2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

pSigene: Webcrawling to Generalize SQL Injection Signatures

Gaspar Modelo-Howard
Narus
gmhoward@narus.com

Christopher N. Gutierrez, Fahad A. Arshad, Saurabh Bagchi, Yuan Qi
Purdue University
{*faarshad,gutier20,sbagchi,alanqi*}*@purdue.edu*

***Abstract*—Intrusion detection systems (IDS) are an important component to effectively protect computer systems. Misuse detection is the most popular approach to detect intrusions, using a library of signatures to find attacks. The accuracy of the signatures is paramount for an effective IDS, still today's practitioners rely on manual techniques to improve and update those signatures. We present a system, called *pSigene*, for the automatic generation of intrusion signatures by mining the vast amount of public data available on attacks. It follows a four-step process to generate the signatures, by first crawling attack samples from multiple public cybersecurity web portals. Then, a feature set is created from existing detection signatures to model the samples, which are then grouped using a biclustering algorithm which also gives the distinctive features of each cluster. Finally the system automatically creates a set of signatures using regular expressions, one for each cluster. We tested our architecture for SQL injection attacks and found our signatures to have a True and False Positive Rates of 90.52% and 0.03%, respectively and compared our findings to other SQL injection signature sets from popular IDS and web application firewalls. Results show our system to be very competitive to existing signature sets.**

***Keywords*-web application security; signature generalization; biclustering; SQL injection;**

I. Introduction

Network intrusion detection systems (NIDS) are an important and necessary component in the security strategy of many organizations. These systems continuously inspect network traffic to detect malicious activity and when this happens, send alerts to system administrators. One type of NIDS, called *misuse-based detector*, uses signatures of attacks to inspect the traffic and flag the malicious activity. But a potential problem faced by these signature-based NIDS is that as new attacks are created and as new kinds of benign traffic are observed, the signatures need to be updated. *The current approach to this process is manual*. Consequently, keeping them updated is a Herculean task that involves tedious work by many security experts at organizations that provide the NIDS software. A big drawback of the signature-based schemes that has been pointed out by many researchers and practitioners [20], [9] is that due to their relatively static nature, they miss zero-day attacks. These are attacks that target hitherto unknown vulnerabilities and consequently, no signature exists for such attacks. Our goal in this work is to automatically generate signatures by performing data mining on attack samples. Further, we aim to create generalized signatures; "generalized" implies the signatures will be able to match some zero-day attacks as well, not just the attack samples that it has been trained on. We use SQL injection attacks for developing and demonstrating our method.

We look at the rulesets of three popular misuse-based detectors—Snort [37], Bro [28], and ModSec [41]. From these, we observe the reflection of the ad hoc and manual nature of the signature creation (and update) process. We observe that many rulesets include signatures that are too specific, many are disabled by default, and several show clear overlaps. For example, 70% of the almost 20,000 signatures included in the ruleset for Snort (snapshot 2920) are disabled by default. Also, several sets of signatures contained in the Snort ruleset file `sql.rules` could be merged as one. For example, signatures with identifiers 19439 and 19440 have the same regular expression, except for the last character and hence could easily be merged. Additionally, we found multiple examples of signatures using very simple regular expressions, which increases the risk of generating false positive (FP) alerts. As an example, several rules in Snort use the regex `.+UNION\s+SELECT` to detect SQL injection attacks, by searching for the SQL statements `UNION` and `SELECT`. The problem is that such search might be too simple as these statements are also commonly found in benign database queries from web applications.

In this paper, we propose a solution for the automatic creation of generalized signatures represented as regular expressions, by applying a sequence of two data mining techniques to a corpus of attack samples. The goal is to make the process less manual (and tedious) and target detection of zero-day attacks. We call our solution *pSigene* (pronounced `psy-gene`), which stands for **p**robabilistic **Si**gnature **gene**ration. *pSigene* follows a four-step process. In the first step, it crawls multiple public cybersecurity portals to collect attack samples. In the second step, it extracts a rich set of features from the attack samples. In the third step, it applies a specialized *clustering* technique to the attack sample (training) data collected in step 2. The clustering technique also gives the distinctive features for each cluster. In the last step a generalized signature is created for each cluster, using *logistic regression* modeling, which is trained

This work was done while G. Modelo-Howard was a graduate student in the Department of Electrical and Computer Engineering, Purdue University.

978-1-4799-2233-8/14 $31.00 © 2014 IEEE
DOI 10.1109/DSN.2014.21

both on attack sample data (from step 2) and some benign traffic data. Logistic regression gives a probabilistic classifier — given a new sample, it can tell with a probability value what is the likelihood of the sample belonging to any given cluster.

Regular expressions (regexs) are a structural notation for describing similar strings. Regexs are a powerful tool used to define languages, per the automata theory definition. Current NIDS have incorporated the usage of regexs to generalize signatures, so variations of attacks can be detected. We adopt the use of regexs for our generalized signatures.

Most of the efforts to date to automate the signature creation process has been related to malware activity [44], [19], such as for worms and botnets. This landscape is different from ours in that we target attack steps that have a small "distance" from legitimate activity. Consider for example, an SQL injection attack in which a small variation in the `where` clause, such as a tautology `1==1`, followed by a comment demarcation symbol "`;`", can cause a legitimate-looking SQL query to become an attack sample. Second, we consider activities where the feature set of each sample is very rich. For example, we first started with 477 features for SQL injection attacks, corresponding to various keywords, symbols *and their relative placements*. The rich feature set poses challenges and constraints on the kinds of machine learning techniques that can be used.

We demonstrate our solution specifically with SQL injection attacks (shortened as SQLi attacks). Although there exist elegant preventive solutions to solve this problem, like parameterizing SQL statements [4], dynamic bracketing [38], taint inference [36], and escaping special SQL characters [18], in practice it seems elusive to completely implement such solutions. SQLi attacks have been very dominant in the last few years, being used in high-profile cases such as intrusions to large technology organizations [14], government agencies [2], and software companies [23], [8]. Signatures to improve detection mechanisms, such as what *pSigene* delivers, are necessary as they complement prevention mechanisms.

pSigene effectively suggests the number of signatures necessary to detect the attacks while helping to reduce the size of each signature, in terms of the number of features necessary to define each signature. In our experiments, *pSigene* collected a set of 30,000 attack samples from which we manually extracted a set of 159 features and then *pSigene* created nine signatures, all but one of which required 14 or fewer features. For testing, we used the popular SQL injection tools SQLmap [7], Arachni [1] and Vega [39]. The experimental results showed that our signature set was able to detect 86.53% of all attacks while only generating 0.037% of false positives. This is a higher true positive rate for SQLi than Snort (79.55%) and Bro (73.23%), which use manually created and progressively refined signatures. Bro had no false positive while Snort had about 5X false positives compared to *pSigene*. ModSecurity however performed better than *pSigene* with a true positive rate of 96.07%, and a false positive rate slightly worse (0.0515% compared to our 0.037%). We also compared our technique to an existing signature generation algorithm [29] and found that its TPR was very low but it had no false positive at all.

