A Scalable Architecture for Countering Network-Centric Insider Threats

Faisal M. Sibai
Volgenau School of Engineering
George Mason University
Fairfax, VA 22030, USA
Email: fsibai@gmu.edu

Daniel A. Menascé
Dept. of Computer Science, MS 4A5,
George Mason University
Fairfax, VA 22030, USA
Email: menasce@gmu.edu

***Abstract*—Dealing with the insider threat in networked environments poses many challenges. Privileged users have great power over the systems they own in organizations. To mitigate the potential threat posed by insiders, we introduced in previous work a preliminary architecture for the Autonomic Violation Prevention System (AVPS), which is designed to self-protect applications from disgruntled privileged users via the network. This paper extends the architecture of the AVPS so that it can provide scalable protection in production environments. We conducted a series of experiments to asses the performance of the AVPS system on three different application environments: FTP, database, and Web servers. Our experimental results indicate that the AVPS introduces a very low overhead despite the fact that it is deployed in-line. We also developed an analytic queuing model to analyze the scalability of the AVPS framework as a function of the workload intensity.**

Keywords-insider threat; scalability; network security.

I. INTRODUCTION

Defeating the insider threat is a very challenging problem in general. An insider is a trusted person that has escalated privileges typically assigned to system, network, and database administrators; these users usually have full access and can do almost anything to the systems and applications they own. Users with escalated privileges within an organization are trusted to deal with and operate applications under their control. This trust might be misplaced and incorrectly given to such users. It is extremely difficult to control, track or validate administrators and privileged user actions once these users are given full ownership of a system. The recent disclosure by Wikileaks of U.S. classified embassy foreign policy cable records provides a perfect example of an insider attack [1]. In this disclosure, an insider with unfettered access to data at his classification level was able to access data over a secure network using laptops that had functional DVD writers. Our approach to mitigate the insider threat allows for users or groups of users to be treated differently despite having the same classification level [2]. The approach limits and controls network access through an in-line component that checks access to specific applications based on policies that can be as specific or granular as needed.

In our prior work, we introduced a framework that self-protects networks in order to mitigate the insider threat [2]. The framework, called AVPS (Autonomic Violation Prevention System), controls and limits the capabilities provided to administrators and privileged users in organizations. AVPS concentrates entirely on detecting and preventing usage policy violations instead of dealing with viruses, malware, exploits, and well-known intrusions. In our implementation, the AVPS monitors events and takes actions for conditions that occur, as specified by Event-Condition-Action (ECA) commonly used in security-centric systems and autonomic computing [3]. Our prior work does not address scalability though.

The design of the AVPS architecture must consider scalability, manageability, application integration, ease of use, and the enforcement of separation of duties. There has been prior work in this area at the application, host, and network levels [4], [5], [6], [7], [8]. The previous methods have applied self-protecting capabilities by either considering single applications on the host or more towards vulnerabilities, malware, exploits and traditional threats.

This paper significantly extends our previous work ([2]) in that it presents a scalable AVPS architecture and supports its design with experimental results and theoretical queuing modeling. We present here the results of experimental evaluations of the AVPS architectures as well as the analysis of its performance overhead on three different types of application servers: FTP server, database server, and web server. We specifically measured the average throughput, average transfer time, average CPU utilization, and provided 95% confidence intervals for all three measurements. We also used a queuing theoretic analytic model to predict the scalability of AVPS for different workload intensity values for these three types of applications

The rest of the paper is organized as follows. Section II presents some of the major challenges and requirements faced in the design of AVPS. The next section presents a scalable architecture for the AVPS framework. Section V presents an experimental evaluation and a full performance and scalability analysis of AVPS. Finally, Section VI presents the conclusion, final remarks, and future work.

II. CHALLENGES AND REQUIREMENTS

The following major challenges play a primary role in the success of the AVPS framework: scalability in production en-

 ISBN: 978-1-61208-010-9

vironments, support for encrypted network traffic, integration with multiple types of application servers on the network, and ease of deployment in large production environments. This paper mainly addresses scalability and performance issues and sheds some light on all four challenges.

Scalability is an absolute requirement for production environments. The AVPS solution is an in-line solution that intercepts every single packet that traverses the local area network that is destined to an application server. Therefore, it could become a focal point and a possible bottleneck. The primary goal of our solution is to scale with growing network and application demands. The AVPS architecture should allow for horizontal scaling to cope with high-volume environments. This requirement is further discussed in more detail in the following sections.

Encryption is another important challenge in the design of our solution. SSH and SSL are widely used in local area networks for information retrieval and administration of applications and devices. The AVPS performs packet inspection on some or all (depending on the application) packets that pass through it. This poses a challenge that is handled in our solution through one of the following methods: (1) decrypting the traffic that passes through the AVPS and then re-encrypting it for delivery to its destination using viewSSLd [9] or netintercept [10] for example, (2) completely off-loading the encryption/decryption requirements to external hardware-based devices that sit before and after the AVPS, or (3) decrypt the traffic by having a legitimate man-in-the-middle host that decrypts and re-encrypts the traffic and delivers it to the destination [11]. This paper does not discuss encryption in any further detail.

