# Learning DFA Representations of HTTP for Protecting Web Applications


Kenneth L. Ingham [a] Anil Somayaji [b] John Burge [a]
Stephanie Forrest [a,c]

[a] *Department of Computer Science,*
*University of New Mexico,*
*Mail stop: MSC01 1130,*
*Albuquerque, NM 87131-0001, USA*

[b] *School of Computer Science,*
*Carleton University,*
*5302 Herzberg Building,*
*1125 Colonel By Drive,*
*Ottawa, Ontario, K1S 5B6, Canada*

[c] *Santa Fe Institute*
*1399 Hyde Park Road*
*Santa Fe, New Mexico 87501, USA*


---


**Abstract**

Intrusion detection is a key technology for self-healing systems designed to prevent or manage damage caused by security threats. Protecting web server-based applications using intrusion detection is challenging, especially when autonomy is required (i.e., without signature updates or extensive administrative overhead). Web applications are difficult to protect because they are large, complex, highly customized, and often created by programmers with little security background. Anomaly-based intrusion detection has been proposed as a strategy to meet these requirements.

This paper describes how DFA induction can be used to detect malicious web requests. The method is used in combination with rules for reducing variability among requests and heuristics for filtering and grouping anomalies. With this setup a wide variety of attacks is detectable with few false positives, even when the system is trained on data containing benign attacks (e.g., attacks that fail against properly patched servers).



*Key words:* anomaly intrusion detection, finite automata induction, web server security


---


*Email addresses:* `ingham@cs.unm.edu` (Kenneth L. Ingham), `soma@scs.carleton.ca` (Anil Somayaji), `lawnguy@cs.unm.edu` (John Burge), `forrest@cs.unm.edu` (Stephanie Forrest).

## 1 Introduction

Self-healing systems detect problems and repair them with little or no human intervention, and do so in a timely fashion. Although such detection/response combinations are desirable for many classes of problems, one of the most compelling applications is that of computer security, where Internet-connected computers are routinely attacked by computer viruses, worms, and human adversaries. In living systems, response to damage from analogous threats is coordinated primarily by the immune system. Similarly, "computer immune systems" have been proposed to respond to threats that circumvent conventional computer defenses. Earlier work in computer immunology has addressed many security threats, by program code [1,2] monitoring network connections [3,4,5] and low-level program behavior [6,7,8,9]. However, as we explain below, these methods in their current form are not appropriate for protecting web server applications.

Web servers are important to protect because they so ubiquitous, yet they are currently poorly defended. For example, web-based vulnerabilities have been estimated to account for more than 25% of the total number of reported vulnerabilities from 1999 to 2005 [10]. Web servers must accept complex, highly variable inputs from virtually any host on the Internet, and with the emergence of web application services, they must process those inputs in arbitrarily complex ways. These characteristics make the problem of securing web servers especially challenging.

One approach to protecting computer systems, including web servers, is to craft specific defenses for each observed problem, either in the form of code patches or attack signatures. Both strategies, however, require a human to analyze each problem and develop the solution. This limits the feasible response time to a timescale of hours or days, but attacks by self-replicating programs (viruses or worms) can manifest in a matter of seconds [11]; thus, there is a need for automated mechanisms that can detect and respond to threats in real time. Anomaly-detection methods have been proposed as an alternative because they can potentially detect novel attacks without human intervention [12,13,14,6]. In anomaly detection, a model of normal behavior is developed from empirical observations, and subsequent observations that deviate from the model are labeled anomalies. Anomaly detection is problematic in the case of web servers, however, because web traffic is highly variable and it is difficult to characterize normal behavior in a way that both detects attacks and limits the rate of false alarms.

Other researchers have addressed the anomaly detection problem for web servers by measuring the frequency distribution of one or more simple features of a web request (the conduit of most attacks on web servers) and combining those features into a single anomaly measure. Although the features taken individually, such as character frequency, are often not be sufficient for accurate detection, multiple features can be combined with better results [15]. however, these anomaly detectors

are trained on data that are free of attacks. This requirement is unfortunate because the normal background of today's Internet contains large numbers of attacks, most of which are harmless against properly patched servers. Signature-scanning intrusion-detection systems (IDS) such as *snort* [16] can be used to filter out known harmless attacks; however, the high accuracy required for training can requires frequent updates to the attack signature database and careful site-specific tuning to remove rules that generate false alarms. This manual intervention reduces the main advantage of using anomaly detection.

To summarize, there is a need for web servers that can detect and repair damage caused by security violations with minimal human interaction. To accomplish this will require techniques for detecting anomalous web requests that are tolerant of the benign attacks that are inevitably present in observations of normal web traffic. This paper describes a system that addresses these criteria, which learns normal requests using deterministic finite automata (DFA) induction. To account for the high variability in normal web requests, we combine the DFA with a set of simple parser heuristics, which remove the most variable parts of the web request before the induction step, and anomaly heuristics for classifying detected anomalies during the testing phase.

In order to test our system, we collected a data set consisting of 65 attacks against web servers (a much larger corpus than has been used in earlier work, and we studied the normal traffic from four distinct web sites over the course of months. The tests show that our system can detect 79% of the attacks with 40 false alarms per day on a complex website; with simpler websites, our system can detect 90%+ with just a few alarms per day. These numbers are competitive with similar methods but do not require pre-filtering of training data to eliminate attacks in the data.

In the remaining sections, we first give background information on intrusion detection Section 2. We then describe the DFA method for modeling HTTP requests (Section 3), the datasets we developed for testing (Section 4), and our experimental results Section 5. Implications of the work are discussed in Section 6. Section 7 reviews related work, and Section 8 gives a summary of our results and plans for future work.

## 2 Background

A brief overview of intrusion detection in the context of web server security is given, with an emphasis on anomaly-based intrusion detection. We then compare our system to the most similar approaches, leaving the discussion of other related work to Section 7.

### *2.1 Intrusion Detection*

As discussed earlier many intrusion detection techniques are relevant to web server protection, and some are used in production environments. These systems generally scan network packets, searching for signatures of known attacks [16]. These signature-based methods require significant tuning to reduce false positives, they cannot detect zero-day attacks, the rule sets must be continually updated to track evolving threats, and they may be vulnerable to "squealing" when an attacker carefully crafts normal packets to match attack signatures [17].

Specification-based systems detail what behavior is allowed instead of what is not allowed, as signature detectors do. A common example is that of network firewalls, which restrict classes of network traffic allowed into or out of a network. Web servers, however, must accept connections from almost any machine on the Internet, which means that traffic content must be examined in addition to traffic type. Some application-specific firewalls [18,19,20,21] pre-process and filter incoming traffic to repair or remove malformed requests. Sometimes, the allowed behavior at a lower-level, for example, by restricting the types and arguments of system calls into the operating system kernel [22]. Because web servers are so complex, and the behavior of web-based applications is variable, it is difficult to create an appropriate set of rules. In fact, it might be easier to simply audit the targeted programs, removing the sources of vulnerabilities. And, new functionality may violate existing rules, so the specifications must be re-evaluated each time a web server receives a significant upgrade.

For an IDS to fit within the self-healing framework, we need a system that works autonomously, specifically, one that does not require manual monitoring or updating, one that easily adapts to modified applications, and one that responds to novel (zero day) attacks. To be practical, the system should also have modest CPU and storage requirements. These are challenging requirements, and no existing system (research or commercial) meets all of them. The systems that come closest, however, are based on the principles of anomaly detection.

### *2.2 Anomaly-Based IDSs*

Anomaly-based IDSs assume that intrusion attempts are rare and that they have different characteristics from normal behavior. Because they typically operate in an environment with frequent configuration changes, hardware faults, software errors, and other transient problems, all of which can generate unusual behavior, anomaly detection systems almost always generate some false positives. False positives are thus an important metric for evaluating IDS performance. Most systems are evaluated by testing against a corpus of known attacks, and system parameters are ad-

justed to maximize the number of detected attacks while minimizing false positives. Thus, it is essential that the evaluation use a realistic sample of normal behavior; otherwise, it is easy to choose parameter settings that are too sensitive and will produce high false positives in practice.

