

# SPHINX: AN ANOMALY-BASED WEB INTRUSION DETECTION SYSTEM

Damiano Bolzoni, Emmanuele Zambon

*University of Twente,  
Distributed and Embedded System Group,  
P.O. Box 2100, 7500 AE Enschede, The Netherlands  
{damiano.bolzoni, emmanuele.zambon}@utwente.nl*

## Abstract

We present Sphinx, an anomaly-based web intrusion detection system. Sphinx analyzes the parameters sent to the web server, infers the “type” of each parameter and combines different techniques to detect attacks, specifically designed to cope with each parameter type. Sphinx, like the widely known ModSecurity, is implemented as an additional module for the Apache web server, therefore can cope with encrypted traffic.

**Keywords:** anomaly detection, intrusion detection, web application firewall

## 1 Introduction

In the last decade web-based services (also known as web applications), e.g. virtual user communities, discussion forums, e-shopping and e-banking, have played an exponentially increasing critical role in the Internet’s expansion. This success is mainly determined by two factors: firstly, web applications do not require users to set up any configuration (unlike stand-alone client softwares) to access contents, secondly new technologies (server and client-side script languages and *ad hoc* media formats) have been introduced to provide dynamically-generated and advanced multimedia contents. Furthermore, content management systems (CMSs) (e.g. PostNuke [19]) offer to inexperienced users too powerful functionalities to ease the appearance and content management of web-sites; interfaces for low-level interactions with

other services (e.g. web-mail interfaces, database and operating system administration tools) have been developed as well to improve users' efficiency.

Since most of the web applications are either open-source software, supported by a vast community of different developers with both different and poor security skills, or custom private platforms, it is even more difficult than during the usual software development to identify and fix security flaws before the final users install and use a web application. Robertson et al. [20] state that, considering the entries of the Common Vulnerabilities and Exposures (CVE) database [24] from 1999 to 2005, it is possible to estimate in more than 25% the presence of web-application-related security flaws. Furthermore, because of their accessibility and large installed base, these applications turned to be one of the most attractive attack target [9], especially when managing personal user data or financial transactions.

Detecting possible malicious activities and preventing attacks against web applications is a difficult task: network intrusion detection systems (NIDS), either signature (e.g. Snort [21, 22] has more than 1200 web-related signatures) or anomaly-based, are commonly used to detect web-based attacks. However, they present limited capacities when dealing with encrypted connections: to overcome this problem it is either possible to provide the encryption key to the NIDS, thus exposing the whole system security, or instructing the NIDS to act as a proxy, creating a dangerous single point of failure for the whole system availability. This is the main reason why specific intrusion detection systems have been studied and introduced to manage data at web application level; moreover, by taking advantage of the specificity of the web application domain, it is possible to achieve better detection results[11].

These specialized IDS are commonly known as *web application firewalls* [29] (WAFs): they work by interfacing with the web server running the web application at the top level of the network stack, thus having the possibility of analyzing encrypted data (the web server performs the decryption) as well as HTTP meta data. WAFs can be, like common NIDSs, either signature (e.g. ModSecurity<sup>TM</sup> [25]) or anomaly-based [11]: both types suffer different problematics.

Signature-based systems (SBSs), based on pattern-matching engines, use a database containing signatures of well-known attacks to detect malicious activities. New signatures must be developed as new attacks or modifications of a previously known attack are discovered: this activity, although considered necessary, is time-intensive and error-prone and requires substantial security expertise [16]). Nevertheless, attack patterns can be found also in licit data, generating false positive alerts.

On the other hand, anomaly-based systems (ABSs) usually establish an *ad hoc* model describing the behavior of the monitored system and flag any deviating activity as suspicious. Therefore, these systems are able to detect new attacks as they take place, without any previous knowledge: however a (extensive) training phase is required to build the model and, because of their nature, ABSs are well-know to generate a higher false positive rate than SBSs.

Despite these disadvantages, anomaly-detection turns to be an effective approach in discovering web attacks [20], especially when dealing with custom or high-dynamic applications (whose structure results being difficult yet impossible to be described by signatures).

**Contribution** Clearly, we cannot detect all the possible attack kinds. For instances, detecting a Denial of Service attack based on the consumption of the web server resources (e.g. generating thousands of connections per second and requesting same heavy-weight pages) is not possible analyzing the URL request parameters only. Attacks to the application logic cannot be detected if they do not involve the use of *ad hoc* forged URLs.

