Noname manuscript No.
(will be inserted by the editor)

Deception-Enhanced Threat Sensing for Resilient Intrusion Detection

Frederico Araujo · Gbadebo Ayoade · Kevin W. Hamlen · Latifur Khan

Received: date / Accepted: date

Abstract Enhancing standard web services with deceptive responses to cyber attacks can be a powerful and practical strategy for improved intrusion detection. Such deceptions are particularly helpful for addressing and overcoming barriers to effective machine learning-based intrusion detection encountered in many practical deployments. For example, they can provide a rich source of training data when training data is scarce, they avoid imposing a labeling burden on operators in the context of (semi-)supervised learning, they can be deployed post-decryption on encrypted data streams, and they learn concept differences between honeypot attacks and attacks against genuine assets.

The approach presented in this chapter examines how deceptive web service responses can be realized as software security patches that double as feature extraction engines for a network-level intrusion detection system. The resulting system coordinates multiple levels of the software stack to achieve fast, automatic, and accurate labeling of live web data streams, and thereby detects attacks with higher accuracy and adaptability than comparable non-deceptive defenses.

1 Introduction

Detecting previously unseen cyber attacks before they reach unpatched, vulnerable web servers (or afterward, for recovery purposes) is an increasingly central component to multi-layered defense of modern computer networks.

The research presented in this chapter was supported in part by AFOSR award FA9550-14-1-0173, NSF #1513704, NSA award H98230-15-1-0271, and ONR award N00014-17-1-2995.

F. Araujo
IBM T.J. Watson Research
E-mail: frederico.araujo@ibm.com

G. Ayoade, K.W. Hamlen, and L. Khan
The University of Texas at Dallas
E-mail: {gbadebo.ayoade,hamlen,lkhan}@utdallas.edu

High-impact zero-day vulnerabilities now appear at a weekly or daily rate, and studies indicate that over 75% of web sites have unpatched vulnerabilities [19]. The cost of data breaches resulting from software exploits was estimated at $2.1 trillion for 2019 [14].

Intrusion detection [10] is an important means of mitigating such threats. Rather than implement vulnerability-specific mitigations (which is difficult when the vulnerability is unknown to defenders), intrusion detection systems more generally alert administrators when they detect deviations from a model of *normal* behavior in the observed data [20]. This capitalizes on the observation that the most damaging and pernicious attacks discovered in the wild often share similar traits, such as the steps intruders take to open back doors, execute files and commands, alter system configurations, and transmit gathered information from compromised machines. Starting with the initial infection, such malicious activities often leave telltale traces that can be identified even when the underlying exploited vulnerabilities are unknown to defenders. The challenge is therefore to capture and filter these attack trails from network traffic, connected devices, and target applications, and develop defense mechanisms that can effectively leverage such data to disrupt ongoing attacks and prevent future attempted exploits.

However, despite its great power, the deployment of machine learning approaches for web intrusion detection is often hindered by a scarcity of realistic, current cyber attack data with which to train the system, and by the difficulty of accurately and efficiently labeling such data sets, which are often prohibitively large and complex. This can frustrate comprehensive, timely training of intrusion detection systems (IDSes), casuing the IDS to raise numerous false alarms and elevating its susceptibility to attacker evasion techniques [6, 9, 13, 16, 18].

To mitigate these dilemmas, this chapter presents a deception-based approach to enhancing IDS web data streams for faster, more accurate, and more timely evolution of intrusion detection models to emerging attacks and attacker strategies.

2 Deceptive Collection of Attack Data

Deception has long been recognized as a key ingredient of effective cyber warfare (cf., [23]), but many realizations limit the potential power of deception by isolating and separating deceptive assets from the data stream in which intrusions must actually be detected. A typical example is the use of dedicated *honeypots* to collect attack-only data streams [21]. Such approaches unfortunately have limited training value in that they often mistrain IDSes to recognize only attacks against honeypots, or only attacks by unsophisticated adversaries unable to identify and avoid honeypots. For example, attacks that include substantial interactivity are typically missed, since the honeypot offers no legitimate services, and therefore collects no data characterizing attacks against legitimate services.