The contributions of our work are:

1) An automatic approach to generate and update signatures for misuse-based detectors.
2) A framework to generalize existing signatures. The detection of new variations of attacks is achieved by using regular expressions for the generalized signatures.
3) We rigorously benchmark our solution with a large set of attack samples and compare our performance to popular misuse-based IDS-es and a web application firewall. Our evaluation also brings out the impact of practical use case whereby periodically new attack samples are fed into our algorithm and consequently the signatures can be progressively, and automatically, updated.

The remainder of this paper is structured as follows: Section II presents the threat model used along with the different components for the proposed framework. Section III describes the dataset used to evaluate *pSigene*, the evaluation results along with a comparison to existing open-source rulesets and to another signature generation algorithm. We also determine the performance implications of using our approach. A discussion follows in Section IV about the usage scenarios and limitations of our approach. Then we give an overview of previous approaches to automatically generating signatures and detecting attacks through interactions between web services and databases. We end with some conclusions and future work.

II. Design

The goal of *pSigene* is to generate generalized signatures from traces of attack samples and non-malicious network traffic. As shown in Figure 1, the generation of the signatures involves four phases. First multiple public cybersecurity portals on the Internet are crawled to collect attack samples. In the second step, a set of features is extracted from the attack samples. The third step calls for the sample set to be grouped using a *clustering* technique. This step also gives the features that distinguish each cluster. In the final step, a generalized signature is created for each cluster, using *logistic regression* modeling. The process allows to create signatures that represent a set of similar attacks, reducing the number of rules handled by a NIDS.

To develop our system, we consider SQL injection (SQLi) attacks. They have been a very relevant and popular attack vector for the last few years. IBM [16] reported that in 2012 the majority of the security incidents detected on their monitored clients around the world were attributed

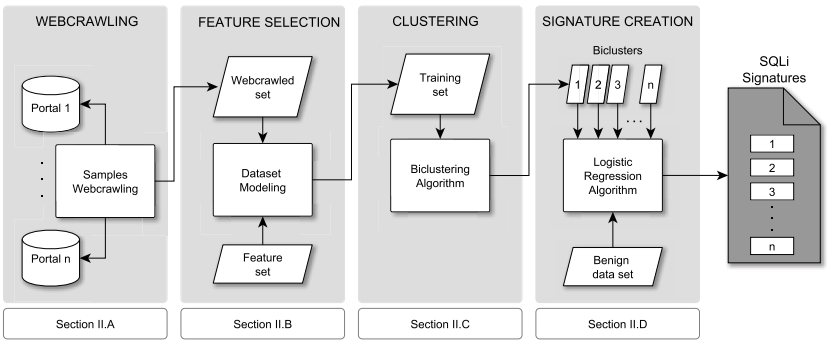

Figure 1. Components of the pSigene architecture, a webcrawl-based creation process for SQLi attack signatures. For each component, there is a reference to the section providing further details. It is shown below each component.

to SQLi. Also, injection attacks were the primary cause of incidents, making SQLi a primary attack vector for hackers. To consider this attack type, our threat model assumes attacks against web applications, connected to a database (commonly known as three-tier system). An attacker starts by having a publicly accessible description of the system and then browses the web application, looking for forms where she can provide user input and then this input can serve as parameters for an SQL statement.

A. Webcrawling for Attack Samples

The first phase is to crawl the web to collect attack samples that later are used to generate the generalized signatures. The objective is to take advantage of the multitude of public web sources available that provide attack samples. This approach looks to proactively collect samples from multiples web sources, which is the opposite of a more common strategy to use honeypots to collect attack samples.

We chose to proactively collect samples because we are targeting slow moving attacks (such as SQLi), they present a greater diversity than typically handled by honeypots, the distance between legitimate requests and malicious requests is often quite small, and above all, for a purely logistical reason — to speed up the data collection. Our approach was facilitated in practice by the wide availability of well-maintained data sources of SQLi attack samples, some of which provide APIs to enable automated sample collection. A practical point here is that what we see during the web crawling is the entire HTTP request payload and we extract the SQL query from it by leaving out the HTTP address, the port, and the path (typically a ? indicates the start of the query string).

To collect the attack samples, we crawled different cybersecurity portals between April and June of 2012. Each portal or site is a public repository of computer security tools, exploits, and security advisories, where security professionals and hackers share examples of different attacks. Examples of cybersecurity portals include Security Focus [35], the Exploit Database [34], PacketStorm Security [27], and the Open Source Vulnerability Database [13]. This last site also provides its own search API, making it easy for security practicioners and researchers to automate the collection process of data on those sites. For sites that do not provide such capability, one can use the APIs provided by search engines, such as Google custom search API [15].

Table I
EXAMPLES OF SQLI VULNERABILITIES PUBLISHED IN JULY 2012.

VULNERABILITY	CVE ID
Joomla 1.5.x RSGallery 2.3.20 component	CVE-2012-3554
Drupal 6.x-4.2 Addressbook module	CVE-2012-2306
Moodle 2.0.x mod/feedback/complete.php 2.0.10	CVE-2012-3395
RTG 0.7.4 and RTG2 0.9.2 95/view/rtg.php	CVE-2012-3881

There are also open forums and mailing lists where users share attack samples. In our experiments, we collected over 30,000 SQLi attack samples and used these as our dataset to generate the generalized signatures during the experiments.

It is important for our signature generalization approach to work effectively that the sample collection be as comprehensive as possible. As one heuristic-based check for this, we manually inspected the high and medium risk SQL injection vulnerabilities published during the month of July 2012 in the National Vulnerability Database [25] for web applications using the MySQL database — approximately 30 in number. In each case, we found examples of SQLi attacks in our dataset that could be launched against each of the web applications reviewed. Table I includes some examples of the SQLi vulnerabilities published in July 2012 and for which, we found attack samples in our collected dataset.

Once the attack samples are collected, we use a set of 5 transformations, including uppercase → lowercase, URL

Table II
SOURCES OF SQLI FEATURES.

FEATURE SOURCE	EXAMPLES	DESCRIPTION

MySQL Reserved Words	`create`
	`insert`
	`delete`
		Words are reserved in MySQL and require special treatment for use as identifiers or functions.

NIDS/WAF Signatures	`in\s*?\(+\s*?select`
	`\)?;`
	`[^a-zA-Z&]+=`
		SQLi signatures from popular open-source detection systems are deconstructed into its components.