False positives are related to the variability of observed behavior and the techniques used to model that behavior. If the problem domain is highly variable under normal conditions (e.g., web requests), then generalization can compensate. If the observed behavior has low variation, then less generalization is required. Because generalization can cause dangerous behavior to be classified as normal, we in general want to minimize generalization while still achieving a reasonable false-positives rate. This tradeoff exists in all anomaly-detection applications, but the details of how it is managed depend on the details of the application itself.

Anomaly detection was introduced early to the field of intrusion detection [12,13]. NIDES, in particular, relied on sophisticated statistical modeling methods [13]. Much of this work focused on user behavior to address insider threats, a domain with high variation. In practice, these these early systems had high false-positive rates relative to other approaches, for example those based on monitoring program behavior [23].

Intrusion detection had more success when applied at the level of network connections and to low-level program behaviors. At the network level, detection was based on unusual traffic based upon its source and destination IP addresses, protocol (TCP or UDP), and ports [14,24,25]. However, because nothing is unusual or particularly dangerous about a new machine connecting to a web server, another approach is needed for the web server problem. Forrest et al. [6] introduced the idea of modeling program behavior by recording short sequences of system calls. This work generated significant interest and many proposals for alternative models of system call behavior, including variable length sequences [26] and rule-based data mining [27]. Warrender et al. [28] evaluated many of these alternatives, concluding that there was little difference in performance among the various methods, and notably, that the simplest methods performed about as well as the more complex approaches. This result illustrates the simplicity of the learning problem when the problem is formulated so that the observed data are relatively stable.

Unfortunately, system call monitoring does not appear to be a viable approach to detecting attacks against web servers. Earlier work with pH [9], a Linux-based IDS that monitors sequences of system calls, produced results suggesting that large server programs such as web servers are difficult to monitor in practice, sometimes taking months to achieve a stable model of normal behavior (although see [29] for how to address this). A more serious problem is that many attacks against web servers are due to configuration mistakes or errors in the web application code. Exploits based on these vulnerabilities may not generate unusual system call patterns because they may not cause any unusual executions of web server code (because

the web server is simply interpreting the vulnerable scripts). This kind of attack is likely to be undetected by many system call-based approaches. Because all I/O behavior typically is passes through the system call interface or something equivalent, virtually all attacks on web servers should, in principle, be detectable by observing arguments to system calls. Earlier attempts to model system call arguments, however, generated high false positives [30], despite the use of a much more sophisticated modeling strategy than Forrest et al. originally proposed. This discrepancy appears to be a natural consequence of the high variability of system call arguments relative to sequences of system calls.

### *2.3 Anomaly detection for web servers*

Earlier work using anomaly detection to protect web servers monitored network data rather than patterns of network connections or low-level program behavior. Some systems treat web servers as a generic network service. Incoming and outgoing traffic are modeled as a stream of bytes or as discrete packets. Some approaches search for anomalous packet headers [31], while others look for patterns in the first few packets of connections [32]. Packet payloads are natural places to look for malicious content. Wang and Stolfo [33] modeled packet payloads by comparing character distributions between the payloads of similar-sized packets, and Vargiya and Chan [34] developed multiple statistical techniques for dividing packet data into tokens used for anomaly detection. All of these systems were evaluated using the 1999 MIT Lincoln Labs data [35], a data set that has only four attacks against web servers.

In contrast to these protocol-independent approaches, Kruegel and Vigna [15] and Kruegel et al. [36] studied anomalous web requests that were destined for common gateway interface (CGI) programs. They combined multiple characteristics of request parameters including order, presence, and variability. Their system was tested using extensive normal data sets from multiple sites (including Google), but only twelve attacks.

Tombini et al. [37] combined misuse and anomaly detection to find attacks in logged HTTP requests. Their approach takes advantage of the strengths of both misuse (signature) and anomaly detection to improve performance, and they tested a portion of it with 56 web request attacks. Their approach is limited from a self-healing standpoint, however, because it requires significant site-specific manual configuration and tuning. Robertson et al. [10] recently proposed that false positives could be reduced in the anomalous web request domain by grouping anomalies and using heuristics to identify the attack type associated with request. This proposal is similar in spirit to the heuristics we present in Section 3.4.

One key problem not addressed by earlier systems is the presence of benign attacks in normal training data. Most systems pre-filtered common attacks from their training data before building their models. To understand why this is problematic, the example of a cross-site scripting (XSS) attack. In this class of attack, an attacker exploits a web site that allows comments to be added to a page—for example, most blogs have a way for readers to comment on the blog. In the case of a XSS attack, attacker injects code as part of the comment, which is executed on the browser of the machine that displays the comments. In some circumstances, scripts gain access to cookies, form data, and other browser information that might be sensitive. The scripts can also initiate communication to hostile web servers.

Suppose the protected web server accepted normal requests of the form:

```
GET /scripts/access.pl?user=johndoe&cred=admin
```

If the `cred=` portion had been vulnerable to a cross-site scripting (XSS) attack in the past, then the training data for the anomaly detector might include instances such as (the two examples are equivalent, but encoded differently) [1] :

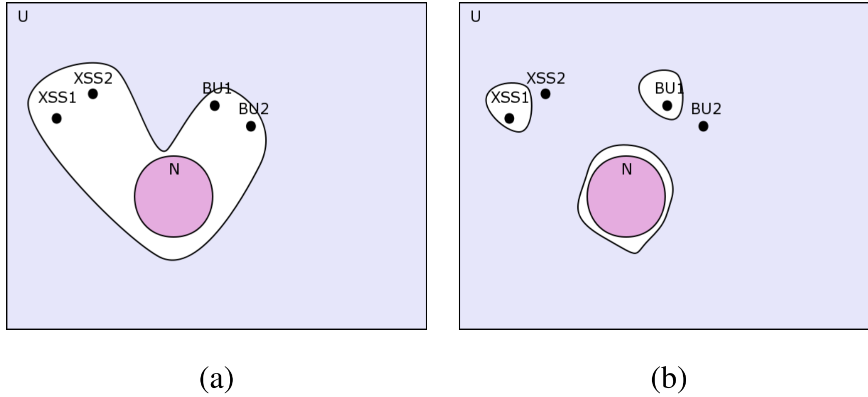
```
GET /scripts/access.pl?user=johndoe&cred=<script>doc
ument.location='http://www.cgisecurity.com/cgi-bin/c
ookie.cgi?' +document.cookie</script>

GET /scripts/access.pl?user=johndoe&cred=%22%3e%3c%73
%63%72%69%70%74%3e%64%6f%63%75%6d%65%6e%74%2e%6c%6f
%63%61%74%69%6f%6e%3d%27%68%74%74%70%3a%2f%2f%77%77
%77%2e%63%67%69%73%65%63%75%72%69%74%79%2e%63%6f%6d
%2f%63%67%69%2d%62%69%6e%2f%63%6f%6f%6b%69%65%2e%63
%67%69%3f%27%20%2b%64%6f%63%75%6d%65%6e%74%2e%63%6f
%6f%6b%69%65%3c%2f%73%63%72%69%70%74%3e
```

If attacks such as these appear in the training data, they will eventually affect statistical representations of normal:

- The attribute character distribution would be biased toward cross-site scripting-type values.
- The attribute and overall request length would be shifted toward larger values (XSS attacks are longer by the length of the attacking script plus required HTML or related code).

[1] These attack strings use examples from http://www.cgisecurity.com/articles/xss-faq.shtml.




(a) (b)

Fig. 1. Generalization strategies: U is the universe of all possible HTTP requests, N illustrates the set of normal requests, XSS1, XSS2 indicate cross-site scripting attacks, and BU1 and BU2 indicate buffer overflow attacks. If XSS1 and BU1 are in the training data, the lines surround what an anomaly detection system would be expected to accept. (a) Existing systems, for example, a simple statistical system such as character distribution. (b) Desired systems.

