1** refers to the use of bloom-filter signatures of threshold value 1. Further, **sw-3** refers to the failure chain length of 3 used in the software-based mechanism. Figure 9(a) shows the 10 most clock consuming bytes for the hardware-based mechanism for the synthetic trace. While **Baseline** needs at least **495 cc**, the use of signatures brings it down to at most



(a) Worst-case performance (b) Average-case Performance

Fig. 9. Synthetic Trace Comparison Result for Hardware-based Mechanism



(a) Worst-case performance (b) Average-case Performance

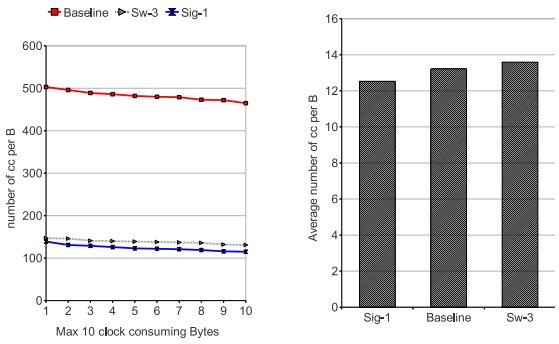
Fig. 10. Synthetic Trace Comparison Result for Software-based Mechanism

119 cc. Additionally, on a closer examination of various threshold values, we see that **Sig-1** gives the best performance. For **Sig-1** we see a worst-case performance of **119 cc** - a 4.33X improvement over the **Baseline**. Figure 9(b) shows the average-case performance, and we see that it remains unaffected.

Figure 10 shows the comparison results for the software-based mechanism. We again observe that keeping an upper-bound of the failure chain length significantly brings down the worst-case performance. While **Baseline** needs at least **495 cc** in these bytes, the software-based mechanism reduces it to at most **219**

cc. Figure 10(b) shows the average-case performance and we see that it remains largely unaffected.

We observe that **Sig-1** is the best performing configuration for the hardware-based mechanism. Further, **sw-3** performs best for the software-based mechanism. So for the remaining traces we compare the performance of **Sig-1**, **sw-3** and **Baseline**.



(a) Worst-case performance (b) Average-case Performance

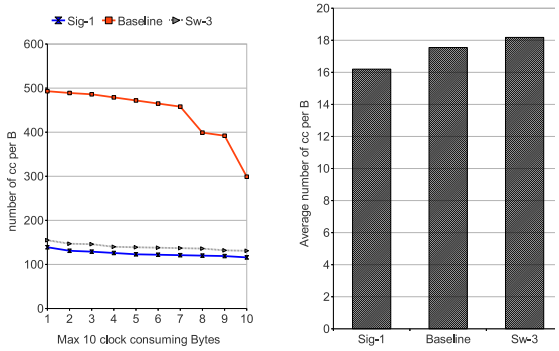
Fig. 11. Defcon Trace Comparison Results

For Defcon trace we observe a similar performance behaviour (refer to Figure 11). Comparing the worst-case performance, the hardware-based mechanism reduces the worst-case performance to **139 cc** - over 3X improvement over the **Baseline**. On the other hand, the software-based mechanism reduces the worst-case performance to **147 cc**. On comparing the hardware-based and software-based mechanisms, we observe that the hardware-based mechanism moderately outperforms the software-based mechanism.

Figures 12, 13 and 14 show the performance results for week2, week3, and Honeypot respectively. We again observe a similar behaviour, with **Sig-1** providing the best performance for the worst-case. Note however that there is a mild average-case performance degradation for the software-based mechanism.

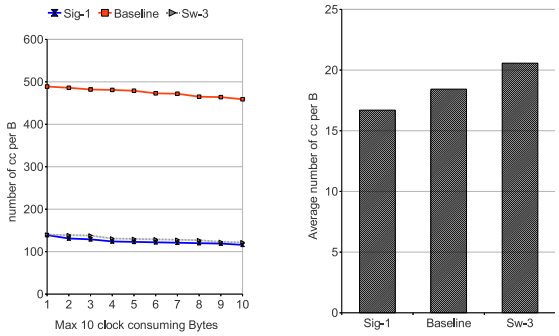
Our mechanisms needs additional memory in comparison to the **Baseline**. So we evaluate the additional storage space needed (measured in KBs) for our proposal. Figure 15 shows the storage space required for various schemes. The memory required has been normalized to the **Baseline** (706 KB). In case of the hardware-based mechanism, the additional storage space is between 34% and 84% to that of the **Baseline**.