SQLi Reference Documents	`' ORDER BY [0-9]-- -`
	`/\*/`
	`\"`
		Common strings found in SQLi attacks, shared by subject matter experts.

encoding → ascii characters, and unicode → ascii characters.

B. Feature Selection

We characterize each sample using a set of features, which will be used as input for the clustering algorithm. To create the set of features, we use three sources that are domain-specific for the SQLi attack scenario: (1) SQL reserved words [26], (2) SQLi signatures from the Bro and Snort IDS and the ModSecurity web application firewall (WAF), and (3) SQLi reference documents [33], [6]. A summary of the feature sources is presented in Table II.

The SQL reserved words are used as features since they represent identifiers or functions, necessary to create SQL queries like in SQLi attacks. Examples of reserved words used to create the feature set for SQLi attacks include `SELECT`, `DELETE`, `CURRENT_USER`, and `VARCHAR`. For this paper, we limited the feature set to only include the reserved words for the MySQL database management system, thus excluding special-purpose keywords used in Microsoft SQL and other non MySQL databases.

We also looked at existing signatures for features since the signatures are the result of a usually long optimization process, so it could be assume that these signatures have components (strings inside a signature) that can be used as features to help identify attacks. We did not use a whole signature as a single feature, but rather divided the signature into logical components and each component then was used as a feature. To achieve this, we used metacharacters such as parentheses `()` and the alternation operator `|` that delimit logical groups and branches inside a regular expression.

As an example, let's consider a signature taken from the ModSecurity Core Rule Set (CRS), defined as a regular expression with seven case insensitive groups, joined by the alternation operator: `(?:<group 1>)|...|(?:is\s+null)|(?:like\s+null)|...|(?:<group 7>)`. In this case, we created seven features, one for each regex group in the signature.

Our choice allowed for our system to also consider the relative position of SQL tokens among them, when creating the features. As an example, feature `=[0-9%]+` only considers a number if it is preceeded by the = character.

All features included in the set were of numeric type, each one measuring the number of times a feature was found in an attack sample. The resulting feature set used in the experiments had 159 entries (from an initial set of 477), after removing those features that were not found in any of the samples used in the training phase of the system. The removed features also corresponded to cases for attacks to non-MySQL databases (not considered in our experiments) or because of multiple features looking for similar SQLi strings (overlapping features).

70 (out of 159) entries in the resulting feature set performed as binary features. That is, the value for each of these features was either one (confirming the existence of the corresponding SQL token or string in a sample) or a zero (non existence) in each of the attack samples.

The process of creating the feature set might at first blush seem intensely manual. But in our experience, the process was automatable for the most part. Both the reserved words and the fragmentation of the existing signatures (rows 1 and 2 in Table II) could be automated since they follow from unambiguous rules. In the case of analyzing the reference documents, this was partially automated and served more to validate features created with the other sources. Additionally, we believe that the feature space was exhausted so the creation of the feature set should be considered a one-time task, for each kind of attack (such as SQLi).

We also considered using only binary features, i.e., the binary flag whether a feature is present or absent in a sample, rather than its count. However, this did not produce good results.

Each attack sample that provides the input to the clustering algorithm later used is characterized by its values for the 159 features. The resulting data is organized in a matrix where the samples are the rows of the matrix and the features are the columns. The size of the matrix was then 30,000 by 159 and can be classified as sparse because 85% of its cells were populated with zeroes. About 6% of its cell values were ones.

C. Creating Clusters for Similar Attack Samples

We use the biclustering technique [30] to analyze our matrix, which is popularly used in gene expression data analysis. The objective of this technique is to identify blocks in the sample dataset built by a subset of features to characterize a subset of samples. Given a set of m rows and n columns (i.e., an $m \times n$ matrix), the biclustering algorithm generates biclusters - a subset of rows which exhibit similar behavior across a subset of columns. To achieve this, the biclustering technique first clusters the rows (samples) of

the matrix and then clusters the columns (features) of the row-clustered data.

To formalize the concept of bicluster, a sample set D is given as a $|N| \times |F|$ matrix where N is the set of samples and F is the set of features. The elements d_{ij} of the matrix indicate the relationship between sample i and feature j. Then, a bicluster B_{RC} is a block that includes a subset of the rows $R \subseteq N$ and a subset of columns $C \subseteq F$, sharing one or more similarity properties.

The objective of using biclustering is to identify subsets of attack samples which share similar values for a subset of features. Each subset of samples (cluster) may use different sets of features. We want to create a signature for each bicluster and the biclustering technique allows using different features for different biclusters. This enables us to create compact and distinctive signatures for the wide variety of SQLi attacks. The biclusters are nonoverlapping (i.e., no two biclusters have spatial overlap) and nonexclusive (i.e., two biclusters may use overlapping set of features) (Figure 2). The heatmap shows eleven clusters that are formed, by visual analysis of the color patterns. A contiguous region with one color pattern constitutes one cluster. Note that not all features are used in the cluster formation; thus, there are some gaps for the feature dimension when you consider all the clusters. Note also that not all samples are covered in a cluster, indicating that some attack samples are considered so different that they do not fit within any cluster. This may indicate that our training set has some noise in it. Being able to deal with some noise in a training set is an important property for any machine learning algorithm and we are heartened to see that that is the case with *pSigene*.

We use a simple approach to achieve the biclustering technique, performing a two-way hierarchical agglomerative clustering (HAC) algorithm, using the Unweighted Pair Group Method with Arithmetic Mean (UPGMA). The way biclustering worked is first it did a clustering of the samples and then within each cluster, it clustered by the features. Thus, it identified what were the discriminating features for each cluster.

The UPGMA algorithm produces a hierarchical tree, usually presented as a dendrogram, from which clusters can be created. It works in a bottom-up (agglomerative) approach by first partitioning the sample set of size N into N clusters, each one containing a single sample. Then, the Euclidean pairwise distance is calculated among the initial, single sample clusters in order to merge the two closest ones. After the first round of paired clusters finishes, UPGMA is used to recursively merge the clusters. At each step, the nearest two clusters are combined into a higher-level cluster. The distance between any two clusters A and B is taken to be the average of all distances between pairs of objects "x" in A and "y" in B. This biclustering process is repeated until a single cluster containing all the samples is formed. Note that this is just the termination point from biclustering; its results will guide us to pick the multiple clusters as we explain below.

