Hardware/Software Mechanisms for Protecting an IDS Against Algorithmic Complexity Attacks

Govind Sreekar Shenoy*, Jordi Tubella* and Antonio González*†
*Department of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, Spain.
†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. An IDS like the popular Snort[17] detects attacks on the network using a database of previous attacks. So in order to detect these attack strings in the packet, Snort uses the Aho-Corasick algorithm. This algorithm first constructs a Finite State Machine (FSM) from the attack strings, and subsequently traverses the FSM using bytes from the packet. We observe that there are input bytes that result in a traversal of a series of FSM states (also viewed as pointers). This chain of pointer traversal significantly degrades (22X) the processing time of an input byte. Such a wide variance in the processing time of an input byte can be exploited by an adversary to throttle the IDS. If the IDS is unable to keep pace with the network traffic, the IDS gets disabled. So in the process the network becomes vulnerable. Attacks done in this manner are referred to as algorithmic complexity attacks, and arise due to weaknesses in IDS processing.**

In this work, we explore defense mechanisms to the above outlined algorithmic complexity attack. Our proposed mechanisms provide over 3X improvement in the worst-case performance.

***Keywords*-Intrusion Detection Systems, Defense Mechanisms, Hardware Support.**

I. Introduction

Intrusion Detection Systems (IDS) protect the network against malicious activities. By monitoring the traffic in real time, an IDS detect and acts against intrusion attempts. There are broadly two types of Intrusion Detection Systems, an anomaly-based IDS and a misuse detection IDS. An anomaly-based IDS functions by detecting anomalous system behaviour. A common example of an anomalous behaviour are abnormal series of system calls in stack smashing[12]. So an anomaly-based IDS learns the system behaviour[21], and then detects an intrusion attempt. On the other hand, a misuse-detection IDS uses a database of past reported attacks. This database of attacks can be viewed as attack strings, and an IDS like Snort[17] searches for these attack strings in the packet.

An issue however is the huge database of attack strings. For example, the April-2010 Snort release contains more than 40,000 attack strings. Hence, searching for >40,000 attack strings in the packet payload is computationally very intensive. We observe that the pattern module consumes more than 60% of the execution time in Snort. Thus detecting attack strings in the payload is a performance bottleneck.

An IDS commonly uses the Aho-Corasick algorithm[1] to detect attack strings in the packet payload. This algorithm functions by constructing a Finite State Machine (FSM) from the attack strings. Later the FSM is traversed using the payload bytes. The main advantage in using the Aho-Corasick algorithm is that it guarantees a linear time search irrespective of the number of strings. The issue, however, lies in devising a practical implementation The FSM constructed by the algorithm gets very bloated in terms of the storage size. This also affects the performance efficiency of the IDS due to the large working set size.

Earlier works in this direction[2, 4, 11, 18, 20, 22, 23, 25] have studied mechanisms to compact the FSM. A common optimization that significantly reduces the FSM storage size is the removal of failure edges[2, 11, 18, 22, 25]. We observe that 93% of the edges in the FSM are failure edges. So the removal of these failure edges provides important storage benefits. However, the failure edge optimization incurs a processing overhead, namely, that of traversing a series of FSM states (also viewed as pointers). We observe that in the worst-case, the processing of a payload byte needs a traversal of up-to 31 pointers. Such a chain of pointer traversal incurs a huge performance drop (22X) in comparison to the average-case. This wide variance between the average-case and worst-case performance can be exploited by an adversary to throttle an IDS. If the IDS is unable to process packets at the incoming rate, then the IDS is disabled. Thus the network becomes vulnerable and open to attacks. Such attacks that exploit weaknesses in IDS processing are termed as algorithmic complexity attacks.

In this work we present a defense mechanism to the above outlined algorithmic complexity attack. Our countermeasure focuses on improving the worst-case performance. We propose two mechanisms - hardware and software mechanisms - to counter the performance drop. In the hardware mechanism, we identify the candidate pointer from the chain of pointers and directly jump to it. We use bloom-filters for identifying the failure pointer. In the software-based mechanism, we propose a modified FSM that places a fixed upper bound on the chain of pointers traversed. Both these scheme result in over 3X improvement in worst-case

performance.

The rest of this paper is organized as follows. Section II provides a brief background on the Aho-Corasick algorithm. Section III presents the motivation of this work. Section IV details our proposed counter-measure. The simulation methodology is discussed in Section V, and Section VI presents the performance results. Section VII discusses the related work in this area. Section VIII provides future directions.