Following the classification provided by the *Web Application Security Consortium* [28], our approach can identify:

- command execution attacks (i.e. buffer overflow, format string, parameter injection)
- information disclosure (i.e. directory indexing, information leakage, path traversal)

**Structure of the paper** This paper is organized as follows: in Section 2 we present the system architecture and its property. In Section 3 we report the results of our benchmarks. In Section 5 we discuss related work.

## 2 Architecture

Our web-based intrusion detection system is implemented as an Apache additional module. This design choice gives several advantages compared with previous solutions. Firstly, unlike NIDSs, no encryption key is needed to analyze the incoming requests; secondly, attackers cannot tamper logs to prevent attack detection (unlike [11]); furthermore, we can protect even third-party applications which do not provide their own source code (unlike [8]). One could argue that this implementation actually protects only

one of the main competitors on the market: although no easy solution is possible (no interface is offered by other web servers), an Apache-reverse-proxy-based solution can be deployed to virtually protect any web server. Two main components forms our system: one pre-processes the training data before this is passed to the real anomaly-detection engine.

## 2.1 Data pre-processing

Clearly, because of the anomaly-detection nature of our engine, the *quality* of the data used during the training phase greatly affect the overall system completeness [4]. In fact, the use of noisy data to build the detection models could lead to false negatives (i.e. missed attacks), since the engine would recognize an anomalous input as normal.

Data cleaning can be considered one of the most serious concerns when deploying ABSs in general, since it is really difficult to collect a huge amount (even days of collected data are used) attack-free data [26]. In fact, nowadays BOTNets, and automatic scanning tools in general, are used to constantly monitor computer networks. Although not always harmful in the immediate, commonly these activities aim to identify potential victims and collect information for further operations. This is a typical example of noise inside the training data set.

Despite this issue, previous research did not focus on this important problem. The standard way to deal with this is by manually cleaning the data set. Clearly this approach totally relies on the human expertise and is labor-intensive, especially because ABSs models must be updated regularly to adapt to any environment change. The manual inspection can be supported by an automatic inspection using a SBS, that pre-processes the training data and should detect well-known attacks (e.g. web-scanners, old exploits, etc.). Wang et al. [26] apply a similar technique to improve the training phase of their ABS. Unfortunately, since signatures generate false positives (i.e. false attacks) too, they must be manually selected based on the specific network context (increasing workload).

### 2.1.1 Introducing the technique

Despite the problems previously showed, it is possible to clean the data set in such a way that the resulting model built by the engine better represents the legitimate requests; moreover, it is possible to achieve this in an automatic way, without any (or little) human involvement. Our claim is supported by some facts: (1) noise typically forms a small percentage of the total

data [18], (2) its content is typically very different from the content of regular data [5, 7] and (3) data do not need to be processed on-line (i.e. at the same moment we are collecting it), thus we do not need a powerful (and costly) hardware to accomplish the computation in a short time. The fact that we could eliminate also a percentage of legitimate data is – at this stage – not a serious concern, since data can be collected in high volumes.

With these boundaries, the problem becomes similar to the detection of outliers (i.e. anomalies) in large data sets (e.g. inside databases), where the amount of data makes infeasible any human intervention but an automatic process to discover noisy data. Therefore, intuitively we apply an unsupervised (to avoid human specialists overload) outlier detection algorithm from the data mining field to discard as many as possible attack vectors.

Any outlier detection algorithm is based on the evaluation of *features* to partition data and discover anomalies. Therefore, the selection of features is the most important step. On the other hand, this can be considered the most relevant weakness of these approaches: it is difficult to define in every context (e.g. an IDS dealing with network packets or with TCP connections) meaningful features to detect anomalies. Luckily, a web-based context provides several parameters which can be used as features to detect outliers, namely: request method, server response, bytes sent/received, number of parameters used inside the request.