Table III
FEATURES INCLUDED IN SIGNATURE 6

FEATURE NUMBER	FEATURE (Regular Expression)
25	`=`
37	`=[-0-9\%]*`
53	`<=>|r?like|sounds\s+like|regex`
36	`([^a-zA-Z&]+)?&|exists`
28	`[\?&][^\s\t\x00-\x37|]+?`
32	`\)?;`

The results from applying the biclustering technique to the webcrawled dataset are presented as a heat map in Figure 2. On each axis, the corresponding dendrograms are also shown. The heat map shows the graphical representation of the reordering of the matrix $|N|\times|F|$ into a set of bi-clusters. Each bicluster is represented as an area of similar color as the heat map simultaneously exposes the hierarchical cluster structure of both rows and columns, as explained in [10]. Each column in the matrix is standardized as follows: the statistical mean and standard deviation of the values is computed. The mean is then subtracted from each value and the result divided by the standard deviation. As a value is closer to the mean, it is shown with the black color in the heat map. The highest and the lowest values are shown in red and green, respectively. Figure 2 also shows the dendrograms produced by the HAC algorithm for both rows (sample set) and columns (feature set).

To validate the accuracy of the HAC algorithm, we also calculated the cophenetic correlation coefficient for each dendrogram. The cophenetic correlation for a cluster tree is defined as the linear correlation coefficient between the cophenetic distances obtained from the tree, and the original distances (or dissimilarities) used to construct the tree. Thus, it is a measure of how faithfully the tree represents the dissimilarities among observations. In our experiments, we found the cophenetic correlation coefficient value of 0.92, a promisingly high number.

Ultimately, the above-mentioned explorations of the design space required visual inspection of multiple heatmaps rather than the alternative: use of multiple security experts and an almost *zen master*-like grasp of regular expressions.

D. Creation of Generalized Signatures

From each bicluster b_j, we create a signature Sig_{b_j} which characterizes the samples in that bicluster, plus is more generalized. Specifically, in our solution, a signature Sig_{b_j} is a logistic regression model built to predict whether an SQL query is an attack similar to the samples in cluster b_j.

Logistic regression is a very popular classification method since the output values for the hypothesis function, lay in the range between 0 and 1. These values are interpreted as the estimated probability that a sample belongs to a class.

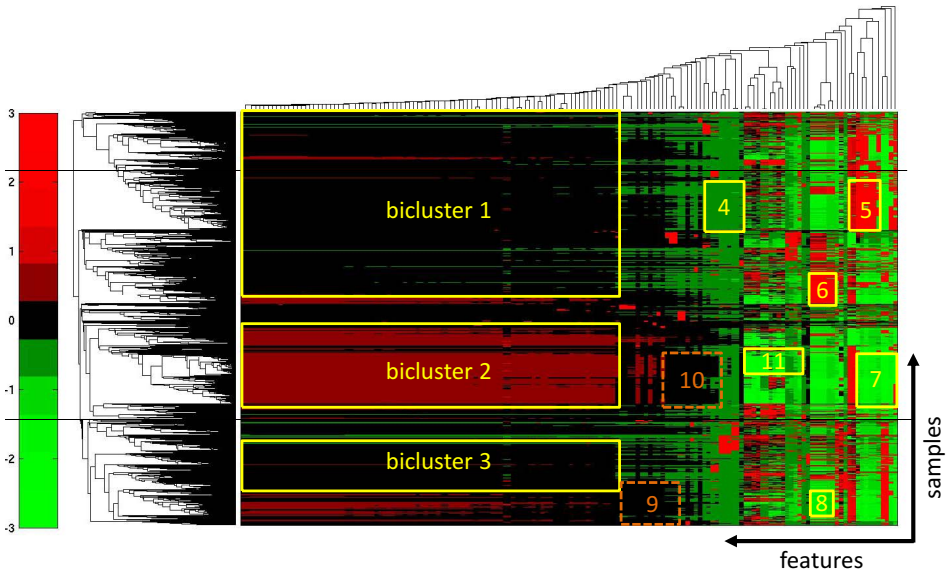

Figure 2. Heat map with two dendrograms of the matrix data representing the samples dataset. The 30,000 attack samples are the rows and the 159 features are the columns. The heat map also shows the eleven biclusters selected. Signatures were not produces for biclusters 9 and 10 as they were 'black holes' (did not contained sufficient training data).

Each bicluster b_j is defined by a set of features F_j and a set of samples S_j. We then create the corresponding signature Sig_{b_j} of each bicluster by using the features as the variables in the hypothesis function and training this function with the samples from the bicluster, as well as normal traffic.

pSigene calculates the parameters Θ_j (which is a vector of individual parameter values), using the labeled data of attack samples from cluster b_j as well as benign network traffic data. The intuition behind the calculation of Θ_j is that it should minimize the errors in the labeled training set.

Having calculated Θ_j, let us see how *pSigene* would work during the operational phase (the test phase). When a sample i is available to *pSigene*, to determine if it belongs to attack class j, it calculates the value of the hypothesis function:

$$h_\theta(F_{ij}) = g(\Theta_j^T F_{ij})$$

where F_{ij} represents the values of sample i for the feature set F_j. We use for g the sigmoid function which is defined as:

$$g(z) = \frac{1}{1+e^{-z}}$$

This gives a value between 0 and 1 and is interpreted as the probability that the test sample i belongs to attack class j.

We used the Preconditioned Conjugate Gradients (PCG) method [11] to find the optimal parameters Θ of the regression model for each bicluster.

As an example of how we used logistic regression in the SQL injection attack scenario, consider a bicluster b_6 obtained after running the biclustering algorithm. This bicluster has a set S_6 of $2,741$ samples and a set F_6 of the features listed in Table III. After training with the set S_6 (attack class) and one day of non-malicious traffic (other class), we compute the parameters Θ_6 of the generalized signature for bicluster S_6:

$$\begin{aligned}\Theta_6^T = \; & -3.761054 + 0.262131 f_{6,25} + 0.262131 f_{6,37} \\ & + 0.261463 f_{6,53} + 0.261584 f_{6,36} \\ & - 0.117270 f_{6,28} + 0.708324 f_{6,32}\end{aligned}$$

III. Evaluation

We implemented *pSigene* using the popular Bro IDS and evaluated it along the signatures in three other IDSes by using SQL attack samples and benign web traffic. We also compared our technique to another proposed algorithm for signature generation.

A. SQLi Signature Sets

We analyzed three different sets of SQLi signatures, taken from two popular open-source IDS (Snort and Bro) and one web application firewall (ModSec). A summary of the different signatures used in the evaluation is presented in Table IV. The fact that some of the SQLi rules are disabled by default in some of the IDSes may indicate the perception that there exists overlaps between rules. The high usage of regex is because it holds the promise that a regex will be able to match a wide set of attacks. This observation motivated us to build on regex's in choosing the features in *pSigene*. A description of each signature set follows:

Table IV
COMPARISON BETWEEN DIFFERENT SQLI RULESETS.