II. Background

In order to identify malicious traffic, an IDS uses signatures of existing attacks which are compared with the payload bytes. Signatures can also be viewed as strings and there are >40,000[1] strings in Snort. So multi-string matching needs to be done on the packet payload. Snort uses the Aho-Corasick algorithm[1] for multiple string matching. 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: **abcdee, bcdeg, cdeh, dei, ek** . Figure 1 shows the corresponding Aho-Corasick FSM constructed from these strings. The FSM is built in two

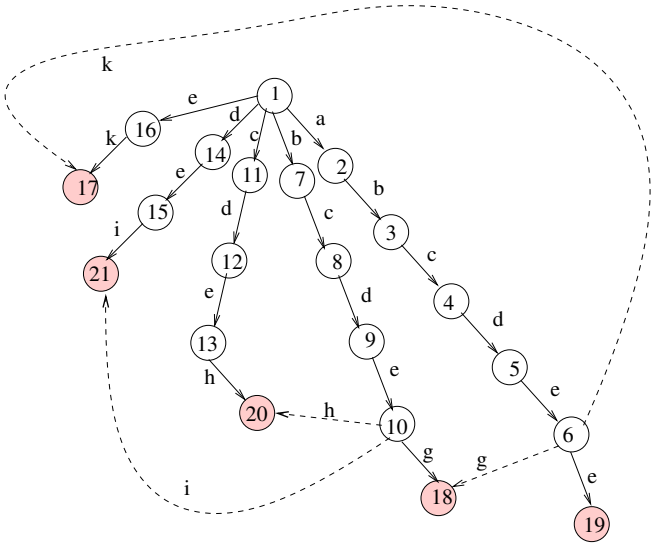


Figure 1. Example of the Aho-Corasick Finite State Machine.

stages. In the first stage, characters from the strings 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. Also note that **nodes 19, 18, 20, 21, 17** indicates a match for strings **abcdee, bcdeg, cdeh, dei, ek** respectively. 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 6** and its edge **g**. This is a failure edge and is identical to the edge from **node 10**. In fact, on a closer look, all failure edges reveal a similar characteristic. So **Node 10** can be viewed as a failure pointer of **node 6**, and this traversal can also be done by jumping to **node 10**. In this way all failure edges are eliminated.

Figure 2 shows the FSM built with the failure pointer optimization. We observe that 93% of the edges are failure edges, and so the failure pointer optimization provides important area benefits. So earlier works[2, 11, 18, 20, 22, 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[15] also use regular expressions to specify attack strings. These regular expressions are again converted to DFAs or NFAs. So this work is equally applicable to regular expressions used in an IDS.

III. Motivation

The optimization of failure edges 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 be **a, b, c, d, e, k**. The first 5 bytes lead up-to **node 6**. For the final byte, **k**, the failure pointer needs to be traversed as there are no matching edges at **node 6**. Hence, **node 10** - the failure pointer of **node 6**- is accessed. Here again there are no matching edges, and so the failure pointer of **node 10** 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 pointers are accessed sequentially and sometimes wastefully. This can lead to significant performance degradation when large such chains are visited.

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

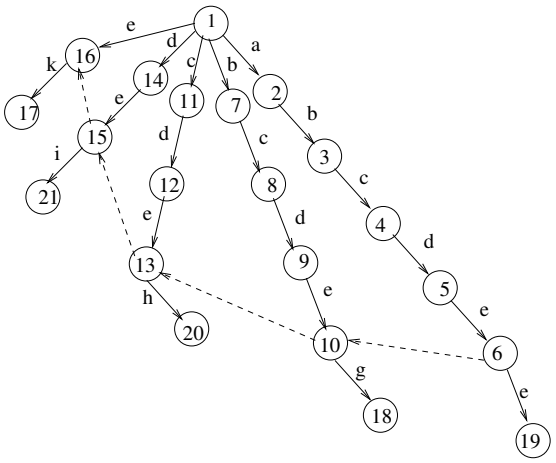


Figure 2. Optimized Aho-Corasick FSM

Figure 3 shows the CDF of processing time required per input byte. We see that 95% of input bytes need less than **31** clock cycles. However it is interesting to note that there are bytes that need up-to **516** clock cycles. This clearly indicates that there is a wide variation in processing time. We observe