Although these approaches can discover only certain kinds of anomalies, the way the information can be easily (and quickly) extracted makes it efficient (and attractive) enough to achieve our goal: previous research [6, 18] demonstrated the validity of classification techniques based on features for anomaly detection. Noisy data typically discovered in this step is constituted by:

- probes searching for common web-applications (e.g. DBMS web-based interfaces, user community support systems, etc.) or common vulnerabilities
- probes searching for mis-configured web servers or functionalities where authentication is not required (e.g. open web proxies)
- requests for web pages providing information unsafely (e.g. *info.php* or *printenv* scripts)

To perform this pre-classification task we choose the Self-Organizing Maps (SOMs) algorithm. SOMs [10] are defined as topology-preserving single-layer maps in which the topological structure, imposed on the nodes

in the network, is not changed during classification (preserving neighbourhood relations) and there is only one layer of nodes. A SOM is also suitable to analyse high-dimensional data and belongs to the category of competitive learning networks [10]. Nodes are also called *neurons*, to remind the artificial intelligence nature of the algorithm. Each neuron  $n$  has a *vector of weights*  $w_n$  associated to it: the dimension of the weights arrays is equal to the length of longest input data. These arrays (also referred as *reference vectors*) determine the SOM behaviour. It is beyond the scope of this paper to describe the SOM algorithm in depth.

From each incoming requests features are extracted and passed to the SOM, but not immediately to the main detection engine: since the algorithm firstly goes through a training phase, the classification is not produced immediately. Afterwards, when SOM training is accomplished, any request labelled as anomalous is discarded, otherwise the request is passed to the detection engine for its training phase. Once the main detection engine has been trained, following requests are not further processed by the SOM. This is because the SOM algorithm can be computationally expensive, therefore during an on-line detection phase it could affect negatively the overall system performance in terms of throughput. On the other hand, the training of the detection engine can be accomplished even off-line.

## 2.2 The detection engine

The detection engine is responsible for building the profiles and, consequently, detecting attacks. The engine works by building for each resource, that uses input parameters, the corresponding detection profiles. When a client makes a request to the web server, the engine identifies the resource (e.g. the web page) and extracts the parameters. A hash-table is used to keep track of the requested resources: each table entry owns the information regarding the parameters (i.e. the built detection profiles). When a new resource is added to the table, the engine firstly determines the type of each parameter. We define three basic types: numerical, short texts and raw data. Only one model is used with a certain parameter: during the training phase, the engine infers the type by observing some parameter characteristics (e.g. length and content). Since we could incur in the unlucky situation of analyzing first an attack vector, thus mistaking the correct parameter type, several samples of the same parameter are analyzed before the engine attaches the detection model. Then, the selected model is updated with the new samples.

### 2.2.1 Numerical model

The numerical model labels all those parameters which are carrying only numerical values (either integer or decimal). The simple idea behind this is that if we observe only numbers (and current web applications are strongly based on numerical input values, e.g. table indexes in back-end DBMSs) and then later some text, that is likely to be an attack. A considerable number of vulnerabilities affect parameters whose content is not carefully parsed before being used by the web application: evaluating the parameter type can detect immediately a SQL Injection or XSS.

### 2.2.2 Short-text model

We define short-text-based parameters those whose both content character distribution and length are either fixed or with small variations (in case of length we consider inputs less long than a given threshold). Typical examples of short-text fields are the ones used to execute functions within the application (e.g. “list”, “add” or “delete”) or to send session cookies. The model for this parameter type works combining two different techniques at the same time. During the training phase, for each sample, the average input length and its standard deviation are computed, based on the current and the previous observations. Then, the content is parsed by a grammar generator. The grammar generator derives a NFA automaton using slight modification of the Mo and Witten’s algorithm [15]. During the detection phase the current input is compared against the generated grammars: if no grammar matches, that is considered an attack. In theory, every sample observed can lead to a different grammar. If we observe thousands of (different) samples, this approach becomes infeasible. To avoid this event (and keep at the same time the size of the model small), when the numbers of generated grammars increase over a given threshold (e.g. ten), the engine tries to derive a grammar from the ones already generated (e.g. “add” and “delete” lead to different grammars, but both can be generalized with “a\*”). Although a general grammar could lead to accept more easily specially crafted attack inputs, these must anyhow look similar to normal inputs (i.e. if the general grammar does allow only alphanumeric characters, the attack vector cannot contain ‘,’ or ’\*’). In the case a general grammar could not be derived (i.e. the character distribution is too sparse, e.g. users’ passwords), the engine will use the length information as anomaly-detector input. Kruegel et al. [11] proved the validity of this approach.

### 2.3 Raw-data model

We define *raw* any binary or long-text data. Common examples of this kind of inputs are attachments in e-mail messages or posts sent to community forums. We cannot model this data with previous techniques since the content presents high variability both in length and distribution (ranging from ASCII characters to executable code). Previous solutions can hardly cope with this type of input, generating both false positives and negatives.