RULES DISTRIBUTION	VERSION	NUMBER SQLi RULES	SQLi RULES ENABLED	USAGE OF REGEX
Bro	2.0	6	100%	100%
Snort Rules	2920	79	61%	82%
Emerging Threats	7098	4231	0%	99%
ModSecurity	2.2.4	34	100%	100%

Bro A network analysis framework, Bro can be used as a signature-based IDS. It comes with a set of signatures to perform low-level pattern matching. We analyzed the 6 SQLi rules present on Bro v2.0 [28] to detect SQLi attacks. All six of the rules make extensive usage of regular expressions.

Snort / Emerging Threats (ET) Snort is an open source network IDS that performs packet-level analysis and comes with its own ruleset. Emerging Threats is an open source project that publishes detection rulesets for IDS such as Snort. For our experiments, we merged Snort version 2920 [40] and ET version 7098 rulesets [12]. Over 98% of those rules use simple regular expressions.

ModSecurity (shortened as "ModSec") is a web application firewall (WAF) used to protect Apache web servers from attacks such as SQLi. The OWASP ModSecurity Core Rule Set (ModSec CRS) project is an open-source initiative to provide the signatures used by ModSecurity to detect attacks to web applications. We analyzed CRS version 2.2.4.

Snort and Bro use a deterministic approach to handle the signatures. In other words, these systems produce an alert only if all the requisites defined in a signature are met. In contrast, ModSecurity takes a probabilistic approach and uses a scoring scheme where signatures are weighted and can contribute to determine the level of anomaly for a particular trace.

The average length of the 6 signatures (regular expressions) found in Bro was 247.7 characters (max: 429, min: 27). Meanwhile, regular expressions in ModSecurity had an average length of 390.2 characters (max: 2917, min: 28) and in Snort were 27.1 (max: 40.1, min: 5).

For all the experiments presented in this paper, we considered only those signatures that used or included regular expressions, which was the overwhelming majority of the rules. We did this to allow for a fairer comparison between *pSigene* and the other IDSes, since *pSigene* uses regular expressions for its features.

B. Datasets

For training, we used the 30,000 attack samples collected by crawling public sources (as detailed in Section II-A) and 240,000 HTTP samples for normal traffic. We used three test datasets to evaluate the performance of the different signature sets, one to determine the FPR and two for the TPR. The test dataset used to compute FPR corresponds to a 1-week network trace at a university institution. We captured all HTTP traffic to the main web servers at the university, including the institutional web servers, the registration and payment servers, and the web interface for the mailing servers. The network trace amounts to 4.53 GB and included over 1.4 million HTTP GET requests. Although no ground truth existed for this trace, we ran it through all three signature sets and manually reviewed the alerts generated. All alerts were false positives; therefore we concluded no malicious attack was included in the trace. Also, no incidents were reported during this time by the network's managers.

A second testing dataset was used to compute the TPR of all signature sets. We generated this testing dataset by running SQLmap [7], a popular SQL injection scanning tool, against a vulnerable web application [43] running Apache Tomcat and MySQL database. SQLmap was launched against the application which contained 136 vulnerabilities, triggering the scanning tool to generate over 7200 attack samples. To collect this testing dataset, we set up an isolated network which only had the test traffic and thus the traces were not contaminated with other traffic.

A third testing dataset was created to further determine the TPR of all the signature sets. Using two tools, Arachni [1] and Vega [39], we generated another SQLi dataset of 8578 samples and used it to test the detection rate (TP) of all the signatures sets. We will refer to the Arachni and Vega together as the Arachni set since we do not present the results separately for them due to space reasons and because they provide similar insights. The use of three different tools to generate the TPR testing sets, with their different methods for generation of attack samples, was important to our evaluation strategy to assess the generality of *pSigene* in detecting a variety of SQLi attacks.

C. Implementation in Bro

To run our experiments, we implemented the signatures generated by *pSigene* into the Bro IDS and then instructed Bro to use only our signatures and not its own. To achieve this, we coded a function `count_all()` that accepted as input two parameters, a regular expression and a string, and returned the number of times the regular expression was found in the string. *pSigene* is invoked by Bro from its upper policy layer, which is analogous to where Bro's own signatures reside.

D. Experiment 1: Accuracy and Precision

We performed the evaluation separately with seven signatures (corresponding to seven biclusters, labeled 1 to 7 in Figure 2) and with nine signatures (adding biclusters 8 and 11 to previous set of seven biclusters). The set of seven signatures obtained a higher TPR than Bro and Snort, while also producing a very low FPR. The results from the set of nine signatures allowed determining how much the TPR can be improved while also measuring the increase in the FPR.

Table V
ACCURACY COMPARISON BETWEEN DIFFERENT SQLI RULESETS.

RULES	TPR (%) (SQLmap)	TPR (%) (Arachni)	FPR (%)
ModSecurity	96.07	98.72	0.0515
pSigene (9 signatures)	86.53	90.52	0.037
pSigene (7 signatures)	82.72	89.48	0.016
Snort - Emerging Threats	79.55	76.59	0.1742
Bro	73.23	76.33	0.0000

We visually identified eleven biclusters from the heatmap using a rule of 5%. That is, for any bicluster we selected from the heatmap, it would have to include at least 5% of all samples in the training dataset. This permitted to include a large percentage of all the samples in the original training set, while giving reasonably homogeneous colored areas.

From the list of eleven biclusters selected, *pSigene* discarded those considered as *black holes* which are defined as biclusters composed of vectors of mostly zeroes. These biclusters will show on the heat map in black color and more than 99% of all the features values in corresponding samples, are zeroes. In our experiments, biclusters 9 and 10 were black holes (shown in Figure 2) so no signatures were generated from them.

The results are shown in Table V for all testing sets. From the SQLmap set, our signatures had higher detection rate (86.53% for 9 signatures and 82.72% for 7 signatures) than Snort and Bro, but lower than ModSecurity (96.07%) and a similar result was obtained from the Arachni set. Both our signature sets had the lowest FPRs, only behind Bro's signature set (which did not raise a single false positive). Although the other FPRs were very low, one should not be deceived by these numbers. A FPR of 0.174%, as recorded for Snort, represents over $2,463$ false alarms generated over the one week traffic, while ModSec's represents over 730 false alarms. In comparison, our sets produced 523 false alarms in the case of nine signatures and 226 in the case of seven signatures.

ModSecurity achieved the highest TPR of all signatures sets. We had suspected this to be a difficult result to improve upon. The ModSec set is part of a popular open source WAF tool and has been manually developed by expert security administrators (we had personal communication with the lead developer of the project, confirming this belief that we had already obtained through reading of public discussion groups). The resulting set is a group of complex regular expressions, making it difficult for regular system administrators to update it when new vulnerabilities are discovered and to adjust it to the traffic of a particular network.