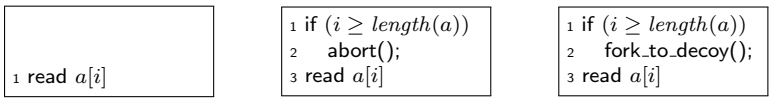


Fig. 1 Pseudo-code for a buffer overflow vulnerability (left), a patch (middle), and a honey-patch (right)

One way to overcome this limitation is to integrate deceptive attack response capabilities directly into live, production web server software via *honey-patching* [2,3,4]. Honey-patches are software security patches that are modified to avoid alerting adversaries when their exploit attempts fail. Instead of merely blocking the attempted intrusion, the honey-patch transparently redirects the attacker's connection to a carefully isolated decoy environment running an unpatched version of the software. Adversaries attempting to exploit a honey-patched vulnerability therefore observe software responses that resemble unpatched software, even though the vulnerability is actually patched. This deception allows the system to observe subsequent actions by the attacker until the deception is eventually uncovered. Thus, honey-patches offer equivalent security to conventional patches, but can also enhance IDS web data streams by feeding them a semantically rich stream of pre-labeled (attack-only) data for training purposes. These *deception-enhanced* data streams thus provide IDSes with concept-relevant, current, feature-filled information with which to detect and prevent sophisticated, targeted attacks.

Honey-patches are often easy to implement via only a minor change to a vendor-released software patch. For example, buffer overflow vulnerabilities are typically patched by adding a bounds check that tests whether a dereferenced pointer or array index falls within the bounds of the buffer. Such patches can easily be reformulated into honey-patches by retaining the check, but changing what happens when the check fails. Instead of aborting the connection or reporting an error, the honey-patch redirects the connection to an unpatched decoy, where the buffer overflow is permitted to succeed.

Figure 1 demonstrates the approach using pseudo-code for a buffer-overflow vulnerability, a conventional patch, and a honey-patch. The honey-patch retains the logic of the conventional patch's security check, but replaces its remediation with a deceptive fork to a decoy environment. The decoy contains no valuable data and offers no legitimate services; its sole purpose is to monitor attacker actions, such as shellcode or malware introduced by the attacker after abusing the buffer overflow to hijack the software. The infrastructure for redirecting attacker connections to decoys can remain relatively static, so that honey-patching each newly discovered vulnerability only entails replacing the few lines of code in each patch that respond to detected exploits.

Honey-patches constitute an integrated deception mechanism that offers some important advantages over conventional honeypots. Most significantly, they observe attacks against the defender's genuine assets, not merely those directed at fake assets that offer no legitimate services. They can therefore capture data from sophisticated attackers who monitor network traffic to identify service-providing assets before launching attacks, who customize their

attacks to the particular activities of targeted victims (differentiating genuine servers from dedicated honeypots), and who may have already successfully infiltrated the victim's network before their attacks become detected. The remainder of this chapter examines how the deception-enhanced data harvested by honey-patches can be of particular value to network-level defenses, such as firewalls equipped with machine learning-based intrusion detection.

3 Intrusion Detection Challenges

Despite the potential power of machine learning in intrusion detection applications, its success in operational environments can be hampered by specific challenges that arise in the cyber security domain. In this section we argue that cyber deception can be a highly effective strategy for avoiding or overcoming many of these challenges.

Fundamentally, machine learning algorithms perform better at identifying similarities than at discovering previously unseen outliers. Since normal, non-attack data is usually far more plentiful than realistic, current attack data, many classifiers must be trained almost solely from the former, necessitating an almost perfect model of normality for any reliable classification [18]. Deceptive defenses help to offset this imbalance by providing a continuous source of realistic attack data specialized to the defender's network and assets.

Feature extraction [7] is also unusually difficult in intrusion detection contexts because security-relevant features are often not known by defenders in advance. The task of selecting appropriate features to detect an intrusion (e.g., features that generate the most distinguishing intrusion patterns) can create a bottleneck in building effective models, since it demands empirical evaluation. Identification of attack traces among collected workload traces for constructing realistic, unbiased training sets is particularly challenging. Current approaches usually require manual analysis aided by expert knowledge [6, 9], which reduces the model's evolutionary and update capabilities, making it susceptible to attacker evasions. The approach presented in this chapter shows how including deceptions within software security patches can overcome this difficulty.