Differently from before, we cannot make any assumption on the data content, therefore we need a general-purpose model. In [3] we present POSEIDON, a 2-tier payload-based ABS that combines neural networks and n-gram analysis to detect anomalies. POSEIDON performs a packet-based analysis: every packet is classified by the neural network, then, using the classification information given, the real detection phase takes place based on statistical functions considering the byte frequencies and distributions (n-gram analysis). In the context of web applications, we are dealing with streams instead of packets, therefore we must adapt POSEIDON to the new analysing format.

The new engine works as follows. Since a stream does not have a maximum determined length (as for network packets), when an input stream comes we extract sequences of fixed length. The following steps remain the same: each sub-sequence is processed first by the neural network layer, then by the n-gram engine. The longer the sub-sequence length the better the detection is performed. Experimentally, we have determined the best length to be 10.

## 3 Experiments and results

To validate our architecture, we benchmark SPHINX using the DARPA 1999 data set [12]: despite criticism [14, 13] this is a standard data for benchmarking NIDSs (e.g. [27, 17]) and it has the advantage that it allows one to compare experiments. Secondly, we use a private data set.

### 3.1 Tests with the DARPA 1999 data set

The testing environment of the DARPA 1999 data set contains several internal hosts that are attacked by both external and internal attackers. Moreover, hosts inside the local area network are able to conduct attacks against external hosts.

Since we want to compare POSEIDON to Sphinx, we must apply the same testing condition used during previous tests [3]: therefore, we consider only incoming requests belonging to attack connections against hosts inside the network 172.016.0.0/16.

We train both POSEIDON and Sphinx with the data of weeks 1 and 3 (attack-free, thus we do not need to apply any pre-processing phase). Afterwards, we test using the week 4 and 5 traffic. In order to distinguish between true and false positives we can refer to the attack instance table provided by the DARPA data set authors. Table 1 reports the obtained results.

		POSEIDON	Sphinx
HTTP	DR	100%	100%
	FP	0,0016%	0%

Table 1: Comparison between POSEIDON and Sphinx on DARPA data set; DR stands for detection rate (attack instance percentage), while FP is the false positive rate (requests and corresponding percentage).

### 3.2 Tests with a private data set

We consider a private data set we collected at the University of Twente: this is *data set A*. Data were collected on a public network for 5 consecutive working days (24 hours per day) from a heavy-loaded web server. This web server hosts the department official web sites as well as student and research staff personal web pages. We did not inject any artificial attack. This data set has been used to benchmark our NIDS POSEIDON, therefore we can easily compare the achieved results by both POSEIDON and Sphinx.

To train the anomaly-detection engines of both POSEIDON and Sphinx on the data set A, we simply use a snapshot of the data collected during *working hours* (approximately 3 hours, 1,8 Gigabytes of data, randomly chosen). The chosen training data set has not been pre-processed and made attack-free: thus we can test the pre-processing capabilities of Sphinx too. For the same reason, we randomly choose another snapshot (approximately 1,8 Gigabytes of data) for testing purposes.

The alerts have been classified by the authors: we found evidences of Cross-site Scripting (XSS) and SQL Injection attacks [28]<sup>1</sup>, plus some

<sup>1</sup>This is not surprising, since Symantec Corporation [23] reports that more than 60% of *easily exploitable vulnerabilities*, whenever the exploitation code is not needed or it is

probes checking for well-known paths (33 attack evidences in total). Table 2 summarizes the results we obtained.

		POSEIDON	Sphinx
HTTP	DR	100%	100%
	FP	2,85%	0,2%

Table 2: Comparison between POSEIDON and Sphinx on our private data set; DR stands for detection rate (attack instance percentage), while FP is the false positive rate (requests and corresponding percentage).

## 4 Generating signatures

One of the most struggling task for IT personnel is managing IDSs and although brand-new detection engines arise constantly, security people is still facing the same problems of a decade ago. IT personnel is still overloaded by alert verification (for both SBSs and ABSs) and by IDS deployment. In the latter case, especially in the case of SBS deployment, the personnel could be involved in the generation of specific signature to cope with the deployment site. This is particularly true for web applications, where the input parameter number can be extremely high.