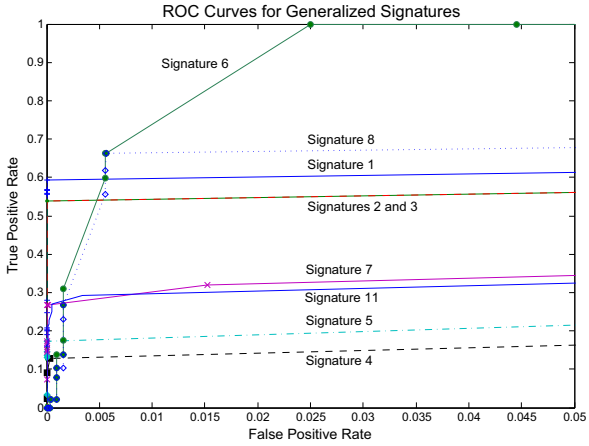

Figure 3. ROC curves for each of the signatures generated for the generalized set. The plot shows different performance for each signature, suggesting that each one can be tuned separately which can improve the overall detection rate of the set.

Accuracy and Precision of Individual Signatures

We wanted to drill deeper into the overall accuracy and precision result of *pSigene* to see what the contribution from each of the signatures is. For this, we plotted the ROC curves for each of the 9 signatures for the entire test data. The result is shown in Figure 3. To generate the ROC curve for a given signature, we ran *pSigene* with only that signature enabled and we varied the probability threshold for the output of logistic regression. In the ROC curve, the point (0, 1) corresponds to the ideal case and the greater is the area under the curve, the better the performance is. Note that in this plot, the FPR only goes till 0.05, not till 1. This is because the maximum value of FPR for the systems under test does not grow beyond 0.05.

The first observation is that there is wide variability in the quality of the signatures. Signature 6 performs well while signature 4 lags. Second, signatures 1, 2, 3, and 8 are quite insensitive to the threshold settings. Third, signature 6 will produce false positives faster than signatures 1 and 8. From a ROC curve like this and with an idea of a desired TPR and FPR, a security administrator can visually, and approximately, decide which signatures to enable or disable.

Coverage of Individual Signatures

Another aspect of the clusters and the corresponding signatures is how many samples does each cover and how many features are used in each cluster's signature. The results are shown in Table VI. There is quite a large range of cluster sizes and number of features. The largest cluster has 44% of the samples while the smallest has 5.5%. Three clusters use 57% of the total number of features (90 out of 159). However, an interesting, and *not a priori* obvious, observation is that logistic regression does significant amount of pruning of features for these three clusters. Thus, logistic

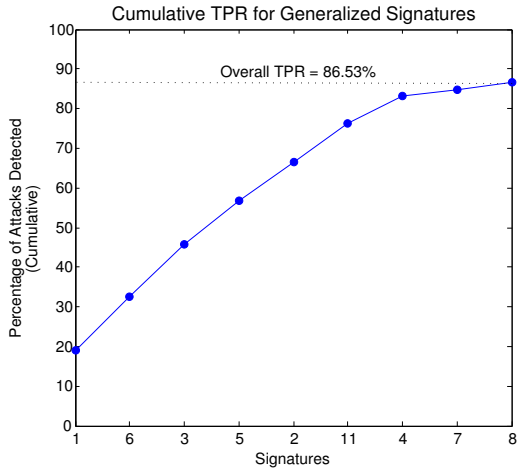

Figure 4. Cumulative TPR for set of 9 signatures produced by pSigene. Seven signatures contributed between 7 and 19 percent of the total TPR. Signatures 7 and 8 contributed 1.64 percent each.

regression downplays the role of some features in classifying a sample as being malicious or benign. For example, for cluster 3, logistic regression throws out 88% of the features, for cluster 2 86% of the features, and for cluster 1 63% of the features. We hypothesize that this large amount of filtering by logistic regression is due to two causes. First, the reduction of the feature set from 477 to 159 is a manual process and there still remain overlaps between some of them. Second, logistic regression is focused on picking features that help to classify while biclustering has that only as an indirect goal. Nevertheless, biclustering is a crucial step and needs to precede logistic regression. Biclustering creates some order out of the chaos of the large amount of samples and large set of features, by identifying the samples which are similar and by identifying a superset of features according to which they are similar.

Figure 4 shows the contributions of the individual signatures (for the 9 cluster case) toward the TPR. The signatures are arranged in descending order of their quality. The plot shows that signatures are of differing qualities, but all of the signatures make non-trivial contribution toward the overall TPR—signature 1 contributes the most at 19%, while signature 8 contributes the least with 1.63%.

E. Experiment 2: Incremental Learning

In this experiment, we first incremented the number of attack samples while learning the Θ parameters in logistic regression to create the signatures. We progressively added some attack samples from the test dataset into the training dataset - we experimented with 20% and 40% of the test dataset being included in the training. This reflects the real world scenario where fresh attack samples will be fed to *pSigene* to do incremental training with these new samples. Thus, over time, *pSigene* will be able to detect more and more of the attacks as it operates for longer periods and gets incrementally trained. Note that the incremental training is also an automatic process and therefore, we are spared the tedium of manually updating prior signatures.

Table VI
DETAILS OF SIGNATURES FOR EACH CLUSTER CREATED BY *pSigene*.

BICLUSTER	NUMBER OF SAMPLES	NUMBER OF FEATURES (BICLUSTERING)	NUMBER OF FEATURES (SIGNATURE)
1	13272	90	33
2	5477	90	13
3	2629	90	11
4	6947	12	8
5	4245	8	5
6	2741	6	6
7	3928	10	5
8	1676	8	6
11	1671	15	14

When adding 20% of the SQLmap dataset, we obtained a TPR $= 89.13\%$ and a FPR $= 0.039\%$. After augmenting the training dataset with 40% of the samples from the SQLmap set, the TPR increased to 91.15% while the FPR also increased to 0.044%. In both cases (20% and 40%), we used sets of 9 signatures.

From the results, the TPR showed an increment of a bit over 2%, for each round of the experiment. This can be explained as we first randomized the SQLmap set and then divided it into 20% parts. So one can hypothesize that *pSigene* is seeing some similar attack samples in the test phase.

F. Experiment 3: Comparison to Perdisci's Approach

We also compared *pSigene* to the approach presented in Perdisci *et.al.* [29] for the automatic generation of signatures for HTTP-based malware. We selected this approach as (1) it has as one of its claims the ability to create signatures for variants of malware that have not yet been seen, (2) it involves the analysis of HTTP traces like in our SQLi scenario, and (3) is based on the popular token-subsequence technique used to generate signatures for polymorphic worms [24].