A third obstacle is analysis of encrypted data. Encryption is widely employed to prevent unauthorized users from accessing sensitive web data transmitted through network links or stored in file systems. However, since network-level detectors typically discard cyphered data, their efficacy is greatly reduced by the widespread use of encryption technologies [13]. In particular, attackers benefit from encrypting their malicious payloads, making it harder for standard classification strategies to distinguish attacks from normal activity. Deceptive defenses can often be placed after decryption within the software stack, evading this problem.

High false positive rates are another practical challenge for adoption of machine learning approaches [16]. Raising too many alarms renders intrusion detection meaningless in most cases, as actual attacks are often lost among the many alarms. Studies have shown that effective intrusion detection therefore demands very low false alarm rates [5]. Deception-enhanced data streams can

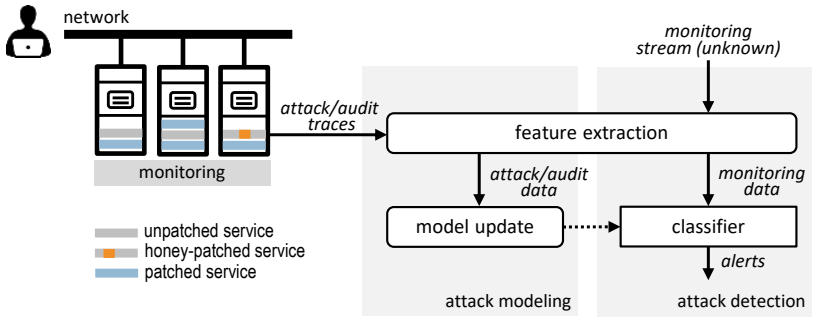


Fig. 2 System architecture overview

ameliorate this by improving the concept-relevance of the collected training data, improving attack detection accuracy.

4 Mining Deception-Enhanced Threat Data

To mitigate these challenges, this chapter introduces an approach to enhance intrusion detection with threat data sourced from honey-patched [4] applications. Figure 2 shows an overview of the approach. Unlike conventional approaches, our framework incrementally builds a model of *legitimate* and *malicious* behavior based on audit streams and attack traces collected from honey-patched web servers. This augments the classifier with security-relevant feature extraction capabilities not available to typical network intrusion detectors, effectively reducing the anomaly detection task to a *semi-supervised* learning process.

Such capabilities are transparently built into the framework, requiring no additional developer effort (apart from routine patching) to convert the target application into a potent feature extractor for anomaly detection. Since traces extracted from decoys are always contexts of *true* malicious activity, this results in an effortless labeling of the data and supports the generation of higher-accuracy detection models.

Honey-patches add a layer of deception to confound exploits of known (patchable) vulnerabilities. Previously unknown (i.e., zero-day) exploits can also be mitigated through IDS cooperation with the honey-patches. For example, a honey-patch that collects identifying information about a particular adversary seeking to exploit a known vulnerability can convey that collected information to train a classifier, which can then potentially identify the same adversary seeking to exploit a previously unknown vulnerability. This enables training intrusion detection models that *capture features of the attack payload*, and not just features of the actual exploitation of the vulnerability, thus more closely approximating the true invariant of an attack.

To facilitate such learning, our approach classifies *sessions* as malicious, not merely the individual packets, commands, or bytes within sessions that comprise each attack. For example, observing a two-phase attack consisting of (1) exploitation of a honey-patched vulnerability, followed by (2) injection

of previously unseen shellcode might train a model to recognize the shellcode. Subsequent attacks that exploit an unpatched zero-day to inject the same (or similar) shellcode can then be recognized by the classifier even if the zero-day exploit is not immediately recognized as malicious. Conventional, non-deceptive patches often miss such learning opportunities by terminating the initial attack at the point of exploit, before the shellcode can be observed.

Our approach therefore essentially repurposes security patches in an IDS setting as automated, application-level feature extractors. The maintenance burden for these extractors is relatively low: most of the patch code is maintained by the collective expertise of the entire software development community, as they discover new vulnerabilities and release patches for them. Via honey-patching, defenders can reimagine those patches as highly accurate, rapidly co-evolving feature extraction modules for an IDS. The extractor detects previously unseen payloads that exploit known vulnerabilities at the application layer, which can be prohibitively difficult to detect by a strictly network-level IDS.