Application server integration is also extremely important. With the wide range of applications deployed in production environments, the AVPS framework must be capable of interpreting and understanding requests and responses that it intercepts. The AVPS is based on intercepting, not necessarily inspecting, every single packet initiated by a host that is delivered from and to an application. This makes application integration completely possible and achievable. Policies deployed on the AVPS are customizable to the desired granularity level and types of attributes (e.g., from very generic, such as IP or user level, to very specific, such as IP, user, application type, request, and response). Thus, it is completely up to the AVPS owner to specify the granularity of what should be inspected and what should be ignored.

Finally, the successful deployment of AVPS in large environments is crucial. The AVPS solution should be easy to deploy and maintain and should be capable of handling heavy traffic loads. Current environments have hundreds if not thousands of servers with networks that are capable of handling and processing 100 to 1000 Mbps of traffic. A solution that handles thousands of servers through a handful of clustered AVPS compute nodes is part of the architecture discussed in the remaining sections of this paper.

III. Scalable AVPS Architecture

For the AVPS to achieve its goal of solving the insider threat problem, it must be placed in-line between clients and internal application servers. This way, the AVPS is capable of intercepting every single packet that flows from clients to applications and back in order to take the correct actions when a rule in a policy is matched.

Figure 1 depicts the architecture of the AVPS framework. Performance and high availability are extremely important since the AVPS is located between the clients and the application servers. Traffic coming from a pool of M clients goes through a load balancer that handles incoming requests. The load balancer forwards the traffic to one of N AVPS engines that process and inspect the incoming traffic. The AVPS engines compare traffic policies that contain rules and actions on how to handle traffic. The policies are stored on a database local to the AVPS engine or on an external database shared by all AVPS engines. Events are stored on a centralized database. Actions are taken on traffic once a rule in a policy has been matched. Examples of possible AVPS actions include dropping, blocking, or replacing traffic as it traverses the engine on its way to application servers. Let there be K different types of applications servers (e.g., FTP server, database server, Web server).

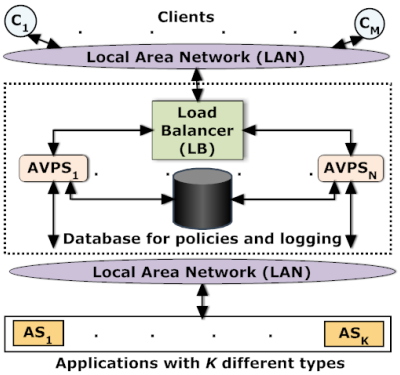

Fig. 1: Architecture of the AVPS framework.

Figure 2 depicts the steps taken by the AVPS engine. Traffic is first received by a layer 2 bridge that is responsible for handling incoming and outgoing traffic. Traffic is then forwarded to the normalization and processing module where packets are broken down into pieces that can be matched against rules. Traffic is then matched against policies and rules that are pre-loaded into memory. If there is a rule match, an event or action is generated. Finally, if an event or action occurred, it is logged into a database.

As an example of the advantage of using the AVPS architecture, consider a scenario with multiple database servers scattered over a large geographically distributed network. Assume that a *top secret* table is replicated in every database server and that we want to have fine access control to this table. Using conventional access control methods, we would be able to limit specific users or roles from accessing the table. This would

 ISBN: 978-1-61208-010-9

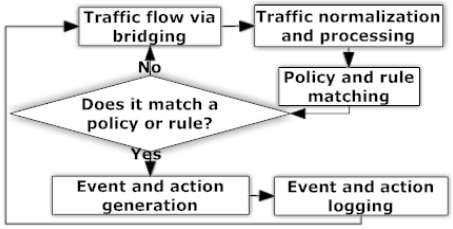

Fig. 2: Steps of the AVPS engine.

require manually setting these controls on every database server. This approach has several drawbacks: (1) Manually setting access controls into each server is time consuming and might have a high error rate. (2) This method requires an administrator to know all of the DB servers that live on the network; newly installed DB servers or even covert ones may be missed. (3) The DB owner actually does the changes with no oversight, which contradicts the separation-of-duties concepts. (4) Last but not least, it would be almost impossible with traditional access control methods to limit access for a specific population of administrators or privileged users, coming from a specific location on the network, accessing the information at a specific time and targeting a specific table.

The AVPS is designed to block detected violations that match specific rules in a policy. Therefore, the AVPS reduces or possibly completely eliminates the drawbacks listed above. The AVPS is also tamper resistant. It enforces a separation-of-duties policy, i.e., the primary application system owner has no control over the AVPS policies [2]. The AVPS can be deployed to carry insider and regular user traffic or to only carry insider traffic. The proper deployment depends on how the network is setup and on how the network is segmented.

Emerging technologies, such as new network TAPs (e.g., Network Critical V-line TAP [12]), that can handle 1/10 Gpbs traffic and allow in-line functionality without introducing a single point of failure, make systems such as AVPS possible to implement without fault-tolerance concerns.