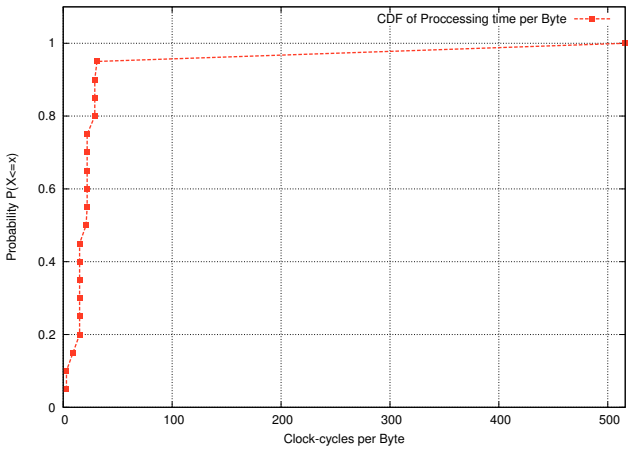


Figure 3. Processing Time CDF

that the cause of the enormous processing time is due to the traversal of a chain of 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 algorithmic complexity attacks. Hence it is important to accelerate the failure pointer traversal.

IV. Proposed Counter-measure

Intrusion detection systems are commonly deployed in routers. Routers in turn can use specialized hardware for detecting attack strings in the payload bytes. Alternatively, an IDS can also be deployed in commodity processors. So we propose two mechanisms, a hardware mechanism and a software mechanism, for improving the worst-case performance.

A. Hardware Mechanism

The processing takes a performance hit due to the chain of failure pointers traversed. On the other hand, if the candidate failure pointer to jump to is directly identified, then this performance hit can be reduced. We first describe a mechanism to identify the candidate failure pointer, and to incorporate it in the traversal algorithm.

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 pointers. So we break this chain of traversal into a chain of comparing outgoing edges. We illustrate this more clearly with an example. Let the input bytes to the FSM in Figure 2 be **a, b, c, d, e, k**. The first 5 bytes lead up-to **node 6**. For the final byte, **k**, since there are no matching edges, the failure pointer is traversed. The failure pointer of **node 6**, **node 10**, is traversed. Since it is a mismatch, the failure chain is followed until **node 16**.

So the main operation in the failure pointer traversal is the comparison of the input byte with 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 6** (from Figure 2), we create and store bloom filter signatures for all its failure chains, namely, **nodes 10, 13, 15, 16**. These signatures are generated using the outgoing edges of the corresponding node.

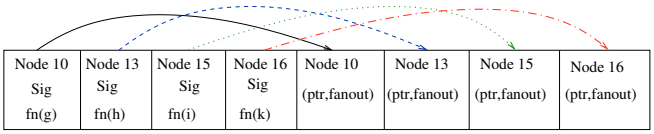


Figure 4. Node 6 Signature Storage.

Figure 4 shows the signature storage of **node 6** generated using this way. 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 is as follows. Consider the above example of traversing **node 6** with **k**. Since there are no matching edges in **node 6**, we check if there are any matching edges in the failure chain. A signature is generated using **k**, and compared

against all the failure chain signatures of **node 6**. Since **node 16** has a matching signature, we directly traverse to **node 16**. 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.

In the design of bloom filters, a critical parameter is the false positive rate of the bloom filter. So we study the bloom filter signature generation in detail.

The bloom filter is on the critical path since a signature needs to be generated for every payload byte. So the bloom filter signature generation needs to be computationally simple and efficient. Hence we select bits from the failure edge for generating the bloom filter signature. In particular 5-bits are selected from the failure edge and indexed into a 4B word. The bit thus indexed is also set. This process is repeated with a different 5-bit pattern. The 4B word thus created is the signature for the failure pointer.

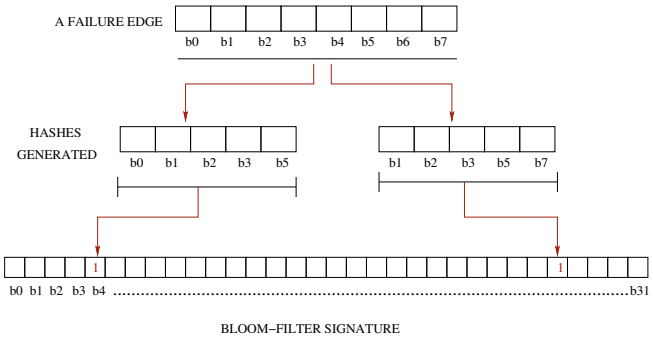


Figure 5. Bloom-filter Signature Generation.

Figure 5 outlines the steps in the signature generation for a failure edge. In order to check whether the input byte matches a failure edge, this process is repeated for the payload byte. The signature thus generated is compared with the signature stored for the failure pointer. If the signature matches, then the failure pointer is traversed.

The main advantage in using bloom filters is the absence of false negatives. However, bloom filters do encounter false positives. We observe a false positive rate of 15% with the above design of 2 hash functions (bit selection) and 4B wide signature. Furthermore, there is a negligible impact on the false positive rate when the number of hashes and/or the signature size is increased. So we use 2 hash functions and 4B wide signatures.