If a new XSS vulnerability were discovered, this time associated with a different program, exploits for it would likely have similar character distributions, attribute lengths, and request lengths to that of the earlier attacks. And, if these measures were part of a model of normal web requests, the system would be trained to tolerate not only the original attacks but the newer ones as well. The same is true of other attacks as well, for example buffer-overflows. If harmless attacks are in the training data, then variants and combinations of these classes of attacks are likely to be accepted as normal by anomaly detection systems that combine multiple, independent measures of requests. This process is illustrated in Figure 1(a).

To minimize the impact of benign attacks in normal data, we need to minimize generalization, as illustrated in Figure 1(b). The DFA method described in the next section is designed to perform much less generalization than other proposed methods for modeling web requests. Naively applied, our method in fact performs too little generalization: When running on unprocessed data (such as raw characters or whitespace-separated tokens), it will not generalize over simple variations in benign requests (such as time stamps or hashes). By being careful in our choice of tokens, however, we found that this problem can be largely avoided.

## 3 Modeling web requests

In this section we first describe our method for modeling web requests, first by discussing how we tokenize web requests, and then by explaining how we build a DFA model of requests. Next, we we describe how new behavior is compared to our DFA model. Then, we discuss how to mitigate false positives.

### 3.1 *Tokenizing HTTP requests*

Our tokenizer generates tokens that are then fed to the DFA induction algorithm. The tokenizer first extracts the tokens as described in the HTTP RFC standard [38]. These tokens combine the token type (e.g., `pathname component`) and optionally the value (e.g., `cgi-bin`). The parsing process is complicated by the fact that some web browsers and many web robots fail to follow the standard. This means that if software (including our own) is to understand the HTTP request it must be more general than the standard. Although achieving this functionality was laborious, the process was rather straightforward.

A second challenge was variation in the stream of tokens that were produced by the tokenizer. We found that most of the values are needed to properly distinguish attacks from normal requests; thus, we have had to tolerate higher variability in web requests than is desirable. We dealt with this by identifying which tokens are the most variable values and devising parser heuristics for reducing their variability. Specifically, we use the following parser heuristics:

- All upper case letters in the tokens (names and values) are mapped to lower case letters.
- File names are mapped to their file type. In the case of an unknown type, the filename is instead mapped to its length.
- Hostnames and IP addresses are replaced by a values indicating whether or not they are syntactically correct.
- Dates are parsed and replaced by a value indicating whether it follows a known date format. The parser recognizes the three formats specified in the HTTP standard along with several others that appeared in the test data.
- Hash values from PHP session identifiers and Entity Tags are validated and removed.
- Q-values are floating point values in the range [0,1] that are used in feature negotiations such as preferred languages and file types. Q-values are replaced with a valid/invalid flag.
- User names in email addresses are replaced with their length.

By replacing these highly variable attribute pairs with values that are both more predictable and indicative of whether those particular pairs are valid or not, we simplified the DFA induction problem significantly. However, we still need an induction method that generates relatively compact and effective models of web requests. We adopted a induction algorithm developed by one of us (JB).

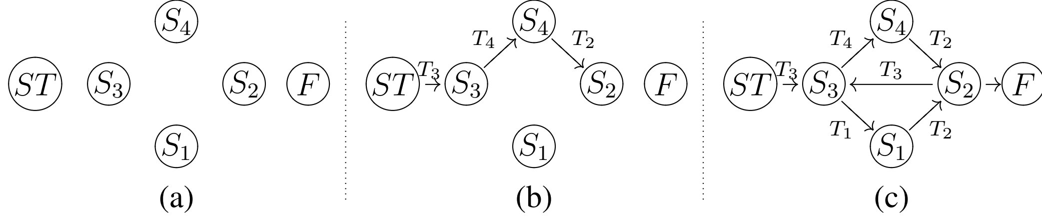



Fig. 2. The DFA induced by the Burge algorithm for the series of tokens $T_3, T_4, T_2, T_3, T_1, T_2$. a) The initial empty DFA with one state for each token. The *ST* and *F* nodes correspond to the $S_{START}$ and $S_{FINISH}$ states. b) The state of the DFA after the $T_3, T_4, T_2$ tokens have been read. c) The final state of the DFA after the entire series of tokens has been read in. Note that all states corresponding to consecutive tokens in the HTTP request are connected with a transition.

### *3.2 Burge DFA Induction Algorithm*

The Burge DFA induction algorithm is an $O(nm)$ algorithm, where $n$ is the number of samples in the training data set and $m$ is the average number of tokens per sample. The algorithm does not require negative examples, and as we will describe, the resulting DFA can be modified easily to deal with a nonstationary environment, i.e. one that constantly changes.

To model a web request, we construct an initial DFA as follows. First, let $\Sigma = \{T_1, \ldots, T_n\}$ be the set of $n$ unique tokens in the HTTP request, and let $L = (l_1, \ldots, l_t)$ with $l_i \in \Sigma$ be the series of $t$ chronologically ordered tokens from the HTTP request, with `FINISH` added in as a special token to indicate the end of the request. Let $G = (S, A)$ be a DFA with states $S$ and transitions $A$. $S = \{S_{START}, S_1, \ldots, S_n, S_{FINISH}\}$ where states $S_1, \ldots, S_n$ have a one-to-one correspondence with tokens $T_1, \ldots, T_n$, and $S_{START}$ and $S_{FINISH}$ are additional states. $E(t)$ is a function that returns the state to which $t$ will cause a transition. $A = \{A_{i,j}\}$ where $A_{i,j}$ indicates a transition labeled $T_j$ between states $S_i$ and $S_j$.

Given these definitions, the algorithm proceeds as follows:

(1) Set the current state $C = S_{START}$, $A = \emptyset$.
(2) For $j = 1$ to $t$ :
  (a) If $A_{C,E(l_j)} \notin A$ then $A \leftarrow A \cup A_{C,E(l_j)}$
  (b) $C \leftarrow E(l_j)$
(3) $A \leftarrow A \cup A_{C,S_{FINISH}}$

A DFA $G$ is constructed with one state for each unique token in the HTTP request, as well as the two additional states $S_{START}$ and $S_{FINISH}$. At the start of the algorithm, no transitions are present in the DFA and a current state $C$ is set to the $S_{START}$ state. Then, in step 2, tokens are sequentially processed from $L$. In step 2a, if no existing transition exists between the current state and the state corresponding to the current token, the transition is created and the new transition is labeled with the current

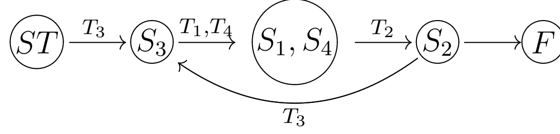



Fig. 3. Compressed version of the DFA in Figure 2c. States that have identical sources and destinations ($S_1$ and $S_4$) are compressed into the same state ("$S_1, S_4$"). Tokens that originally caused a transition into one of the uncompressed states cause transitions into the compressed state.

token. $C$ is then updated to be the state corresponding to the current token. After all the tokens have been processed, a transition from $C$ to $S_{FINISH}$ is created. Figure 2 shows an example. At this point, the DFA model has one state corresponding to each unique token present in the HTTP request, and the DFA has a transition between any two states that were seen consecutively in the request. The transition is labeled with the token corresponding to the destination state.

Additional requests are processed by running the algorithm with the tokens from the new request, adding nodes as needed for new tokens and adding edges to the DFA as described above. As the learning proceeds, the compacting processes described below is regularly performed.

### *3.2.1 Compacting and Generalizing the DFA model*

In practice, the DFA induction algorithm described above leads to large, complex DFAs that potentially could grow without bound in dynamic environments with perpetually novel HTTP requests. We use two techniques to manage this complexity, one that compacts an existing model and one that adds states and transitions incrementally and "forgets" structures that have not been used.