Commonly, automatic signature generation systems, once an attack is detected, try to extrapolate a significant attack payload to generate signatures for a SBS. Signatures can more easily be used to describe what is malicious rather than what is normal. The previous situation is common in NIDSs, where the monitored environment is heterogeneous (different services providing different application-level functionalities) and it is difficult to characterize normal data. In the context of a web application the characterization task is much easier, especially because being based on parameters our analysis can be more precise and granular. Thus, unlike other automatic signature generation systems, we do not generate (although this could be possible) attack signatures but rather normal data signatures, which can be used to automatically configure WAFs, e.g. ModSecurity. Typical examples of generated signature are the following:

- SecRule QUERY\_STRING “!^(id=\d+)” → *accepts only digits for id*

---

well-know (like SQL Injection and XSS attacks), affect web-based services

- SecRule QUERY\_STRING “!\^(func=(add|delete|view))” → *accepts only certain values for func*

Signature generation is possible only for numerical and (some) short-text parameters, while it is infeasible for raw data. A possible scenario we can imagine is to have a hybrid IDS with ModSecurity (using the generated signatures, processing numerical and short-text parameters) and Sphinx (processing raw data) joint together.

## 5 Related work

**Web application firewalls** Despite the fact that web applications have been widely developed only in the last half-decade years, the detection of web-based attacks has immediately received considerable attention.

Kruegel et al. [11] present a system able to analyze HTTP that takes significant advantage of the parameter-oriented URL format common in web applications. The system applies up to nine different models at the same time to detect possible attacks, namely: attribute length, attribute character distribution, structural inference, token finder, attribute presence, attribute order, access frequency, inter-request time delay and invocation order. It is easy to note that the detection models perform essentially either a content or time-series analysis on the requests. Three different private data sets are employed to execute experiments: authors observe that applying only attribute length and character distribution all the attack types are detected. The authors further expand their previous work in [20]: only five of the previous nine detection models, discarding the ones that did not detect any attack type, are employed to detect attacks and, lately, to generate anomaly-signatures once an anomaly has been found.

Almgren et al. [1] and Almgren and Lindqvist [2] present similar systems which are based on signature-based techniques and analyze in the former case web server logs and in the latter are directly integrated as part of the web server itself. Hence, these systems hardly rely on signature generation while our system aims to detect attacks in a totally automatic and unsupervised manner.

ModSecurity [25] is the widest deployed WAF: its detection engine is mostly based on pattern-matching (some very simple anomaly-based checks have been recently added). It is implemented as a stand-alone module interfaced directly to the web server: thus, it can examine (previously) encrypted data too. It offers a considerable range of possibilities to examine data and check for different attacks and the ability of using Snort [22] signatures as

well. The Achille's heel of ModSecurity lies in its configuration: unfortunately, since most web applications are tailored to specific requirements, ModSecurity can offer only a limited out-of-the-box configuration (reliable for some standard attacks only) and it must be adapted to any update/add of the web applications.

**Improving the training data set** Most researchers focus on improving the detection model once it has been built rather than the training data set. Wang et al. [26] employ Snort [22] signatures for a semi-supervised learning to improve the accuracy of their NIDS Anagram. Proper signature selection is a labour-intensive task, since it is possible that a good deal of signatures results in conflict with the monitored system(s). Furthermore, the authors recognize the difficult task of avoiding the processing of partial licit traffic enclosed in the selected signatures (i.e. valid paths with harmful parameters). We believe that this approach is unfeasible in common organization environments (well-trained IT personnel required and time wasted to fully accomplish the task).

The authors draft also an a-posteriori technique that involves the presence of so-called *shadow servers* (equipped with host-based protections and tools to detect zero-day attacks). Again, this technique is not feasible/valuable in small/medium organizations or when monitoring non-high-critical systems, since a twin-machine is required for each monitored one.

## References

- [1] M. Almgren, H. Debar, and M. Dacier. A lightweight tool for detecting web server attacks. In *NDSS '00: Proc. of 11th ISOC Symposium on Network and Distributed Systems Security*, 2000.
- [2] M. Almgren and U. Lindqvist. Application-integrated data collection for security monitoring. In *RAID '01: Proc. 4th Symposium on Recent Advances in Intrusion Detection*, volume 2212 of *LNCS*, pages 22–36. Springer-Verlag, 2001.
- [3] D. Bolzoni, E. Zambon, S. Etalle, and P. Hartel. POSEIDON: a 2-tier Anomaly-based Network Intrusion Detection System. In *IWIA '06: Proc. 4th IEEE International Workshop on Information Assurance*, pages 144–156. IEEE Computer Society Press, 2006.
- [4] H. Debar, M. Dacier, and A. Wespi. A Revised Taxonomy of Intrusion-Detection Systems. *Annales des Télécommunications*, 55(7–8):361–378, 2000.
- [5] D. E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