The failure chain signature matching can be performed independently and in parallel with the conventional node processing. So we identify the candidate failure pointer concurrently with the conventional node processing. If there is no need to traverse the failure pointer, then the failure pointer identification is discarded.

B. *Software Mechanism*

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

We illustrate it more clearly with an example. Consider the failure pointer optimized FSM shown in Figure 2. If we use a threshold value of **3**, then failure edges are inserted for node **10**. In this way we limit the failure chain traversal to a threshold. Figure 6 shows the FSM thus constructed with an upper bound.

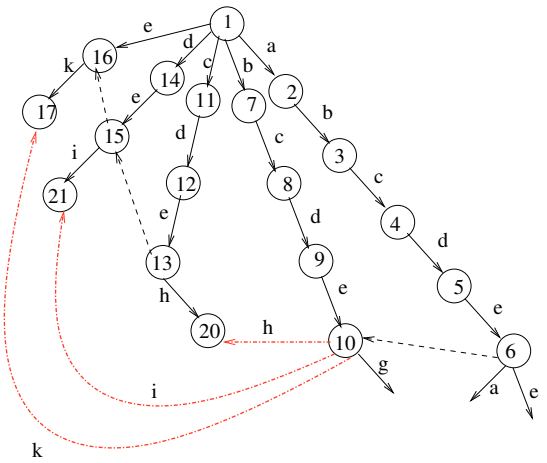


Figure 6. Software Mechanism.

V. Simulation Methodology

We evaluate the performance of our proposed mechanisms and compare it with the conventional method of sequential failure pointer traversal[18]. We have used a publicly available attack trace, the Defcon trace, for our evaluation. This is a trace captured from the Capture the flag (CTF) game[8]. CTF is a hacking contest in the Defcon conference[6]. The objective of this contest is to break into computers of other teams, while at the same time preventing others from do so.

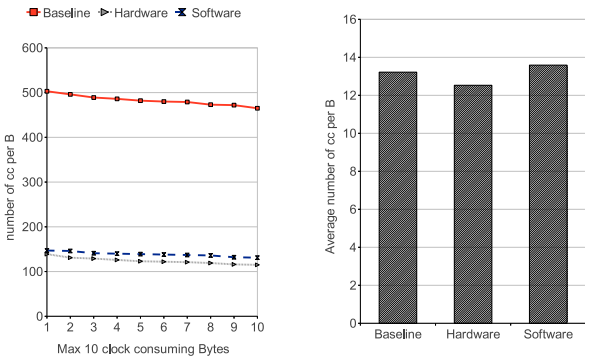
We have used the Snort database released on April 2010 and containing 40,678 strings. We also report results for the Snort database September 2007 release containing 23,653 strings.

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[18], and the signature computation time. The **total memory access time** is obtained from the trace-driven cache simulator [9], which is

modified to model cache access times and processing times. The cache miss penalty is obtained from CACTI [24] by plugging the FSM memory size into the SRAM model. We have used a 16k direct-mapped cache-configuration for the caches. The cache hit time of 2 cc is used (also obtained from CACTI). The core frequency is assumed to be 3 GHz.

VI. Results

We compare the performance of our proposed architecture with the **Baseline**[18]. The **Baseline** performs traversal using the conventional way of sequentially following failure pointers. In order to evaluate the worst performance cases, we compare the 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. Figure



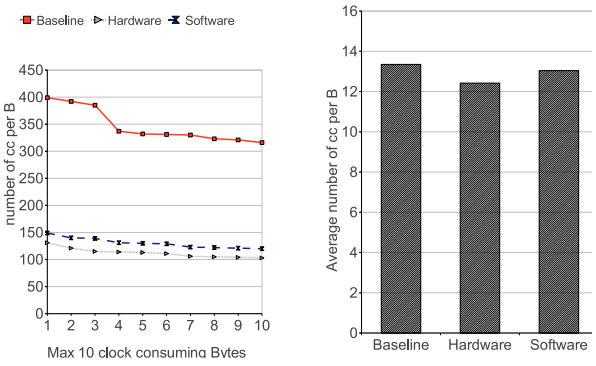
(a) Worst-case performance (b) Average-case Performance

Figure 7. Performance Comparison with the April-2010 Database

7(a) shows the 10 most clock consuming bytes for the Defcon trace. While **Baseline** needs at least **503 cc**, the worst-case performance in the **Hardware Mechanism** is **139 cc**. Furthermore, in the **Software Mechanism** it is **147 cc**. So both these mechanisms reduce the worst-case performance by over 3X magnitude. Figure 7(b) shows the average-case performance. We observe that the average-case performance remains largely unaffected, though there is a mild performance improvement in using the **Hardware Mechanism**. Figure 8 shows the performance comparison with the September-2007 database, and we again observe a very similar performance behaviour.