To reduce DFA size, our algorithm searches at regular intervals for nodes that have the same source and destination (sorting the nodes by source and destination nodes can allow this comparison to run in $O(n \log n)$ time). These nodes are combined into a single node, as illustrated in Figure 3.

This compression is a useful form of generalization. For example, suppose that during learning the DFA was compressed, producing Figure 3. Then, as learning continued $T_1$ token was observed leading to the compressed state $(S_1, S_4)$. Given the DFA's current topology, the next expected token would be $T_2$. However, if a new token, $T_5$, was observed before the expected $T_2$ token, the DFA learning algorithm would insert a new state, $S_5$, into the DFA (Figure 4). This new topology is a generalization of the observed token sequence because either a $T_1$ or $T_4$ token followed by a $T_5$ token will lead to the new $S_5$ state (Figure 5). Thus, modifying the topology of a compressed DFA allows the resulting DFA to generalize from any of the constituents of a compressed node to other similar but unobserved behavior involving the node's other constituents. This form of generalization is well matched

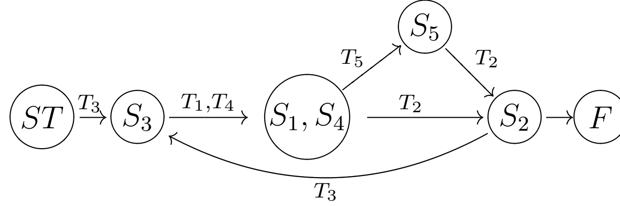



Fig. 4. Compressed DFA from Figure 3 after additional learning.

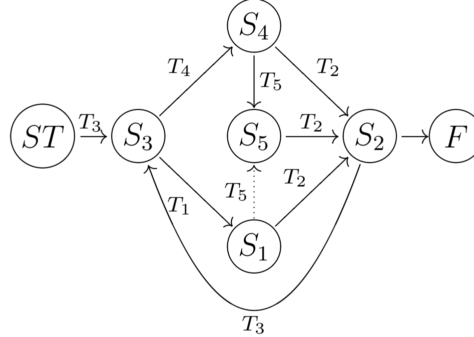



Fig. 5. The DFA in Figure 4 without compression. The dotted link between states $S_1$ and $S_5$ indicates what is generalized when learning on the compressed DFA.

with the generalization we need for web requests.

Without compression before the $T_5$ token was observed, this generalization would not occur because once the $T_5$ token is observed, the $S_1$ and $S_4$ states can no longer be merged. However, the method may miss some generalizations by not compressing the DFA before generalizable behavior is observed. This limitation could be addressed by counting the number of times certain links have been observed and compressing all states that differ by a small number of incoming and outgoing link counts.

### *3.3 Determining similarity between a request and the DFA*

A DFA by itself is simply a language acceptor; however, we expect some variation in normal behavior (changes in clients, site content, or usage patterns) that is not covered by the generalization algorithms. In this section we define a distance measure for comparing a DFA model with a web request to determine how anomalous it is. With this measure, we can tune the relative rate of true positives to false positives.

During testing, the DFA model is used differently from a traditional DFA. When a token is processed that is illegal according to the DFA, a "missed token" event is recorded and an attempt is made to resynchronize. For example, in Figure 3, suppose the current state were $S_1, S_4$ and the next token were $T_3$. If the edge corresponding to the next token exists elsewhere in the DFA, then the tester could

transition to the edge's destination. Continuing the example, the new current state would be $S_3$, and a missed-token event would be recorded. If no such edge exists, a second missed-token event is recorded, and the tester moves to the next token, again attempting to resynchronize. The number of missed tokens provides an estimate of how anomalous the request is with respect to the DFA. The similarity $s$ between an HTTP request and the DFA is calculated by:

$$\frac{\text{\# of tokens reached by valid transitions}}{\text{\# of tokens in the HTTP request}} \in [0, 1]$$

The similarity measure reflects the changes that would have to be made to the DFA for it to accept the request (i.e., for each missed token a new transition would have to be added). Note that the significance of a single missed token by this measure depends upon the total number of tokens in the request. One benefit of this sensitivity is that more complex requests (i.e., those specifying many of the HTTP header options) have more room for slight changes and can still be accepted as normal. A measure that is less sensitive to the token count is likely to have a higher false positive rate due to the common, benign variations in complex requests.

Unlike Hidden-Markov Model (HMM) approaches to learning, our method does not compute probabilities for each link in the state transition table. Rarely accessed parts of a web site (e.g., pages "about this web site") or rarely used configurations of web clients can thus be tolerated. Of course, the DFA itself is induced from an observed sample of requests, and we expect the more common examples to be present in the sample. Because the distance measure is itself a form of generalization, it is possible for our system to miss attacks even if they were not present in observed normal traffic. We address the security impact of this "feature" in Section 6

### *3.3.1 Nonstationarity*

To track web sites that change over time, the learned model needs to update itself automatically. When a normal HTTP request arrives that is not captured by the current DFA but has a high similarity value, the DFA is modified to accommodate the request. The threshold for this operation is controlled by a parameter specified by the system administrator; the threshold would normally be set near 1.0, so that only requests that are very similar to the DFA are learned.

To detect unused edges, a counter is updated every time an edge is traversed. The counters are used during periodic pruning sweeps when infrequently used edges are deleted. After each sweep, all counters are reset to 0. A parameter controls how aggressive the pruning pass is, and if it is set to 0, only edges that have not been used since the last pruning pass are deleted. This allows recently added edges to survive for one full interval between prunings.

### 3.4 *Heuristics for reducing false positives*

It is desirable to reduce false positives as much as possible without compromising the system's ability to detect attacks. After studying the system in operation, we observed some consistent patterns exist in the false alarms, and based on these regularities we have developed two heuristics that are applied only to anomalous requests.

Many false positives are caused by HTTP requests with unusual lines. Because these lines are not critical for the web server to identify the requested resource, we delete the following HTTP header line on at a times: Referer, Cookie, Accept-Language, Accept-Charset, and Accept. If, after deleting a line, the request passes the similarity test, then it is accepted and processed without the anomalous header lines. Aside from cookies, the worst potential impact on a user is that a web client might receive the default version of a web page instead of one customized to its preferred language, character set, or file format.

Deleting cookies is potentially more serious, because they might encode state (e.g., the PHP session identifier cookie) that is required for proper operation of the web site. Deleting a cookie could interfere with the user's ability to visit the web site. Some web clients send cookies to web sites that do not match the cookie's list of sites that may read and modify it. These unexpected cookies then cause the request to be identified as abnormal.

We added a second heuristic after noticing a web robot which came online during testing, and was not observed in the training data. The one robot was responsible for nearly 10,000 false positives because its request format was different from the normal requests. To account for such situations, this second heuristic groups putative attacks into classes. When a request is classified as being abnormal, it is compared to the DFAs in a set of DFAs representing attacks. If the anomalous request is similar enough (controlled by a parameter we call the "grouping threshold"), it incorporated into the DFA representing a class of attacks. If it is not similar to any of the existing attack classes, it is used to start a DFA for a new class of putative attacks. The comparisons and additions to DFAs are as previously described. By grouping related unusual requests, an administrator could look at a single exemplar of the class and determine if it is acceptable. If so, the request could be added to the main DFA in the normal way, and future similar requests would be accepted as normal. Note that a similar heuristic was also independently developed by Robertson et al. [10].

| Web site | Files | Database | Requests |
|---|---|---|---|
| cs.unm.edu | 181,132 | none | 390,950 |
| aya.org | 5,095 | MySQL | 40,149 |
| explorenm.com | 6,146 | PostgreSQL | 36,944 |
| i-pi.com | 6,644 | none | 7,694 |

Table 1
The four web sites used for testing. **Files** is the number of distinct files on the web site, **Requests** is the number of HTTP requests in the training dataset.

## 4 Test data

We used four production web sites to collect our normal data, and then we developed our extensive database of attacks for testing our system.