In case of software-based mechanism, the additional storage space is between 1% to 140% in comparison to the **Baseline**. This exponential increase in storage space is due to the following. As the threshold failure chain length is reduced from **7** to **3**, the number of nodes that need to store the failure edges grows by more



(a) Worst-case performance (b) Average-case Performance

Fig. 12. Comparison Results for Week2 Trace

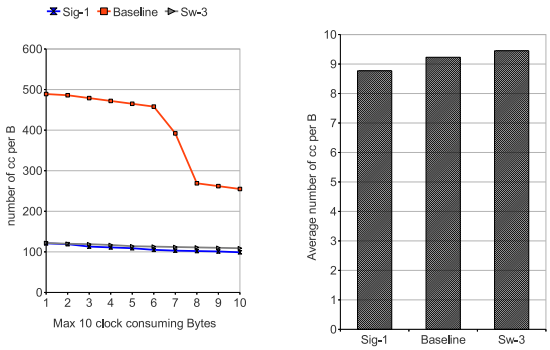


(a) Worst-case performance (b) Average-case Performance

Fig. 13. Comparison Results for Week3 Trace

than 2 order of magnitude. This consequently contributes to the exponentially increased storage space.

It is interesting to note that our proposed mechanisms - hardware based and software based mechanisms - are orthogonal. These mechanisms can also be combined using an FSM constructed with an upper bound failure chain length and a parallel FSM traversal. However, we observe no significant worst-case or average-case performance improvement. Further, the combined scheme also needs additional storage space.



(a) Worst-case performance (b) Average-case Performance

Fig. 14. Comparison Results for Honeypot Trace

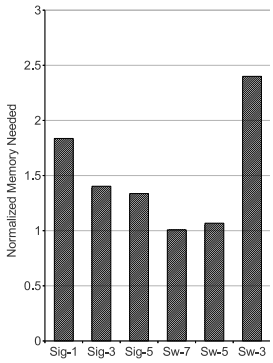


Fig. 15. Storage Space Comparison.

7 Related Work

To the best of our knowledge, Crosby et al[6] were the first to introduce attacks targeting the worst-case performance. They exploited weaknesses in the hash tables used for port scanning in the Bro IDS[16]. A hash table needs O(n) time for insertion on an average and O(n^2) in the worst-case. They carefully construct packets that cause collision in the hash table. In this manner, the performance of the hash table is significantly degraded. As a counter measure, they proposed the use of universal hash functions that significantly reduces collisions.

Smith et al[21] present algorithmic complexity attacks that exploit syntactics of rule specification. There are rules in Snort that are dependent on the relative position of bytes in the packet. They exploited this dependency to create packets that lead to multiple repeated and often redundant processing of the same

byte. So they propose a memoization based technique to prevent such redundant processing of bytes.

Earlier works in this direction have focused on either compacting the FSM or on improving the system throughput. To compact the FSM, Kumar et al[12] used a Delayed input DFA (D^2FA). A DFA is very similar to the FSM studied in this paper. They observed that a DFA typically has numerous states with identical outgoing transitions. So they remove this redundancy using a default transition. This transition is very similar to the failure pointer studied in this paper. So our proposed architecture and traversal complements the D^2FA in improving its worst-case performance.

Tuck et al[25] study different optimizations to reduce the size of each node in the FSM. They use a 256 bit bitmap for each node in the FSM. A bit is set in the bitmap if the corresponding character is an outgoing edge. They further compact the FSM using the failure pointer optimization as discussed earlier. Hence our proposed traversal and architecture is directly applicable to this work.

Becchi et al[2] propose state merging for reducing the storage space. Two states are similar if they have multiple common output states. They combine such states to form a compact FSM. Interestingly, they use the bit mapped based implementation of Tuck et al [25] for representing states. So our proposed architecture is directly applicable to it. Song et al[23] propose using a cached DFA (CDFA) for efficient traversal. In a CDFA, a cached state is used to eliminate 1-step transitions. Among the mechanisms they investigate for compacting the FSM, they also include failure pointer optimization as discussed earlier. So again our proposed architecture is directly applicable to this work.

In addition, there have been numerous works that study a rich variety of DoS attacks. A taxonomy of DoS attacks is given in[13]. Moscibroda et al[15] study DoS attacks against DRAM scheduling in multi-cores. They observe that a malicious application can starve other benign applications, thus leading to significant performance degradation. So they propose a memory architecture that provides fairness to all executing applications. Cai et al[4] study algorithmic complexity attacks against the Unix file system. So in this attack a malicious system process tricks the OS to access system files that are not in its access privileges. They propose a defense mechanism that is provably secure. Hasan et al[10] study DoS attacks that forcefully heat up certain resources in a SMT. In this attack, a malicious thread creates a hot spot in a shared resource by repeatedly accessing it. They study several mechanisms to mitigate the hot-spot including selective throttling of threads.

8 Conclusion

In this paper, we have presented a counter-measure for a performance throttling attack against the string matching algorithm in an IDS. Our study reveals that with certain input bytes, the Aho-Corasick algorithm can end up traversing a chain of up-to 31 pointers. Our results indicate a massive performance degradation, a 22X fall in comparison to the average case performance. We investi-

gate two mechanisms to counter this performance degradation - hardware-based mechanism and software-based mechanism. In the hardware-based mechanism we identify the candidate pointer from the chain of pointers and directly jump to it. We propose a parallel architecture for FSM traversal. The signature matching engine identifies the pointer to jump to, while the FSM engine performs the regular FSM traversal. In the software-based mechanism, we propose a modified FSM that restricts this chain of sequential pointer traversal to a fixed upper bound. Both these scheme result in over 3X improvement in the worst-case performance.