As the current evaluation of pSigene involves a different type of attacks, SQL injection, two changes were made to the original technique in Perdisci's paper [1]. First, we did not implement the first step, coarse-grained clustering, as it was not applicable to our scenario. They clustered malware samples by looking at structural similarities among sequences of HTTP requests but in our case, each HTTP request is considered independently of the rest so the first step is not needed. In the second step, fine-grained clustering, we

[1]We contacted the authors but no implementation was available to us, so we re-implemented their technique for this evaluation. We faithfully followed the method outlined in the paper, with a few specializations which we outline here.

modified how an HTTP request was evaluated to compute the distance metric between pairs of requests. We used the same predefined weights (10 and 8) as in Perdisci, assigning them to the parameter values and names, respectively, and disregarded the method and path of a HTTP request. For the SQLi scenario, the parameter values include the actual SQL query and therefore represent the most important part of a URL when detecting this type of attack.

We used the same training set as pSigene to generate the signature set with the adjusted Perdisci implementation. Controlling the clustering process by using the DB validity index (Section 3 of [29]), 145 clusters were produced during the fine-grained clustering phase. We reduced the cluster set to 27 after removing clusters according to the presented technique, i.e., with a single sample or that produce signatures too short (such as `?id=.*`). At the end of phase 3, cluster merging, 10 signatures were produced. To merge different clusters, we chose a threshold of 0.1 as this meant that two signatures would only be merged if they were nearly identical. At all phases, the DB index was computed to validate our answers.

We tested the signatures produced with Perdisci's approach using the same malicious and benign sets used for other signature sets. Perdisci's achieved a TPR of 5.79% and a FPR of 0%. The FPR is very good as these signatures did not misclassify a single benign sample as malicious. The TPR on the other hand is very low. We were not expecting the TPR to be high as Perdisci's (and Polygraph) objective is not to produce generalized signatures but rather to create a single token subsequence that can represent the set of all samples previously seen . Still, the TPR shows the limitation faced from using approaches like this to detect variations of known attacks. As a counterpoint, when we used the same samples from the training set for the testing, it showed a TPR of 76.5%. This indicates that it is more successful in its objective of automatically creating signatures from already seen samples.

G. Experiment 4: Performance Evaluation

In this section, we report the overhead of *pSigene* signatures. Specifically, we measured the processing-time per HTTP request for each signature in the SQLmap dataset. We observe that *pSigene* reports minimum, average, and maximum processing times of 390, 995, and 1950 μsec, respectively. On average, *pSigene* gives a slowdown of $17X$ and $11X$ against Modsec and Bro signatures respectively. The increased processing-time in *pSigene* is majorly attributed to the `count_all()` function call, which counts the number of regex matches for each HTTP request string. We observe from the data that the signatures with a large number of invocations of `count_all()` take a disproportionately large fraction of the total processing time. Given that we run these measurements on a relatively resource-starved machine (700 MHz (CPU), 512 MB (RAM)) and still the worst case processing time was less than 2 ms, we would expect that signature matching in *pSigene* will not become a bottleneck. Importantly, the signature matching is completely parallelizable - each parallel thread can match one signature and this functionality is inbuilt in Bro (Bro's cluster mode). But we do not have this obvious performance optimization implemented yet.

IV. Discussion

This work throws light on the importance of good training data for creating the clusters and subsequently the signatures. It is imperative that the training data be representative of the kinds of attacks that will be seen in operation, though they do not need to be identical. How far apart can the attacks in training and test be? This is a perennial question that is asked of machine learning algorithms in all different contexts. This answer is probabilistic since our framework gives a probability value to how likely a sample is to belong to a cluster. The heartening insight from *pSigene*'s evaluation is that the match does not need to be exact and thus hitherto unseen attacks can also be detected. The flip side of this is that it suffers from some false positives, which in some deployments may be completely unacceptable. For specificity, consider the following example.

The set of regex patterns used in signature 4 of *pSigene* include `char`, `@`, `information_schema`, and `ch(a)?r\s*?\(\s*? \d`. Looking at the set of samples used to train for this signature, one can find SQLi samples similar to `?id=-1+union+ select+1,2,3, 4,concat(database(),char(58), user(), char(58),version()),6,7,8,9,10, 11, 12, 13, 14, 15, 16,17`, where there are two occurrences of patterns `char` and two of `ch(a)?r\s*?\(\s*?\d`. The testing set on the other hand include samples with different subsets of the regex patterns for signature 4 and for pattern `char` we found samples from zero to thirty occurrences.

The features should be chosen to be rich enough that they are likely to capture important characteristics of the zero-day attacks. Thus, signatures based on such features will likely be able to match some of the zero-day attacks. The feature selection process needs to be repeated for each kind of attack, but not for each attack sample. This makes this process more feasible in practice. In contrast, manual signature update is a process that needs to be done for each attack sample and is therefore not as scalable. Of course, in practice, a signature update is done in a batch mode after a certain number of attack samples have been collected.

V. Related Work

The work presented in this paper is related to three areas of intrusion detection: automatic signature creation, signature generalization, and the interaction between web

applications and databases. We discuss how previous work in these two areas relates to our research.

An important work on automatic signature creation is by Yegneswaran, et al [44]. The authors presented a framework based on machine learning to produce attack signatures. A key point by the authors and which we agree with is that the framework requires protocol knowledge in order to produce effective signatures and such insight impacts the resulting detection mechanism. Distinct from our work, they take a passive approach as HTTP and NetBIOS-based malware traffic is collected from honeynets. Additionally, our framework is agnostic to transport- and network-level information, which is important for their framework. Finally, we rely heavily on regular expressions, looking to produce rich, optimized regex signatures. Their approach to regexs is limited as it only uses simple metacharacters such as *, +, and ? to express clusters of signatures.

Previous work on signature generalization also includes [24], [19], and [22]. In [24], a system called Polygraph generates signatures that consist of multiple disjoint substrings. In doing so, Polygraph leverages our insight that for a real-world attack to function properly, multiple invariant substrings are often be present in the payload. Similarly, [19] applies pattern-matching techniques and protocol conformance checks on multiple levels in the protocol hierarchy to network traffic captured at a honeypot system, to produce worm signatures. [22] extends this idea to detect zero-day polymorphic worms on high-speed networks. In both cases, the goal is to detect worms at the network layer while our general approach considers protocol information and suited for other types of attacks.