### *4.1 Normal Data*

The training and test data are sets of HTTP requests from four web sites:

**aya.org** uses PHP extensively for dynamic content and a MySQL database.
**explorenm.com** uses Perl extensively and some PHP for CGI scripts, as well as a PostgreSQL database.
**i-pi.com** is a static website that only serves files.
**cs.unm.edu** contains official departmental content and a diverse set of student and faculty web pages, ranging from simple content to complex, automatically generated content.

The cs.unm.edu data were collected from November 2004 through February 2005. The aya.org, explorenm.com, and i-pi.com (henceforth referred to as the pugillis data, after the server hosting these domains) data were collected January and February of 2005. The data are broken into data sets; each data set represents approximately one week, with the exact time being determined by filesystem limits on the number of subdirectories. The data sets for a web site are of similar sizes. For results in this paper, the cs.unm.edu training data set was 2004-11-12 and the test data set was 2004-11-17; the pugillis training data set was 2005-01-25 and the test data set 2005-01-29. Table 1 shows the sizes of each of the four web sites

Each data record includes the entire HTTP request sent by the client to the server, allowing us to make use of the HTTP header lines and test for attacks that are not contained in the requested resource path. For the cs.unm.edu data, the attacks were filtered through a combination of automatic (using *snort* plus custom scripts) and manual inspection. The attacks were saved for later testing. The training and

testing phases could therefore be run on data sets with or without harmless attacks (i.e., attacks detected by *snort* but that were targeting programs or exploits that the cs.unm.edu web server was not vulnerable to). The attacks in the pugillis data sets were discarded before their importance was realized.

### *4.2 Attack Data*

The attack database currently contains a collection of 65 attacks, some of which are different examples of the same vulnerability, either a different exploit for the same vulnerability or an exploit for the vulnerability on a different operating system. We collected the attacks from the following sources: Attacks against web servers we were testing (attacks in the wild); BugTraq archives [39]; the SecurityFocus vulnerability database [40]; The Open Source Vulnerability Database [41]; The Packetstorm archives [42]; and Sourcebank [43]. In many cases, we had to debug the attack programs in order to get them to produce malicious web requests. However, we did not verify whether the attacks could actually compromise the targeted web application.

The attack database contains the following categories of attacks: Buffer overflows; Input validation errors (other than buffer overflows); Signed interpretation of unsigned values; and URL decoding errors. The attacks were against a collection of different web servers: Active Perl ISAPI; AltaVista Search Engine; AnalogX SimpleServer; Apache with and without mod_php; CERN 3.0A; FrontPage Personal Web Server; Hughes Technologies Mini SQL; InetServ 3.0; Microsoft IIS; NCSA; Netscape FastTrack 2.01a; Nortel Contivity Extranet Switches; OmniHTTPd; and PlusMail. The victimized operating systems for the attacks include the following: AIX; Linux (many varieties); Mac OS X; Microsoft Windows; OpenBSD; SCO UnixWare; Solaris x86; Unix; VxWorks; and any x86 BSD variant. A snapshot of the database we are using is at
`http://www.i-pi.com/HTTP-attacks-JoCN-2006`.

This attack database is more extensive than those used in earlier work on web server intrusion detection. For example, researchers using the Lincoln Labs data [44,45] have at most four attacks to work with. Kruegel and Vigna [15] used 11 attacks, some of which were chosen specifically because they targeted software available on the web servers used to collect training data. Tombini et al. [37] report using 56 attacks to test a portion of their IDS. Note that the only one of these that is publicly available is the Lincoln Labs data.

| Web site | Edges | Nodes |
|---|---|---|
| cs.unm.edu | 61,574 | 7,843 |
| aya.org | 13,310 | 1,800 |
| explorenm.com | 8,213 | 1,645 |
| i-pi.com | 5,819 | 1,269 |

Table 2
DFA sizes for filtered training data sets. **Edges** and **Nodes** show the size of the induced DFA.

## 5 Experiments

In this section, we report some experimental results obtained by using the training and test datasets just described. The experiments test performance in terms of model size and accuracy (true positives and false positives), for both the case of data in which attacks were filtered and the case where they were unfiltered. We then also present results regarding the specificity of our model in order to illustrate the limited amount of generalization our method performs on web requests.

Table 2 shows size of DFA learned from the training data. For the cs.unm.edu unfiltered data, the induced DFA has 62,897 edges and 7,854 nodes. The alphabet in the tests includes the values of most tokens; hence it is large. Note that the DFA edge count is well below the maximum of $|\Sigma|^2$.

Accuracy results for IDSs are often reported using a receiver operating characteristic (ROC) curve. Figure 6a shows the values for filtered data for all four web sites, and Figure 6b compares filtered and unfiltered data for the cs.unm.edu data. True positives are the fraction of the attack set properly identified, while false positives are the fraction of the test data (containing no attacks) that were improperly identified as attacks. Each point represents a different threshold for normal, and each set of connected points represents a different configuration of the algorithm. In a ROC curve, a perfect IDS would have a point at $(0, 1)$ indicating correct identification of both attacks and normal traffic.

The false positive rate depends on the similarity threshold and the grouping threshold, parameters a system administrator could vary to change the ratio of true positives to false positives. The false-positive rate for the four sites on filtered data is shown in Table 3. We report false positives per day, because this measure reflects the actual load that a system administrator would perceive when using the system. For the cs.unm.edu unfiltered data, the false-positive rate per day was 39.6 (versus 21.20 on filtered data) and the true-positive fraction was 0.79 (versus 0.92). This performance degradation is discussed in Section 6.

To evaluate how general were the models learned by our method, we examined how

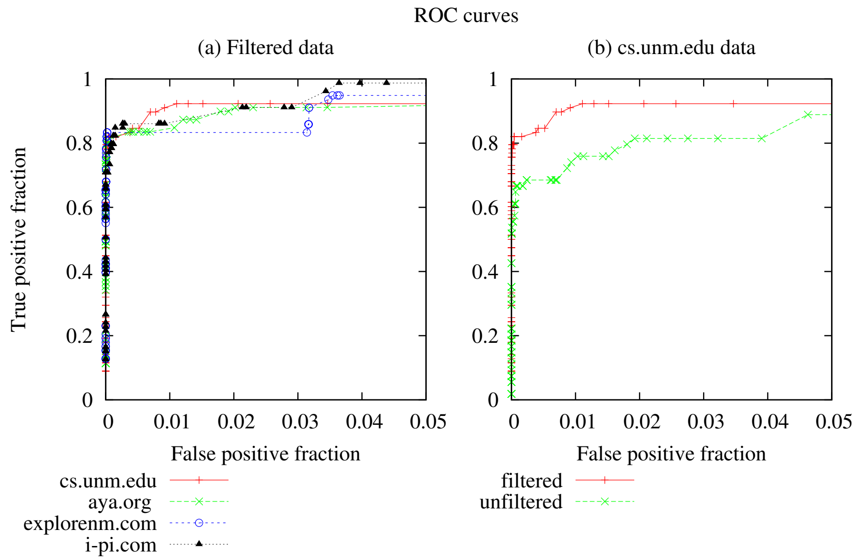



Fig. 6. Receiver operating characteristic curves: False versus true positives for different similarity thresholds. (a) Filtered data for all four web sites. (b) Filtered and unfiltered data from the UNM CS department web site.

| Web site | False positives per day | True positive fraction |
|---|---:|---:|
| cs.unm.edu | 21.20 | 0.92 |
| aya.org | 2.25 | 0.90 |
| explorenm.com | 0.50 | 0.97 |
| i-pi.com | 1.00 | 0.99 |

Table 3
False positive rates per day for four production web sites on filtered data. The results were obtained using a similarity threshold of 0.9 and a grouping threshold of 0.65.

well a model built for a website performed when monitoring another website. Table 4 shows that the DFA built from training data for one web site tends to perform poorly on the other web sites, showing that the a model induced from one site is not general enough to apply to other websites. In comparison, Table 5 shows that a simple statistic such as request length does not vary much between sites.