An applicability of this work is in detecting tampering of the Snort signature database. If an adversary corrupts the memory stack of the IDS using buffer overflow attempts, then the pattern matching module can be compromised. In order to detect such tampering, the hardware-based mechanism needs to be extended for detecting FSM traversal violations. Performance throttling attack is an example of an evasion attempt, there are other ways of evasion including packet re-assembly and packet fragmentation. In both of these attacks, the adversary can force the IDS to maintain an infinite number of states (TCP connections) that finally leads to memory exhaustion. Under this circumstance, even benign packets suffer massively. It will be interesting to study defense mechanisms against these attacks.

Acknowledgements

This work has been supported by the following grants: TIN2010-18368, TIN2007-61763, and SGR2009-1250. We are grateful to the Spanish Ministry and Intel Corporation for providing us the requisite monetary and logistic support.

References

[1] A. V. Aho, M. J. Corasick. *Efficient String Matching: An Aid to Bibliographic Search.* Communications of the ACM, 18(6):333340, 1975.

[2] M. Becchi and S. Cadambi. *Memory-Efficient Regular Expression Search Using State Merging.* Proceedings of INFOCOM 2007, 2007.

[3] B. H. Bloom. *Space/time Trade-offs in Hash Coding with Allowable Errors.* Communications of the ACM, v.13 n.7, p.422-426, 1970

[4] Q. Cai, Y. Gui, and R. Johnson. *Exploiting Unix File-system Races via Algorithmic Complexity Attacks.* Proceedings of IEEE Symposium on Security and Privacy, 2009.

[5] Cisco Inc. *The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor.*
http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.html.

[6] S. A. Crosby and D. S. Wallach. *Denial of Service via Algorithmic Complexity Attacks.* In USENIX Security, 2003.

[7] Defcon, http://www.defcon.org.

[8] J. Edler, and M. D. Hill. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*, http://www.cs.wisc.edu/markhill/DineroIV.

[9] . M. Handley, V. Paxson, and C. Kreibich. *Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-end Protocol Semantics.* In Proceedings of the 10th USENIX Security Symposium, 2011.

[10] J. Hasan, A. Jalote, T. N. Vijaykumar and C. E. Brodley. *Heat Stroke: Power-Density-Based Denial of Service in SMT.* In Proceedings of HPCA, 2005.

[11] Intel Corporation. *Intel IXP 2400 Network Processor Hardware Reference Manual*, Revision 7, 2003.

[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. *Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection.* ACM SIGCOMM 2006.

[13] J. Mirkovic and P. Reiher. *A Taxonomy of DDos Attack and DDos Defense Mechanisms.* In ACM SIGCOMM Computer Communications Review, pages 39-53, volume 34, 2004.

[14] MIT Lincoln Labs, DARPA Intrusion Detection Evaluation. http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/.

[15] T. Moscibroda, O. Mutlu. *Memory Performance Attacks: Denial of Memory Service in Multi-core Systems.* In 16th USENIX Security Symposium, pages 1-18, 2007.

[16] V. Paxson. *Bro: a System for Detecting Network Intruders in Real Time.* In Computer Networks 31(23-24):2435-2463, 1999.

[17] F. Pouget, M. Dacier, and P. Hau. *Leurre.com: On the Advantages of Deploying a Large Scale Distributed Honeypot Platform.* E-Crime and Computer Conference, 2005.

[18] T. Ptacek and T. Newsham. *Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection.* In Secure Networks, Inc., 1998.

[19] M. Roesch. *SNORT - Lightweight Intrusion Detection for Networks.* In LISA '99: USENIX 13th Systems Administration Conference, 1999.

[20] G. S. Shenoy, J. Tubella, and A. Gonzalez. *A Performance and Area Efficient Architecture for Intrusion Detection Systems.* In Proceedings of the 25th IEEE International Conference on Parallel and Distributed Processing Symposium (IPDPS), 2011.

[21] R. Smith, C. Estan, and S. Jha. *Backtracking Algorithmic Complexity Attacks against a NIDS.* In ACSAC 2006.

[22] R. Smith, C. Estan, and S. Jha. XFA: *Faster Signature Matching with Extended Automata.* IEEE Symposium on Security and Privacy, 2008.

[23] T. Song, W. Zhang, D. Wang, and Y. Xue. *A Memory Efficient Multiple Pattern Matching Architecture for Network Security.* In Proceedings of IEEE Infocom, 2008.

[24] S. Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. *CACTI 5.1. Technical Report* HP-2008-20, HP Labs, 2008.

[25] N. Tuck, T. Sherwood, B. Calder, G. Varghese. *Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection.* In Proceedings of the IEEE Infocom, 2004.