IV. AVPS VS. OTHER SOLUTIONS

Our prior work [2] distinguishes the AVPS from other systems such as IPS, Firewalls, Host based IPS and Network Admission Control/Network Access Control (NAC). We use Intrusion Prevention Systems (IPS) and Intrusion Detection Systems (IDS) in this paper interchangeability. The only difference between the two is that IDS is considered a passive network monitoring system and IPS is considered an active\inline network monitoring system. Traditional IDS/IPS systems tend to concentrate on users that do not have access to the system and try to exploit, hack, or crack into it. Other enhanced IPS/Firewall systems such as IBM Proventia [13] or Cisco ASA [14] do have enhanced context-aware security but lack insider threat defeating capabilities. The AVPS, on the other hand, is designed with the insider threat in mind. Our current AVPS implementation relies on well-known methods used in traditional IDS/IPS for the detection of insider attacks. In our in-progress work we are working on enhancing the detection capabilities of the engine to incorporate self-learning/self-adaptation rule learning, enhance application integration and interaction, include user roles and responsibilities and have better session management and detection capabilities which current IDS/IPS systems either lack or have weak functionality.

V. PERFORMANCE ASSESSMENT OF AVPS

This section presents an experimental evaluation of the AVPS in a controlled environment. We describe the experimental testbed, analyze the results, and present a scalability analytical model based on the M/M/N//M queuing model.

A. Experimental Environment

We based our experiments on three different applications: FTP, database, and Web server. The specification of the environment and the experimental testbed is shown in Fig. 3.

In this environment, the client requests services from application servers, which respond to the requests. All traffic between client and server is monitored and inspected by the AVPS. A controlling host controls the environment and collects the results of the experiments (see Fig. 3).

Apache JMeter 2.4 [15] was used on the client to conduct both FTP and Web experiments. We measured the average throughput and average transfer time in both cases. For the database experiment, mysqlslap [16] was used to measure the average response time.

On the AVPS we used Snort-inline 2.8.6.1 [17]. Snort is highly used in academic IDS/IPS research experiments. Other tools are also used in academic research (e.g., Bro [18] and EMERALD [19]). We used Linux iptables [20], a firewall package installed under RedHat, Fedora, and Ubuntu Linux, in conjunction with Snort in-line to filter packets as they come into the AVPS and leave. We used MySQL 5.1 [21] to store events and event packet captures. We used BASE [22] to query the DB and display the events in the browser.

We configured three different application servers: (1) vsftpd 2.3.2 FTP server [23], (2) MySQL 5.1 DB [21], and (3) Apache 2 Web server [24].

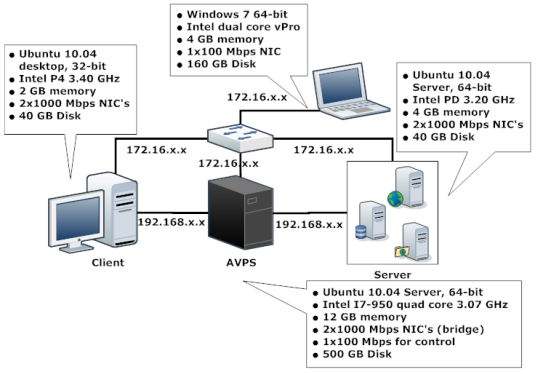

Fig. 3: Experimental environment.

We customized the Snort configuration file to meet the AVPS requirements. All default rules that come with Snort were disabled and our own policies were added inside *local.rules*. We configured Snort to output events into a MySQL database.

The client and server are connected directly to the AVPS as shown in Fig. 3. All three machines are also connected via a second network card to a switch. The controlling host is also connected to the switch to control and collect the results from all three machines.

B. Experimental Results

In this and the following section we show the very little overhead that the AVPS adds in the worst case when traffic passes and is processed by it and alerts are generated. An ideal situation would block the violating traffic without further processing. This causes very little CPU overhead. In this section, we provide measurements for average transfer times and throughputs with 95% confidence intervals, and average and maximum CPU utilization values.

For each application server type, we conducted two types of experiments. The first consisted of manually submitting 10 requests to the application server. This was used to measure the average transfer time, query response time, and throughput. The second consisted of automatically submitting 30 requests to the application server, in sequence with no think time. This process was used to measure the average and maximum CPU utilization of the AVPS engine.

The manual experiments considered the following four scenarios: (1) No AVPS, client and application servers are connected to a 1000-Mbps switch. (2) Client and server are connected to the AVPS but the engine is disabled, traffic is only being bridged. (3) The AVPS is enabled and no rules match the traffic (either because no policies are loaded or because the loaded policies do not trigger a violation). (4) The AVPS is enabled, detects a violation on all rules checked, and generates an alert, which is stored in a database. However, the AVPS is configured not to block the traffic. It should be noted that case (4) above is the one that generates the largest possible overhead because all rules generate a violation, an unlikely event in practice, and traffic flowing through the AVPS is not decreased due to matching offending requests. Thus, all results presented in what follows for scenario (4) represent a worst-case performance scenario.

The automated experiments were used to measure average and maximum CPU utilization of the AVPS engine and consider the following four scenarios: (1) Same as (2) above. (2) Same as (3) above. (3) Same as (4) above. (4) Same as (4) above but the AVPS is configured to block the traffic. Case (3) above is also a worst-case performance scenario for the reasons outlined above. Case (4), the blocking case, is the ideal operational situation. In that case, blocked traffic does not contribute to network and application server load.

The FTP results are discussed in what follows. Table I shows, the measured results for the average throughput (in KB/sec) and average transfer time (in msec) for 10 manually submitted requests using JMeter for four different file sizes: 100 KB, 1 MB, 10 MB and 100 MB. Results include 95% confidence intervals for all file sizes.