## 6 Discussion

In this section we interpret the experiments and explore their significance, focusing on the accuracy of our method in terms of false and true positives, the significance

| Train on: | cs.unm.edu | | aya.org | | i-pi.com | | explorenm.com | |
|---|---|---|---|---|---|---|---|---|
| **FP** | **frac** | **/day** | **frac** | **/day** | **frac** | **/day** | **frac** | **/day** |
| **Test on:** | | | | | | | | |
| cs.unm.edu | 0.007 | 251 | 0.127 | 4551 | 0.081 | 2904 | 0.094 | 3367 |
| aya.org | 0.040 | 33 | 0.005 | 27 | 0.005 | 27 | 0.024 | 119 |
| i-pi.com | 0.116 | 96 | 0.098 | 81 | 0.003 | 2 | 0.087 | 72 |
| explorenm.com | 0.123 | 102 | 0.101 | 437 | 0.014 | 61 | 0.014 | 61 |

Table 4
DFA performance across web sites: The DFA induced from each web site is tested against the other web sites. The units are first false positive fraction, then false positives per day. The results are for a continually learning DFA with similarity threshold of 0.852 and not using the GRP heuristic.

| | cs.unm.edu | aya.org | i-pi.com | explorenm.com |
|---|---|---|---|---|
| **mean** | 343 | 323 | 347 | 328 |
| **stddev** | 126.68 | 115.08 | 136.54 | 93.25 |

Table 5
HTTP request length (in characters) statistics for the four web sites. These data are from the filtered data sets.

of our modeling strategy and how it relates to other approaches to anomaly detection, and the suitability of the method for production systems.

### *6.1 Accuracy*

The accuracy of our system has notable limitations from the perspective of both true and false positives. Table 3 shows a significant number of false positives in a day, over 21 a day in one environment, and almost 40 per day when attacks are not filtered from normal data. The site in question, however, was complex, because it combines a large website for an academic department with class websites, faculty websites, and student websites. Given this complexity and variability, high false positives may be inevitable. The results were much better on the other, more stable sites. In contrast, in the experiments not all of the tested attacks were detected, either in the filtered or the unfiltered data. The performance reduction between the filtered and unfiltered results in Figure 6b is largely an artifact of the lower number of tested attacks: the number of attacks missed is approximately the same in both cases, although not exactly the same attacks were missed.

This variable performance illustrates a common pattern we observed during testing: Although we were never able to detect all of the attacks in a given test, the attacks that were missed depended in subtle ways on the normal training data. For example,

one of the missed attacks (a “Beck” attack) consisted primarily of a URL with hundreds of /’s in a row. This attack was classified as normal in some tests because a user error in a normal request had included two slashes (/) in a path name. This example created a link from the / token to itself in the DFA, allowing the attack URL’s repeated /’s to be classified as normal. In other tests where this training example was not present, the attack was detected. In this case it is clear the DFA generalized too much.

In other cases, however, manual inspection of a missed attack request showed that it was very close to normal requests. These attacks may not be amenable to request-level anomaly detection; however, they may be detectable by other anomaly detection approaches such as system call or user-level modeling. Indeed, rather than insisting that every method for anomaly detection detect every attack, we believe it is more prudent to ask whether a new method detects attacks that are missed by other methods. By this standard, the results are quite promising. For example, Wang and Stolfo [33] only tested on four attacks; Kruegel et al. [15,36] studied 12 attacks, and these were all based on parameters for CGI scripts (which are only 40% of our attacks). We have extended this work by studying 65 attacks, almost all of which are detected in at least one test. Note that no other type of anomaly detection system has been shown to be able to detect most of the attack types that we examined.

Nevertheless, there is room for improvement. Both false and true positives could likely be improved by combining multiple models of web requests, as proposed by Kruegel et al. [15,36]. As a component in such a framework, the DFA approach could potentially extend the framework’s coverage.

### *6.2 Modeling variable data*

Our approach can be summarized as: “Remember past events in as much detail as possible, and use specific rules to reduce variation in the less predictable parts of the data stream.” This approach contrasts with earlier work, in which one or more simple, low information content models are used to characterize normal behavior. Our strategy resembles that used in Forrest et al. [6]. In that work only one heuristic was used to reduce variation (ignore system call arguments), and here we have developed several, more specialized heuristics. The DFA method can produce larger models in terms of storage space than other methods; however, the expanded use of space directly translates into faster execution for both learning and detection. On modern machines such a space/time tradeoff is feasible and often desirable.

Because future events are rarely identical to the past, some generalization is essential for any anomaly detection system. Our DFA induction method overgeneralizes in some circumstances and does not generalize enough in others. Both issues

are recurring problems for anomaly-detection systems, particularly in the context of malicious training and mimicry attacks [46]. Having said this, our results give some some support for the intuitive observation that our method generalizes less than those proposed by other researchers. Specifically, Table 4 shows that the DFA can distinguish between web sites, showing that the learned models are site-specific (and hence not very general). Given that our learned DFAs include site-specific information including pathnames, this result is not surprising. However, note that most statistics used by other researchers do not directly incorporate site-specific information, but rather make generalizations that may or may not vary between sites. As shown in Table 5, measures such as request length do not appear to vary much between sites (at least in our limited sample); other statistics, though, may vary more. Thus, more work is required to determine how general in practice the models produced by other methods [33,15,36] relative to ours and to explore how the degree of generalization affects true positives, false positives, and susceptibility to mimicry attacks. Further investigation of these issue is planned for future work.

Although the work reported in this paper relies on manually developed parser heuristics to reduce variability, this task lends itself to automation. After building a DFA from a set of examples, the program could search for nodes with a high out degree. For these nodes, if most of the outbound edges have a low usage count (e.g., $< 10$), then the DFA is attempting to memorize a highly variable portion of the protocol. Once these candidates are identified, then statistical techniques such as those used by Kruegel et al. [36] could be used to characterize the variable portion. Or, a human could use the standards to determine the legal structure. This automated approach for finding high-variability regions and replacing them with less variable information could be used in other anomaly detection environments.

### *6.3 Suitability for production use*

The technique described in this paper could be implemented as part of an online anomaly-based IDS because of three key factors: the Burge DFA induction algorithm has low complexity, it produces compact DFAs on real-world data when combined with the parser heuristics, and it can learn on a set of easily obtained normal requests sent to web servers without requiring pre-filtering of attacks. We envision such an IDS taking the form of a web proxy. Normal requests (including those that have a line deleted) are passed to the server. Abnormal requests would either be blocked or sent to a separate, less-functional but more secure sever. The proxy approach would also be useful because it can determine from the web server reply whether or not a request is valid. Currently our data set includes improper paths, syntax errors, and other examples of the problems that web client, web proxy and web robot designers have in properly following the HTTP standards. And, the resulting DFAs include this behavior. By avoiding the behavior of known illegal requests, we should be able to generate even smaller DFAs.

There are several outstanding issues to be resolved before a proxy server implementation could be deployed. How long should the system be trained? How do we automatically generate parser heuristics? When do we have sufficient confidence to rewrite or block a request? Although these are all challenging and important issues (particularly in the context of an autonomous, self-repairing system), experience with another online anomaly-based IDS [9] leads us to believe that they are tractable. Because a proxy implementation would also provide a useful framework for testing and comparing approaches, we see this as a promising area for future work.

## 7 Other related work

In general, learning algorithms for one-class, nonstationary problems like the security anomaly detection problem are rare, although this is an area of current interest in the machine learning community (e.g., see [31,47,48]). Methods for handling nonstationary data include forgetting, as described by Salganicoff [49]. Other machine learning algorithms, such as neural networks with adaptive critics, have been proposed as an anomaly detection method [50]. However, these methods require a fixed input size, a limitation that prevents use with HTTP requests. Littman and Ackley [51] looked at cases where the problem can be divided into two parts, variable and invariant, although this approach would not apply to our environment, where little is invariant.