- [6] M. O. Depren, M. Topallar, E. Anarim, and K. Ciliz. Network Based Anomaly Intrusion Detection using Self Organizing Maps (SOMs). In *SIU '04: Proc. 12th IEEE National Conference on Signal Processing and Applications*, pages 76–79, 2004.
- [7] H. S. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Technical Report A010, SRI, 1994.
- [8] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *S&P '06: Proc. 26th IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society, 2006.
- [9] D. V. Klein. Defending Against the Wily Surfer-Web-based Attacks and Defenses. In *Proc. of the Workshop on Intrusion Detection and Network Monitoring*, pages 81–92. USENIX Association, 1999.
- [10] T. Kohonen. *Self-Organizing Maps*, volume 30 of *Springer Series in Information Sciences*. Springer, 1995. (Second Extended Edition 1997).
- [11] C. Kruegel, G. Vigna, and W. Robertson. A multi-model approach to the detection of web-based attacks. *Computer Networks*, 48(5):717–738, 2005.
- [12] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 34(4):579–595, 2000.
- [13] M. V. Mahoney and P. K. Chan. An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection. In G. Vigna, C. Kruegel, and E. Jonsson, editors, *RAID '03: Proc. 6th Symposium on Recent Advances in Intrusion Detection*, volume 2820 of *LNCS*, pages 220–237. Springer-Verlag, 2003.
- [14] J. McHugh. Testing Intrusion Detection Systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):262–294, 2000.
- [15] D. H. Mo and D. H. Witten. Learning text editing tasks from examples: a procedural approach. *Behaviour and Information Technology*, 11(1):32 – 45, 1992.
- [16] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.
- [17] T. Pietraszek. Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In E. Jonsson, A. Valdes, and M. Almgren, editors, *RAID '04: Proc. 7th Symposium on Recent Advances in Intrusion Detection*, volume 3224 of *LNCS*, pages 102–124. Springer-Verlag, 2004.
- [18] L. Portnoy, E. Eskin, and S. J. Stolfo. Intrusion detection with unlabeled data using clustering. In *DMSA '01: Proc. of ACM CCS Workshop on Data Mining for Security Applications, 8th ACM Conference on Computer Security (CCS' 01)*. ACM Press, 2002.
- [19] PostNuke. PostNuke Content Management System, 2006. URL <http://www.postnuke.com/>.

- [20] W. Robertson, G. Vigna, C. Krueger, and R. A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *NDSS '06: Proc. of 17th ISOC Symposium on Network and Distributed Systems Security*, 2006.
- [21] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *LISA '99: Proc. 13th USENIX Conference on System Administration*, pages 229–238. USENIX Association, 1999.
- [22] Sourcefire. Snort Network Intrusion Detection System web site, 1999. URL <http://www.snort.org>.
- [23] Symantec Corporation. Internet Security Threat Report, 2006. URL <http://www.symantec.com/enterprise/threat-report/index.jsp>.
- [24] The MITRE Corporation. Common Vulnerabilities and Exposures database, 2004. URL <http://cve.mitre.org>.
- [25] Thinking Stone. ModSecurity™, 2002–2006. URL <http://www.modsecurity.org>.
- [26] K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In *RAID '06: Proc. 9th International Symposium on Recent Advances in Intrusion Detection*, volume 4219 of *LNCS*, pages 226–248. Springer-Verlag, 2006.
- [27] K. Wang and S. J. Stolfo. Anomalous Payload-Based Network Intrusion Detection. In E. Jonsson, A. Valdes, and M. Almgren, editors, *RAID '04: Proc. 7th Symposium on Recent Advances in Intrusion Detection*, volume 3224 of *LNCS*, pages 203–222. Springer-Verlag, 2004.
- [28] Web Application Security Consortium. Web Security Threat Classification, 2005. URL <http://www.webappsec.org/projects/threat/>.
- [29] Web Application Security Consortium. Web Application Firewall Evaluation Criteria, 2006. URL <http://www.webappsec.org/projects/wafec/>.