File Size →	100 KB	1 MB	10 MB	100 MB
Average throughput (KB/Sec) with 95% confidence interval				
No AVPS, switching	327.0 ± 7.01	2811.9 ± 48.00	9834.2 ± 38.48	13395.3 ± 90.12
AVPS, process not on	331.1 ± 5.30	2754.7 ± 35.57	9984.4 ± 56.32	13539.5 ± 94.95
AVPS, process on but not matching	330.0 ± 5.92	2754.9 ± 45.45	9756.8 ± 29.41	13257.5 ± 57.54
AVPS, matching and policy applied	332.6 ± 4.71	2746.4 ± 34.38	9841.2 ± 77.32	13300.8 ± 78.88
Average transfer time (msec) with 95% confidence interval				
No AVPS, switching	307.2 ± 6.87	365.3 ± 7.16	1043.2 ± 4.17	7647.6 ± 51.59
AVPS, process not on	302.8 ± 5.05	372.3 ± 4.81	1025.9 ± 5.97	7566.4 ± 52.39
AVPS, process on but not matching	304 ± 5.77	372.7 ± 6.5	1049.6 ± 3.16	7725.2 ± 33.3
AVPS, matching and policy applied	301.3 ± 4.44	373.4 ± 4.74	1041.1 ± 8.10	7701.2 ± 44.95
Average CPU utilization (%) with 95% confidence interval				
AVPS - bridging only	0.02 ± 0.01	0.04 ± 0.03	0.05 ± 0.01	0.05 ± 0
AVPS enabled, not matching	0.02 ± 0.01	0.3 ± 0.06	1.44 ± 0.20	2.11 ± 0.06
AVPS enabled, matching, not blocking	0.20 ± 0.06	0.79 ± 0.17	3.90 ± 0.51	6.12 ± 0.17
AVPS enabled, matching, blocking	0.09 ± 0.02	0.12 ± 0.02	0.10 ± 0.02	0.12 ± 0.03

TABLE I: FTP results

In the case where we check against a rule (case (4) in the manual experiments), we loaded into memory the following rule that alerts when user "appserver" tries to log into a specific FTP server.

alert tcp any any → FTPserver any (classtype:attempted-user; msg:"Snortinline Autonomic FTP event";content: " appserver";nocase;sid:2;)

From Table I, we see that the differences, respectively, in average throughput and average transfer time for any of the various file sizes are either statistically insignificant at the 95% confidence level (e.g., for 100 KB and 1 MB files) or are very small (e.g., less than 1.8% different for 10 MB and 100 MB files). This means that there is little or no difference between the case when the AVPS process is disabled (case (2)) and the case where the AVPS engine is enabled and all rules checked generate a violation, but traffic is not blocked (case (4)). This is expected behavior since the AVPS does not inspect packets that contain file data being transferred. It only inspects the initial administration and request commands. Thus, the AVPS has no or very little impact on throughput and transfer time.

For the CPU measurements discussed below, we used the automated submission scenario. We load into memory the following rule that blocks a user when he/she tries to access a specific FTP server using "appserver" by replacing it with "*********".

 ISBN: 978-1-61208-010-9

*alert tcp any any → FTPserver any (classtype:attempted-user; msg:"Snortinline Autonomic FTP block"; content: " appserver"; nocase;replace:"**********";sid:2;)*

Table I shows the measured average CPU utilization of the AVPS engine for 30 automated requests with zero think time using JMeter for four different file sizes: 100 KB, 1 MB, 10 MB and 100 MB. The figure also shows 95% confidence intervals.

Table I shows that the CPU utilization grows linearly with the file size. For large files (e.g., 100 MB) we see an average 6.12% utilization when the AVPS is matching but not blocking. This is considered the worst case but is still considered very small and almost has no effect on the traffic traversing or being processed. If we consider the blocking situation (the default action in an ideal AVPS deployment), we see an average of 0.11% utilization, a negligible overhead. This is expected because in this case, data packets are blocked and are not processed any further.

For the database server experiments we built a database of customers, orders, and order items and developed three different queries. Query Q1 returns the list of all items of all orders submitted by all customers for a total of 51,740 records. Query Q2 returns one record with the number of customers in a geographical region. This query needs to scan 50 customer records. Finally, query Q3 returns the dollar amount of all orders placed by customers in a given geographical region. While this query returns only a number, it needs to do significant work on the database to obtain the result.

Table II shows the measured average response time (in sec) for 10 manually submitted queries using mysqlslap for the three different queries and for the four scenarios described above. The table also shows the 95% confidence intervals for all queries.

For the case in which rules generate a violation alert but no traffic is blocked, we loaded into memory the following rule that alerts when a user tries to access "companyxyz" database located at a specific DB server.

alert tcp any any → DBserver any (classtype:attempted-user; msg:"Snortinline Autonomic DB event";content: " companyxyz";nocase;sid:2;)

We can see from Table II, that the worst case appears in Q1, which returns 51740 records. For Q1 the differences between no AVPS and AVPS matching is almost 5 msec, or 13% additional overhead. We consider the extra time to be small given the large number of records returned. In fact, the overhead is approximately 0.08 μsec per record returned. For queries Q2 and Q3 we can see almost no overhead given that both only return one record. In fact, for Q3, there is no statistically significant difference at the 95% confidence level between the no AVPS and AVPS matching cases. For Q2, the difference in response time is small and equal to 1.2 msec.