In the worst case, learning a DFA from only positive evidence is known to be NP-complete [52], but Lang showed that the average case is tractable [53]. In practice, DFA induction is feasible in many contexts, and there is an extensive literature on DFA induction algorithms [54,55,56]. While the Burge DFA induction algorithm is novel in its simplicity and in the type of generalization it performs, other DFA induction algorithms not requiring negative examples may also be suitable for modeling web requests.

Others have used representations that could be considered a directed graph (or used to generate a directed graph). Forrest et al. used sequences of system calls to represent behavior of programs [6]; sets of sequences implicitly define a DFA of acceptable behavior. Wagner and Dean used a directed graph to represent the system calls made by a protected program [57]. Other researchers (e.g., [58,29,26]) have studied anomaly detection using methods for constructing finite automata-based models of program behavior. In the web server protection domain, Kruegel and Vigna [15] tried a Markov model for representing the characters in CGI parameters. However, they found that most transitions in their Markov model were rare; thus, they instead used their Markov model as a zero/non-zero to test if the structure of the CGI parameters had been seen in training. By ignoring the magnitude of transition probabilities, they in effect used their Markov model as a DFA.

## 8 Conclusion

Protecting web servers is a challenging and important problem, especially in the context of custom web-based applications. We described a method for detecting anomalous web requests via DFA induction that meets many of the requirements of an autonomous security solution, including efficient online unsupervised one-class learning, tolerance for highly variable normal behavior that contains benign attacks, and detection of a wide variety of attacks on web applications. We have also developed and released an extensive corpus of attack examples that should help other researchers develop better web application defenses.

Although the current implementation has limitations both in terms of attack detection and false alarms, we have demonstrated that the principle of modeling normal data as precisely as possible, with variability being excluded only when necessary, is an important and viable approach to anomaly detection. We forsee that systems that combine multiple models based on similar principles will eventually be accurate enough to deploy as part of security-oriented, autonomous self-healing systems.

## 9 Acknowledgments

SF and KI gratefully acknowledge the support of the National Science Foundation (grants ANIR-9986555, CCR-0331580, and CCR-0311686), Defense Advanced Research Projects Agency (grants F30602-02-1-0146), and the Santa Fe Institute. AS gratefully acknowledges the support of Natural Sciences and Engineering Research Council of Canada (NSERC) and MITACS.

## References

[1] J. O. Kephart, A biologically inspired immune system for computers, in: Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, MIT Press, Cambridge, MA, US, 1994, pp. 130–139, ftp://coast.cs.purdue.edu/pub/doc/genetic_algorithms/bio/immune.ps.Z Accessed 31 May 2002.

[2] S. Forrest, A. S. Perelson, L. Allen, R. Cherukuri, Self-nonself discrimination in a computer, in: 1994 IEEE Computer Society Symposium on Research in Security and Privacy, 16-18 May 1994, Oakland, CA, USA, Los Almitos, CA, USA : IEEE Computer Society Press, 1994, 1994, pp. 202–212.

[3] S. A. Hofmeyr, S. Forrest., Immunizing computer networks: Getting all the machines

in your network to fight the hacker disease., in: 1999 IEEE Symposium on Security and Privacy, 1999, pp. 9–12.

[4] S. A. Hofmeyr, An immunological model of distributed detection and its application to computer security, Ph.D. thesis, University of New Mexico, Computer Science Department (May 1999).

[5] P. Williams, K. Anchor, J. Bebo, G. Gunsch, G. Lamont, CDIS: Towards a computer immune system for detecting network intrusions, in: Lecture Notes in Computer Science 2212, Springer-Verlag, 2001, pp. 117–133, presented at the 4th International Symposium on Recent Advanced in Intrusion Detection (RAID 2001). http://en.afit.edu/ISSA/publications/WilliamsRAID.pdf Accessed 19 August 2002.

[6] S. Forrest, S. Hofmeyr, A. Somayaji, T. Longstaff, A sense of self for Unix processes., in: 1996 IEEE Symposium on Security and Privacy, 6-8 May 1996, Oakland, CA, USA, IEEE Computer Society Press, Los Alamitos, CA, USA, 1996, pp. 120–128.

[7] S. A. Hofmeyr, S. Forrest, A. Somayaji, Intrusion detection using sequences of system calls, Journal of Computer Security 6 (3) (1998) 151–80, http://cs.unm.edu/˜forrest/publications/int_decssc.pdf Accessed 13 March 2002.

[8] A. Somayaji, S. Forrest, Automated response using system-call delays, in: Proceedings of the 9th USENIX Security Symposium, USENIX Association, Berkeley, CA, US, 2000, pp. 185–197, http://www.cs.unm.edu/˜soma/pH/uss-2000.pdf Accessed 31 May 2002.

[9] A. Somayaji, Operating system stability and security through process homeostasis, Ph.D. thesis, University of New Mexico, http://www.cs.unm.edu/˜soma/pH/uss-2000.pdf Accessed 31 May 2002. (2002).

[10] W. Robertson, G. Vigna, C. Kruegel, R. A. Kemmerer, Using generalization and characterization techniques in the anomaly-based detection of web attacks, in: Network and Distributed System Security Symposium Conference Proceedings: 2006, Internet Society, 2006, http://www.isoc.org/isoc/conferences/ndss/06/proceedings/html/2006/papers/anomaly_signatures.pdf Accessed 12 February 2006.

[11] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, N. Weaver, Inside the slammer worm, IEEE Security and Privacy 01 (4) (2003) 33–39.

[12] J. P. Anderson, Computer security technology planning study, Tech. Rep. ESD-TR-73-51, United States Air Force, Electronic Systems Division (October 1972).

[13] T. F. Lunt, Detecting Intruders in Computer Systems, in: 1993 Conference on Auditing and Computer Technology, 1993,

http://www.sdl.sri.com/papers/c/a/canada93/canada93.ps.gz Accessed 22 August 2002.

[14] L. T. Heberlein, G. V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, D. Wolber, A network security monitor, in: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Press, 1990.

[15] C. Kruegel, G. Vigna, Anomaly detection of web-based attacks, in: Proceedings of the 10th ACM conference on Computer and communications security, ACM Press, 2003, pp. 251–261.

[16] M. Roesch, Snort—lightweight intrusion detection for networks, in: 13th Systems Administration Conference—LISA '99, 1999, pp. 229–238, http://www.usenix.org/events/lisa99/roesch.html Accessed 30 June 2002.

[17] S. Patton, W. Yurcik, D. Doss, An Achilles' heel in signature-based IDS: Squealing false positives in SNORT, in: RAID 2001 Fourth International Symposium on Recent Advances in Intrusion Detection October 10–12, 2001, Davis, CA, USA, 2001, available online only. http://www.raid-symposium.org/raid2001/papers/patton_yurcik_doss_raid2001.pdf Accessed 3 January 2003.

[18] M. J. Ranum, A network firewall, in: Proceedings of the First World Conference on System Administration and Security, SANS Institute, 5401 Westbard Ave. Suite 1501, Bethesda, MD 20816, 1992.

[19] G. W. Treese, A. Wolman, X through the firewall and other application relays, in: Proceedings of the USENIX Summer Conference, USENIX Association, Berkeley, CA, 1993, ftp://crl.dec.com/pub/DEC/CRL/tech-reports/93.10.ps.Z Accessed 2002 Feb 20.

[20] M. Handley, V. Paxson, C. Kreibich, Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics, in: Conference Proceedings: 10th USENIX Security Symposium, USENIX Association, Berkleley, CA, USA, 2001, pp. 115–131, http://www.aciri.org/vern/papers/norm-usenix-sec-01.pdf Accessed 2002 Feb 20.

[21] E. Strother, Denial of service protection: the nozzle, in: Annual Computer Security Applications Conference, 11-15 Dec. 2000, New Orleans, LA, USA, IEEE Computer Society, Los Alamitos, CA, USA, 2000, pp. 32–41, http://www.acsac.org/2000/papers/41.pdf Accessed 2002 Feb 20.