VII. Related Work

To the best of our knowledge, Crosby et al[7] were the first to introduce attacks exploiting the algorithmic complexity. They exploited weaknesses in hash tables used for port scanning in the Bro IDS[15]. 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 collisions in the hash table. Thus approaching the worst-case performance.



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

Figure 8. Performance Comparison with the September-2007 Database

As a counter-measure, they proposed the use of universal hash functions which significantly reduces collisions.

Smith et al[19] 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 redundant processing of the input byte. So they propose a memoization based technique to prevent such redundant processing of bytes.

Earlier works on FSM for IDS have focused on compacting the FSM and also on improving the system performance. To compact the FSM, Kumar et al[11] used a Delayed input DFA (D^{2}FA). 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^{2}FA 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. So 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[22] 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, also includes the failure pointer optimization. So again our proposed architecture is

directly applicable to this work.

Algorithmic complexity attacks can also be viewed as a class of DoS attacks that target system resources. A taxonomy of DoS attacks is given in [13]. Moscibroda et al[14] study DoS attacks against the 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[5] 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 heats 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 mitigate the hot-spot by selectively throttling the *hot* thread.

VIII. Conclusion

In this paper, we present an algorithmic complexity attack against the string matching algorithm used in an IDS. We observe that certain input bytes result in a traversal of a chain of 31 pointers. Our results indicate a massive performance degradation in the processing of these bytes, a 22X performance drop. We investigate two mechanisms for countering this performance degradation. In the hardware mechanism we identify the candidate pointer from the chain of pointers and directly jump to it. We use bloom-filters for identifying the failure pointer. In the software mechanism, we propose a modified FSM that restricts this chain of pointer traversal to a fixed upper bound. Both these scheme result in over 3X improvement in the worst-case performance.

Algorithmic complexity attack is an instance of an evasion[16] 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. So defense mechanisms can be explored against such attacks.

Acknowledgements

This work has been supported by the Generalitat de Catalunya under grant 2009SGR-1250, the Spanish Ministry of Science and Innovation under grants TIN2007-61763 and TIN 2010-18368 and Intel Corporation.

References

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

[2] M. Becchi and S. Cadambi. Memory Efficient Regular Expression Search Using State Merging. *In Proceedings of the 26th IEEE Infocom*, 2007.

[3] B. H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 1970.

[4] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. *In Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[5] X. Cai, Y. Gui, and R. Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. *In Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[6] Defcon Conference. http://defcon.org.

[7] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. *In Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, 2003.

[8] Defcon-8, Capture-the-Flag Game Traces. http://cctf.shmoo.com.

[9] J. Edler and M. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. Technical report, Univ Wisconsin at Madison. http://www.cs.wisc.edu/~markhill/DineroIV.

[10] J. Hasan, A. Jalote, T. Vijaykumar, and C. Brodley. Heat stroke: Power-density-based Denial of Service in SMT. *In Proceedings of the 11th High Performance Computer Architecture*, 2005.

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

[12] W. Lee and S. J. Stolfo. Data Mining Approaches for Intrusion Detection. *In Proceedings of the 7th USENIX Security Symposium*, 1998.

[13] J. Mirkovic and P. Reiher. A Taxonomy of DDoS attack and DDoS Defense Mechanisms. *SIGCOMM Compuer. Communication Review*, 2004.

[14] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. *In Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[15] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. *In Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, 1998.

[16] T. Ptacek and T. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks Inc, 1998.

[17] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. *In Proceedings of the 13th USENIX conference on System Administration*, 1999.

[18] G. 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, 2011.

[19] R. Smith, C. Estan, and S. Jha. Backtracking Algorithmic Complexity Attacks against a NIDS. *In Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006.

[20] R. Smith, C. Estan, and S. Jha. XFA: Faster Signature Matching with Extended Automata. *In Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008.

[21] R. Sommer and V. Paxson. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. *In Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[22] T. Song, W. Zhang, D. Wang, and Y. Xue. A Memory Efficient Multiple Pattern Matching Architecture for Network Security. *In Proceedings of 27th IEEE Infocom*, 2008.

[23] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. *In Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[24] T. Thozhiyoor, N. Muralimanohar, J. Ahn, and N. Jouppi. CACTI 5.1. Technical Report. Technical Report HP-2008-20, HP Labs, 2008.

[25] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. *In Proceedings of the 23rd IEEE Infocom*, 2004.