It is important to note that the largest component of the response time is the transfer time over the network and not processing time at the DB server. We measured Q1, Q2, and Q3 directly at the server and we found that Q1 takes14 msec to execute, and Q2 and Q3 take virtually zero seconds to execute. The difference in execution time between Q1 and the other two queries lies on the fact Q1 has to output a very large number of records. Thus, the average transfer time for case (4) for query Q1 is 22 msec obtained by subtracting the average response time at the client (i.e., 36 msec) from the server execution time of 14 msec.

Query →	**Q1**	**Q2**	**Q3**
Average response time (msec) with 95% confidence interval			
No AVPS, switching	31.6 ± 0.24	10 ± 0.31	10.6 ± 0.39
AVPS, process not on	32.4 ± 0.24	10.2 ± 0.2	10.8 ± 0.57
AVPS, process on but not matching	36.4 ± 0.24	11 ± 0.31	10.6 ± 0.24
AVPS, matching and policy applied	36.2 ± 0.57	11.2 ± 0.2	11.2 ± 0.37
Average/Maximum CPU utilization (%)			
AVPS - bridging only	0.024/0.15	0.045/0.23	0.007/0.04
AVPS enabled, not matching	0.43/1.51	0.01/0.05	0.058/0.3
AVPS enabled, matching, not blocking	1.57/4.75	0.152/0.43	0.23/0.71
AVPS enabled, matching, blocking	0.220/1.49	0.262/1.14	0.221/1.05

TABLE II: DB results

As before, the CPU utilization experiments use the automated submission process. In the cases where we block against a rule, we load into memory the following rule that blocks a user when he/she tries to access the "companyxyz" database located at a specific database server by replacing it with "**********".

*alert tcp any any → DBserver any (classtype:attempted-user; msg:"Snortinline Autonomic DB block"; content:" companyxyz"; nocase;eplace:"**********";sid:2;)*

Table II shows the measured average and maximum (after the "/") CPU utilization of the AVPS engine for 30 automated requests with zero think time using JMeter for queries Q1, Q2, and Q3. The minimum CPU utilization was zero in all cases.

In Table II, we notice that the average CPU utilization does not fully reflect the actual CPU utilization due to the very low amount of time that it takes to process a request over the network. The maximum CPU utilization provides a better view of the actual utilization encountered. We can see again that the worst case occurs with a maximum CPU utilization of 4.75% for Q1 when the AVPS is matching but not blocking. This overhead is considered very small and almost negligible given the number of records returned. The other queries have a maximum of 1.14% utilization, which is extremely low and can almost be completely ignored. In the case of blocking (last row), we see extremely low overhead for the worst case (Q1) that has a maximum of 1.49% utilization. Again, in an ideal environment a blocking policy would be in place.

The results of the experiments in a Web server environment

 ISBN: 978-1-61208-010-9

File Size →	**518 KB**
Average throughput (KB/sec) with 95% confidence interval	
No AVPS, switching	43038 ± 1675.01
AVPS, process not on	33861 ± 902.16
AVPS, process on but not matching	23385 ± 372.36
AVPS, matching and policy applied	17938 ± 676.78
Average transfer time (msec) with 95% confidence interval	
No AVPS, switching	6.1 ± 0.23
AVPS, process not on	7.7 ± 0.21
AVPS, process on but not matching	11.1 ± 0.18
AVPS, matching and policy applied	14.6 ± 0.47
Average CPU utilization (%) with 95% confidence interval	
AVPS - bridging only	0.03 ± 0.04
AVPS enabled, not matching	0.24 ± 0.45
AVPS enabled, matching, not blocking	0.54 ± 1.04
AVPS enabled, matching, blocking	0.15 ± 0.11

TABLE III: Web results

are shown in Table III, which presents the average throughput (in KB/sec) and the average transfer time (in msec), with 95% confidence intervals, for 10 manually submitted requests using JMeter for a Web page of 518 KB. In the cases where we check against a rule but do not block, we loaded into memory the following rule that alerts when a user tries to access the page "notallow.html" located at a specific webserver.

alert tcp any any → Webserver any (classtype:attempted-user; msg:"Snortinline Autonomic web event"; content:"notallow.html";nocase;sid:2;)

Table III indicates that the average throughput is reduced by 56% when the AVPS is running, matching, and not blocking as compared with the case of no AVPS. The response time difference in that case (see Table III) increases 2.28 times. However, the increase in time units is only 8.2 msec for a large web page (i.e., 518 KB). This increase in response time is hardly noticeable by a human being. It should be noted that in the Web case, the AVPS has to inspect every single packet of a Web page.

Table III indicates that the CPU utilization results for the web case are equally low as in the previous cases.

C. Scalability Analysis

The experiments reported in previous sections allow us to determine the execution time and overhead of running applications protected by the AVPS system. This section uses a queuing-theoretic model to explore the scalability of our proposed approach. We assume that there are M clients that submit requests that are initially processed by one of N AVPS engines, which then send the requests to an application server (AS) (e.g., FTP server, database server, Web server). Each client pauses for an exponentially distributed time interval, called *think time*, before submitting a new request after a reply to the previous request has been received. The average think time is denoted by Z. See Fig. 4 for a depiction of the model.