[22] C. Ko, G. Fink, K. Levitt, Automated detection of vulnerabilities in privileged programs by execution monitoring., in: Tenth Annual Computer Security Applications Conference, 5-9 Dec. 1994, Orlando, FL, USA, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994, pp. 134–144.

[23] P. G. Neumann, P. A. Porras, Experience with EMERALD to date, in: First USENIX Workshop on Intrusion Detection and Network Monitoring (ID'99), 9–12 April 1999,

Santa Clara, CA, USA, USENIX Association, Berkeley, CA, USA, 1999, pp. 73–80, http://www.sdl.sri.com/papers/det99/ Accessed 20 August 2002.

[24] S. A. Hofmeyr, S. Forrest, Architecture for an artificial immune system, Evolutionary Computation 7 (1) (2000) 1289–1296.

[25] J. Balthrop, S. Forrest, M. Glickman, Revisiting lisys: Parameters and normal behavior, in: Proceedings of the 2002 Congress on Evolutionary Computation, 2002, http://www.cs.unm.edu/~forrest/publications/rev_lysis.ps Accessed 19 August 2002.

[26] C. Marceau, Characterizing the behavior of a program using multiple-length n-grams, in: New Security Paradigms Workshop 2000, 18–22 Sept. 2000, Ballycotton, Ireland, ACM, New York, NY, USA, 2001, pp. 101–10, http://www.atc-nycorp.com/papers/Marceau_multiLengthStrings.pdf Accessed 13 August 2002.

[27] W. Lee, S. J. Stolfo, Data mining approaches for intrusion detection, in: Proceedings of the 7th Usenix Security Symposium, Usenix Association, 1998.

[28] C. Warrender, S. Forrest, B. A. Pearlmutter, Detecting intrusions using system calls: Alternative data models, in: IEEE Symposium on Security and Privacy, 1999, pp. 133–145.

[29] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, A fast automaton-based method for detecting anomalous program behaviors, in: IEEE Symposium on Security and Privacy, IEEE, 2001, pp. 144–155.
URL http://citeseer.ist.psu.edu/sekar01fast.html

[30] C. Kruegel, D. Mutz, F. Valeur, G. Vigna, On the detection of anomalous system call arguments, in: ESORICS 2003: 8th European Symposium on Research in Computer Security, Vol. 2808 of Lecture Notes in Computer Science, Springer, 2003, pp. pp. 326–343.

[31] M. V. Mahoney, P. K. Chan, Learning nonstationary models of normal network traffic for detecting novel attacks, in: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM Press, 2002, pp. 376–385.

[32] M. V. Mahoney, Network traffic anomaly detection based on packet bytes, in: Proceedings of the 2003 ACM Symposium on Applied computing, ACM Press, 2003, pp. 346–350.

[33] K. Wang, S. J. Stolfo, Anomalous payload-based network intrusion detection, in: Recent Advances in Intrusion Detection: 7th International Symposium, RAID 2004, Sophia Antipolis, France, September 15-17, 2004. Proceedings, Vol. 3224 of Lecture Notes in Computer Science, Springer, 2004, pp. 203–222.

[34] R. Vargiya, P. Chan, Boundary detection in tokenizing network application payload for anomaly detection, in: Proceedings of the ICDM Workshop on Data Mining for Computer Security (DMSEC), 2003, pp. 50–59, workshop held in conjunction with The Third IEEE International Conference on Data Mining. Available at

http://www.cs.fit.edu/~pkc/dmsec03/dmsec03notes.pdf.
Accessed 5 April 2006.

[35] R. Lippmann, J. Haines, D. Fried, J. Korba, K. Das, The 1999 DARPA off-line intrusion detection evaluation, Computer Networks 34 (4) (2000) 579–95.

[36] C. Kruegel, G. Vigna, W. Robertson, A multi-model approach to the detection of web-based attacks, Comput. Networks 48 (5) (2005) 717–738.

[37] E. Tombini, H. Debar, L. Mé, M. Ducassé, A serial combination of anomaly and misuse IDSes applied to HTTP traffic, in: 20th Annual Computer Security Applications Conference, 2004.

[38] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext transfer protocol—HTTP/1.1, RFC 2616. ftp://ftp.isi.edu/in-notes/rfc2616.txt Accessed 2002 Oct 2. (June 1999).

[39] SecurityFocus, What is bugtraq?, http://online.securityfocus.com/popups/forums/bugtraq/intro.shtml Accessed 10 January 2003.

[40] SecurityFocus, Vulnerabilities, http://www.securityfocus.com/vulnerabilities Accessed 24 April 2006. (2005).

[41] Open Source Vulnerability Database (OSVDB), Osvdb: The open source vulnerability database, http://www.osvdb.org/ Accessed 24 April 2006.

[42] Packet Storm, Packet storm: Know better, http://packetstorm.linuxsecurity.com/ Accessed 24 April 2006.

[43] Jupitermedia, Sourcebank: The search engine for developers, http://archive.devx.com/sourcebank/ Accessed 24 April 2006.

[44] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, M. Zissman, Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation, in: DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings, Volume: 2, 1999, pp. 12–26.

[45] R. Lippmann, J. Haines, D. Fried, J. Korba, K. Das, Analysis and results of the 1999 DARPA off-line intrusion detection evaluation., in: H. Debar, L. Me, S. Wu (Eds.), Recent Advances in Intrusion Detection. Third International Workshop, RAID 2000, 2–4 Oct. 2000, Toulouse, France, Springer-Verlag, Berlin, Germany, 2000, pp. 162–82.

[46] D. Wagner, P. Soto, Mimicry attacks on host-based intrusion detection systems, in: Proceedings of the 9th ACM conference on Computer and Communications Security, ACM Press, 2002, pp. 255–264.

[47] L. M. Manevitz, M. Yousef, One-class svms for document classification, J. Mach. Learn. Res. 2 (2002) 139–154.

[48] N. V. Chawla, N. Japkowicz, A. Kotcz, Editorial: special issue on learning from imbalanced data sets, SIGKDD Explor. Newsl. 6 (1) (2004) 1–6.

[49] M. Salganicoff, Density-adaptive learning and forgetting, in: International Conference on Machine Learning, 1993, pp. 276–283.

[50] D. V. Prokhorov, D. C. Wunsch II, Adaptive critic designs, IEEE Transactions on Neural Networks 8 (5) (1997) 997–1007.

[51] M. L. Littman, D. H. Ackley, Adaptation in constant utility non-stationary environments, in: R. K. Belew, L. B. Booker (Eds.), Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, CA, 1991, pp. 136–142.

[52] E. Gold, Complexity of automaton identification from given data, Information and Control 37 (3) (1978) 302–320.

[53] K. J. Lang, Random DFA's can be approximately learned from sparse uniform examples, in: Proceedings of the Fifth ACM Workshop on Computational Learning Theory, ACM, New York, N.Y., 1992, pp. 45–52.
URL citeseer.ist.psu.edu/lang92random.html

[54] O. Cicchello, S. C. Kremer, Inducing grammars from sparse data sets: a survey of algorithms and results, J. Mach. Learn. Res. 4 (2003) 603–632.

[55] K. J. Lang, B. A. Pearlmutter, R. A. Price, Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm, Lecture Notes in Computer Science 1433, proceedings of ICGI-98.
URL
http://citeseer.ist.psu.edu/article/lang98results.html

[56] A. L. Oliveria, J. Silva, Efficient search techniques for the inference of minimum sized finite automata, in: Proceedings of the Fifth String Processing and Information Retrieval Symposium, IEEE Computer Press, 1998, pp. 81–89.

[57] D. Wagner, D. Dean, Intrusion detection via static analysis, in: Proceedings of the 2001 IEEE Symposium on Security and Privacy, 2001, available at http://www.csl.sri.com/users/ddean/papers/oakland01.pdf. Accessed 21 April 2006.

[58] A. P. Kosoresow, S. A. Hofmeyr, Intrusion detection via system call traces, IEEE Software 14 (5) (1997) 35–42.