Robertson et al. [32] present an anomaly generalization technique to automatically translate suspicious requests to a web server into anomaly signatures. This approach is complementary to ours and uses heuristics-based techniques to infer web-based attacks. For the class of SQL injection attacks, the technique performs a simple scan for common SQL language keywords and syntactic elements. This results in basic signatures to detect SQLi attacks, but no details were provided on the performance of these signatures. Other papers that present similar anomaly-based intrusion detection techniques for SQLi attacks include [17] and [21].

The interaction between web applications and databases to improve the detection rate of attacks against these resources has been covered in [3], [42], [5], and [31]. [3] et al. present a novel approach for automatically detecting potential server-side vulnerabilities of this kind in legacy web applications through blackbox analysis. [42] proposes a serially composed system with a web-based anomaly detection system, a reverse HTTP proxy, and a database anomaly detection system to increase the detection rate of web-based attacks.

VI. Conclusions and Future Work

In this work, we presented a system called *pSigene*, for the automatic generation and update of intrusion signatures. The system benefits from mining the vast amount of public data available on attacks. We tested our architecture for the prevalent class of SQLi attacks and found our signatures to perform very well, compared to existing signature sets, which have been created manually and with a tremendous amount of security expertise and progressive refinement over the period of multiple years.

Our framework allows one to generalize existing signatures and the detection of new variations of attacks (i.e., some kinds of zero-day attacks) is achieved by using regular expressions for the generalized signatures. We also rigorously benchmarked our solution with a large set of attack samples and compare our performance to popular misuse-based IDS-es. The evaluation also brings out the impact of a practical use case whereby periodically new attack samples are fed into our algorithm and consequently the signatures can be progressively, and automatically, updated. In contrast, to improve the other signature sets requires the manual inspection and testing of the signatures, which could overwhelm a system administrator with limited resources.

Future work will include the implementation of the incremental update operation. This task has some open design choices in terms of the machine learning technique to use and empirical evidence is needed to guide our choice. We will also improve the online performance of the signature matching process. This will be done first by simply parallelizing the process and next by optimizing the code path within Bro through which our signature matching occurs.

References

[1] ARACHNI. Web application security scanner framework. http://www.arachni-scanner.com, February 2013.

[2] BBC. Royal navy website attacked by romanian hacker. http://www.bbc.co.uk/news/technology-11711478, November 8, 2010.

[3] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proc. ACM CCS*, 2010.

[4] P. Bisht, A. P. Sistla, and V. Venkatakrishnan. Automatically preparing safe sql queries. In *Proc. IFCA Conf. Financial Cryptography*, 2010.

[5] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proc. ACM SOSP*, 2011.

[6] J. Clarke. *SQL Injection Attacks and Defense*. Syngress Publishing, 1st edition, 2009.

[7] B. Damele and M. Stampar. SQLmap. http://sqlmap.org, July 2012.

[8] DarkReading. Adobe hacker says he used sql injection to grab database of 150,000 user accounts, November 2012.

[9] R. Di Pietro and L. V. Mancini. *Intrusion Detection Systems*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[10] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proc Natl Acad Sci*, 95(25):14863–14868, Dec 1998.

[11] S. Eisenstat. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM Journal on Scientific and Statistical Computing*, 2(1):1–4, 1981.

[12] O. I. S. Foundation. Suricata intrusion detection and prevention engine. http://www.openinfosecfoundation.org, 2012.

[13] O. S. Foundation. Open source vulnerability database. http://www.osvdb.org, 2012.

[14] D. Goodin. New Sony hack exposes more consumer passwords. *The Register*, June 3, 2011.

[15] Google. Google custom search api. https://developers.google.com/custom-search, 2012.

[16] IBM. IBM X-Force 2012 trend and risk report. http://www-03.ibm.com/security/xforce/, March 2013.

[17] A. Kamra, E. Bertino, and G. Lebanon. Mechanisms for database intrusion detection and response. In *Proc. ACM SIGMOD IDAR*, 2008.

[18] R. Kaushik and R. Ramamurthy. Efficient auditing for complex sql queries. In *Proc. ACM SIGMOD*, 2011.

[19] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.*, 34(1):51–56, jan 2004.

[20] C. Kruegel. *Intrusion Detection and Correlation: Challenges and Solutions*. Springer-Verlag TELOS, 2004.

[21] S. Y. Lee, W. L. Low, and P. Y. Wong. Learning fingerprints for a database intrusion detection system. In *Proc. ESORICS*, 2002.

[22] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proc. IEEE Symp. Security and Privacy*, 2006.

[23] A. Moscaritolo. Oracle's mysql.com hacked via sql injection, March 28, 2011.

[24] J. Newsome, B. Karp, and D. Song. Polygraph: automatically generating signatures for polymorphic worms. In *Proc. IEEE Security and Privacy*, 2005.

[25] NIST. National vulnerability database. http://nvd.nist.gov/nvd.cfm, 2008.

[26] Oracle. Mysql 5.5 reference manual, rev. 31755, August 2012.

[27] PacketStorm. Packetstorm security portal. http://packetstormsecurity.org, 2012.

[28] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[29] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proc. USENIX NSDI*, 2010.

[30] A. Prelic, S. Bleuler, P. Zimmermann, A. Wille, P. Buhlmann, W. Gruissem, L. Hennig, L. Thiele, and E. Zitzler. A systematic comparison and evaluation of biclustering methods for gene expression data. *Bioinformatics*, 22(9):1122–1129, 2006.

[31] W. Robertson, F. Maggi, C. Kruegel, and G. Vigna. Effective anomaly detection with scarce training data. In *Proc. NDSS Symposium*, 2010.

[32] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In *Proc. NDSS Symposium*, 2006.

[33] R. Salgado. Websec sql injection pocket reference. http://goo.gl/FGAhc, 2011.

[34] O. Security. Exploit database. http://www.exploit-db.com, 2012.

[35] SecurityFocus. Bugtraq vulnerability database. http://www.securityfocus.com/vulnerabilities, 2008.

[36] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS Symposium*, 2009.

[37] Sourcefire. Snort IDS. http://www.snort.org, 2008.

[38] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proc. ACM POPL*, 2006.

[39] Subgraph. Vega, web application security platform. http://www.subgraph.com, February 2013.

[40] S. V. R. Team. Snort rules. http://www.snort.org/snort-rules, 2012.

[41] Trustwave. Modsecurity core rule set. http://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project, January 2012.

[42] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel, and E. Kirda. Reducing Errors in the Anomaly-based Detection of Web-Based Attacks through the Combined Analysis of Web Requests and SQL Queries. *J. Comp. Sec.*, 17(3), 2009.

[43] WAVSEP. Web application vulnerability scanner evaluation project. https://code.google.com/p/wavsep, July 2012.

[44] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proc. USENIX Security*, 2005.