We also assume that the average time to process a request, not counting time waiting to use resources at the AVPS and the application server, is exponentially distributed with an average equal to $\bar{x}$.

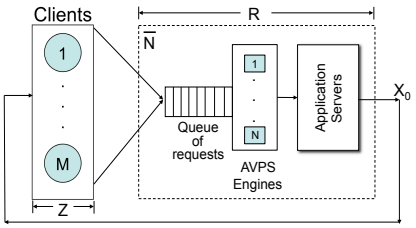

Fig. 4: AVPS analytic model.

We can use the results of the M/M/N//M queue (see [25]) to obtain the probabilities p_k of having k requests being processed or waiting by either the AVPS or the application server. The M/M/N//M queue models a variable service rate finite-population of M request generators that alternate between two states: (1) waiting for a reply to a submitted request and (2) thinking before submitting a new request after receiving a reply to the previous request.

The probabilities p_k are then given by

$$p_k = \begin{cases} p_0 \ (\bar{x}/Z)^k \frac{M!}{(M-k)!\ k!} & 0 \le k \le N \\ p_0 \ [\bar{x}/(N\ Z)]^k \frac{M!\ N^N}{(M-k)!\ N!} & N < k \le M \end{cases} \quad (1)$$

where

$$p_0 = \left[\sum_{k=0}^{N} (\frac{\bar{x}}{Z})^k \frac{M!}{(M-k)!k!} + \sum_{k=N+1}^{M} (\frac{\bar{x}}{N\ Z})^k \frac{M!\ N^N}{(M-k)!N!} \right]^{-1} \quad (2)$$

We can now compute the average number, $\bar{N}$, of requests being processed or waiting to be processed by the AVPS + application server system as

$$\bar{N} = \sum_{k=1}^{M} k\ p_k \quad (3)$$

and the average throughput X_0 as

$$X_0 = \sum_{k=1}^{N} \frac{k}{\bar{x}}\ p_k + \sum_{k=N+1}^{M} \frac{N}{\bar{x}}\ p_k \quad (4)$$

The average response time, R, can be computed using Little's Law [26] as $R = \bar{N}/X_0$.

The workload intensity of such a system is given by the pair (M, Z). An increase in the number of clients M or a decrease in the think time Z imply in an increase in the rate at which new requests are generated from the set of clients. As the processing time $\bar{x}$ increases, contention within the system increases and requests tend to spend more time in the system instead of at the client. In the extreme case, $p_M \approx 1$ and $p_k \approx 0$ for $k = 0, \cdots, M-1$. When that happens, $\bar{N} \to M$, $X_0 \to N/\bar{x}$, and $R = \bar{N}/X_0 \to M\ \bar{x}/N$. In other words, the response time grows linearly with M at very high workload intensities.

We now use the $\bar{x}$ values obtained in our measurements from Section V-B to analyze the scalability of the AVPS for

an FTP server, database server and web server under the same conditions shown in the previous sections and for a single AVPS engine (i.e., $N = 1$). Note that the values of $\bar{x}$ used here correspond to the worst-case scenario in the automated tests, i.e., case (3) in which all rules generate a violation and an alert but traffic is not blocked.

Server type		$\bar{x}$
FTP Server	100 KB	0.360 sec
	1 MB	0.513 sec
	10 MB	1.050 sec
	100 MB	8.100 sec
DB Server	Q1	41.6 msec
	Q2	14.8 msec
	Q3	15.6 msec
Web Server	518 KB	12.1 msec

TABLE IV: Average Service Time $\bar{x}$ for the FTP Server, DB Server and Web Server Applications.

Figure 5 shows the average file transfer time R when the number of clients varies from 5 to 30 for an average think time equal to 10 sec. The AVPS is enabled, matching packets against the policy, but not blocking bad transfers. If blocking were enabled the transfer time would be reduced since some files would not be transferred. As expected, for each file size, the average transfer time increases with the file size. For large files (e.g., 100 MB) and for this value of the think time, the system is close to saturation and the average transfer time increases almost linearly with the number of clients, as discussed above. For example, $R = 233$ sec for $M = 30$. This value is very close to $30 \times \bar{x} = 30 \times 8.1 = 243$ sec. For half the number of clients, R is 111.5 sec, which is almost half the value for $M = 30$. But, even in this worst case, the FTP server with the AVPS system scales linearly with the number of clients.

Before saturation is reached, the increase in average transfer time is more than linear, as can be seen for example in the 10 MB file size case. For example, the value of R for $M = 30$ is about 3.4 times higher than for $M = 15$. However, as M increases way past $M = 30$ for 10-MB files, the system will saturate and the transfer time will increase linearly with M.

Figure 6 shows the average response time, R, for the result of queries Q1, Q2, and Q3 defined in Section V-B for an average think time equal to 0.1 sec. As before, the number of clients varies from 5 to 30. The number of records returned by queries Q1-Q3 are 51740, 1, and 1, respectively. Q3 is a much more complex query and requires more database processing time. Thus, its average response time is slightly higher than that for Q2, even though both queries return the same amount of data. The graph indicates that for 30 clients and for Q1, the system is very close to saturation and the average transfer time is very close to be proportional to M. In fact, $R = 1.148$ sec $\approx 30 \times \bar{x} = 30 \times 0.0416 = 1.248$ sec. Queries Q2 and Q3 do not return enough records to push the system to saturation and therefore we see a more than linear increase in transfer time as a function of M for the values shown in the graph.

Figure 7 shows the average transfer time R for a 518-KB Web page and for an average think time equal to 1 sec. As before, the number of clients varies from 5 to 30. The graph indicates that the increase in transfer time is negligible between 5 and 30 clients. While R increases linearly with the number of clients, the rate of increase is mainly due to increased congestion at the Web server and not to AVPS overhead, which is small (8.2 msec) and hardly noticeable by a human being.

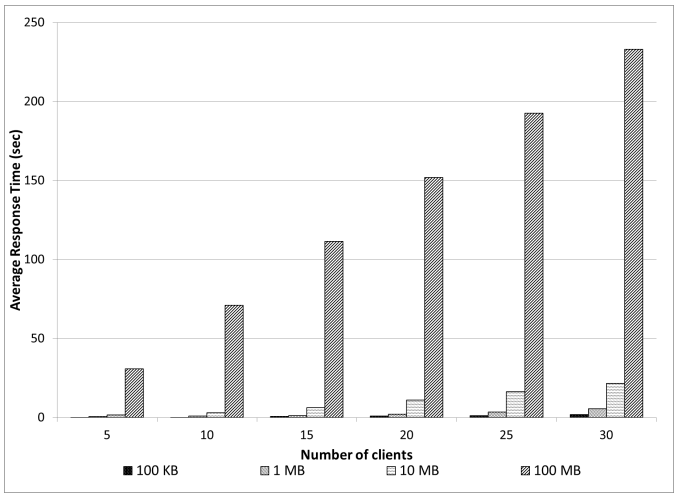

Fig. 5: Average file transfer time vs. number of clients for various file sizes. The average think time is 10 sec.

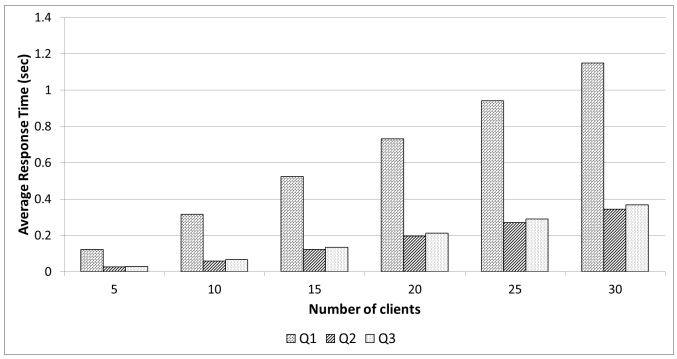

Fig. 6: Average database query result transfer time vs. number of clients for three different queries. The average think time is 0.1 sec.

The clustered architecture for AVPS allows for horizontal scaling. Using the model presented above and the FTP server example, we can see the effect of increasing the number of AVPS engines from 1 to 5. This is illustrated in Table V shows the average file transfer time for 100 MB files, 20 clients, and an average think time of 10 sec. As it can be seen, increasing the number of AVPS engines from 1 to n reduces the average transfer time by a factor larger than n.

VI. Concluding Remarks

This paper presented a scalable AVPS framework to defeat the insider threat. It also presented a performance evaluation assessment for three different application servers. In the

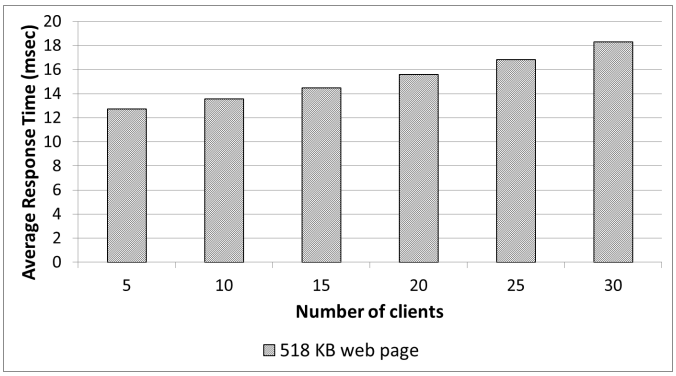

Fig. 7: Average web transfer time vs. number of clients for a 518-KB Web page. The average think time is 1 sec.

N	1	2	3	4	5
R	152	71	44	31	22

TABLE V: Average file transfer time, R, (in sec) for various values of the number of AVPS engines, N. Other parameters: $M = 20$ and $Z = 10$ sec.

performance assessment we measured average transfer times, average throughput, and CPU utilization. We also provided 95% confidence intervals for all three measurements.

The experiments showed that: (1) The impact on the average transfer time and throughput for FTP transfers is either negligible at the 95% confidence level or very small (i.e., less than 1.8%). (2) The response time impact on database queries is heavily dependent on the number of records returned by the queries. For queries that return a very large number of records (e.g., over 51,000), the response time increase is 13% on average. However, this amounts to only 0.08 μsec on average per record returned. (3) When a Web server is accessed through the AVPS system, the response time for a large Web page (e.g., 518 Kbytes) increases by 8.2 msec, an amount hardly noticeable by a human being. (4) The average and maximum CPU utilization of the AVPS engine are very small in all cases tested, not exceeding 7%.

We also presented an M/M/N//M queuing analytical scalability model and generated expected response times for all three application servers. The goal of our scalability and performance evaluation was to show that there is very low overhead incurred when the AVPS is in-line between the clients and the application servers. We used worst-case scenarios in our analysis by considering situations in which all rules checked trigger a violation and generate an alert, but do not block incoming traffic. Blocking traffic in violation situations, which is the normal operational situation, reduces the load on the network and on the AVPS engine and improves performance.

The AVPS is based on Event-Condition-Action (ECA) autonomic policies. If a condition occurs, an event/action is triggered. Rules are entered into policies manually. The use of ECA might be difficult to maintain and manage if the number of rules is very large. For this reason, we are currently looking into self-learning and self-adapting approaches to lower human involvement in rule writing. We are also looking at model based architectures, typically used in self-optimizing systems, and the effects of rule complexity on the overall performance of the system.

REFERENCES

[1] "zdnet," 2010. [Online]. Available: http://www.zdnet.com/blog/perlow/wikileaks-how-our-government-it-failed-us/14988
[2] F. Sibai and D. Menascé, "Defeating the Insider Threat via Autonomic Network Capabilities," in *Proc. Third Intl. Conf. Communication Systems and Networks, 2011.*, 2011.
[3] M. Huebscher and J. McCann, "A survey of autonomic computing - degrees, models, and applications," *ACM Comp. Surveys, Vol. 40, Issue 3*, pp. 1–28, 2008.
[4] G. Jabbour and D. Menascé, "Policy-Based Enforcement of Database Security Configuration through Autonomic Capabilities," in *Proc. Fourth Intl. Conf. Autonomic and Autonomous Systems*. IEEE Computer Society, 2008, pp. 188–197.
[5] ——, "The Insider Threat Security Architecture: A Framework for an Integrated, Inseparable, and Uninterrupted Self-Protection Mechanism," in *Proc. 2009 Intl. Conf. Computational Science and Engineering-Volume 03*. IEEE Computer Society, 2009, pp. 244–251.
[6] M. Engel and B. Freisleben, "Supporting autonomic computing functionality via dynamic operating system kernel aspects," in *Proc. 4th Intl. Conf. Aspect-oriented Software Development*. ACM, 2005, p. 62.
[7] Y. Al-Nashif, A. Kumar, S. Hariri, G. Qu, Y. Luo, and F. Szidarovsky, "Multi-Level Intrusion Detection System (ML-IDS)," in *Intl. Conf. Autonomic Computing, 2008.* IEEE, 2008, pp. 131–140.
[8] R. He, M. Lacoste, and J. Leneutre, "A Policy Management Framework for Self-Protection of Pervasive Systems," in *2010 Sixth Intl. Conf. Autonomic and Autonomous Systems*. IEEE, 2010, pp. 104–109.
[9] "viewSSLd," 2010. [Online]. Available: http://sourceforge.net/projects/viewssld/
[10] "Netintercept, Sandstorm Enterprises, Inc." 2010. [Online]. Available: http://www.sandstorm.net/products/netintercept/
[11] "Ettercap, Sourceforge," 2010. [Online]. Available: http://ettercap.sourceforge.net/index.php
[12] "Network Critical V-Line TAP, Network Critical Solutions Limited," 2008. [Online]. Available: http://www.networkcritical.com/Products/Bypass.aspx
[13] "IBM Proventia Network Intrusion Prevention System , IBM," 2011. [Online]. Available: http://www-01.ibm.com/software/tivoli/products/network-multifunction-security/
[14] "Cisco ASA, Cisco Systems," 2011. [Online]. Available: http://www.cisco.com/en/US/products/ps6120/index.html
[15] "JMeter," 2010. [Online]. Available: http://jakarta.apache.org/jmeter/
[16] "MySQL Slap, Oracle Corporation," 2010. [Online]. Available: http://dev.mysql.com/doc/refman/5.1/en/mysqlslap.html
[17] "Snort, Sourcefire, Inc ," 2010. [Online]. Available: http://www.snort.org/snort
[18] "Bro Intrusion Detection System, Lawrence Berkeley National Laboratory," 2010. [Online]. Available: http://www.bro-ids.org/
[19] "Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD), SRI Intl." 2010. [Online]. Available: http://www.csl.sri.com/projects/emerald/
[20] "Iptables, netfilter," 2010. [Online]. Available: http://www.netfilter.org/
[21] "MySQL DB, Oracle Corporation," 2010. [Online]. Available: http://www.mysql.com/
[22] "BASE Project, Basic Analysis and Security Engine," 2010. [Online]. Available: http://base.secureideas.net/
[23] "Vsftpd," 2010. [Online]. Available: http://vsftpd.beasts.org/
[24] "Apache 2, The Apache Software Foundation," 2010. [Online]. Available: http://httpd.apache.org/
[25] D. Menascé and V. Almeida, *Capacity Planning for Web Services*. Prentice Hall, 2002.
[26] L. Kleinrock, *Queueing systems, volume 1: theory*. John Wiley & Sons, 1975.

 ISBN: 978-1-61208-010-9