By living inside web servers that offer legitimate services, a deception-enhanced IDS can target attackers who use one payload for reconnaissance but reserve another for their final attacks. The facility of honey-patches to deceive such attackers into divulging the latter is useful for training the IDS to identify the final attack payload, which can reveal attacker strategies and goals not discernible from the reconnaissance payload alone. The defender's ability to thwart these and future attacks therefore derives from a synergy between the application-level feature extractor and the network-level intrusion detector to derive a more complete model of attacker behavior.

5 Use Case: Booby-trapping Software for Intrusion Detection

5.1 Architectural Overview

The architecture depicted in Figure 2 embodies this approach by leveraging application-level threat data gathered from attacker sessions misdirected to decoys. Within this framework, developers use honey-patches to misdirect attackers to decoys that automatically collect and label monitored attack data. The intrusion detector consists of an *attack modeling* component that incrementally updates the anomaly model data generated by honey-patched servers, and an *attack detection* component that uses this model to flag anomalous activities in the monitored perimeter.

The decoys into which attacker sessions are forked can be managed as a pool of continuously monitored containers (e.g., LXC on Linux). Each container follows the following life cycle: Upon attack detection, the honey-patching mechanism *acquires* the first available container from the pool. The acquired container holds an attacker session until (1) the session is deliberately closed by the attacker, (2) the connection's *keep-alive* timeout expires, (3) the ephemeral container crashes, or (4) a session timeout is reached. The last two conditions are common outcomes of successful exploits. In any of these cases, the container

is released back to the pool and undergoes a recycling process before becoming available again.

After decoy *release*, the *container monitoring component* extracts the session trace (delimited by the acquire and release timestamps), labels it, and stores the trace outside the decoy for subsequent feature extraction. Decoys only host attack sessions, so precisely collecting and labeling their traces (at both the network and OS level) is effortless.

Evaluating the framework requires distinguishing three separate input data streams: (1) the *audit stream*, collected at the target honey-patched server, (2) *attack traces*, collected at decoys, and (3) a *monitoring stream*, which consists of a actual test stream collected from regular servers. Each of these streams contains network packets and operating system events captured at each server environment. To minimize performance impact, a powerful and highly efficient software monitor is recommended. Recommended candidates include *sysdig* (to track system calls and modifications made to the file system), and *tcpdump* (to monitor ingress and egress of network packets). Specifically, monitored data is stored outside the decoy environments to avoid possible tampering with the collected data.

Using the continuous audit stream and incoming attack traces as labeled input data, the intrusion detector incrementally builds a machine learning model that captures legitimate and malicious behavior. The raw training set (composed of both audit stream and attack traces) is piped into a feature extraction component that selects relevant, non-redundant features (see §5.2) and outputs feature vectors—*audit data* and *attack data*—that are grouped and queued for subsequent model update. Since the initial data streams are labeled and have been preprocessed, feature extraction becomes very efficient and can be performed automatically. This process repeats periodically according to an administrator-specified policy. Finally, the *attack detection* module uses the most recently constructed attack model to detect malicious activity in the run-time *monitoring data*.

5.2 Detection Models

To assess our framework's ability to enhance intrusion detection data streams, we have designed and implemented two feature set models: (1) *Bi-Di* detects anomalies in security-relevant network streams, and (2) *N-Gram* finds anomalies in system call traces. The feature set models and classifier presented in this section serve as illustrative use case. Applications of the IDS framework should consider other machine learning models and contrast tradeoffs and their effectiveness for attack detection.

5.2.1 Network Packet Analysis

Bi-Di (Bi-Directional) is a packet-level network behavior analysis approach that extracts features from sequences of packets and *bursts*—consecutive packets oriented to the same direction (*viz.*, uplinks from client to server, or downlinks

Table 1 Packet, uni-burst, and bi-burst features

Category	Features
Packet (Tx/Rx)	Packet length
Uni-Burst (Tx/Rx)	Uni-Burst size Uni-Burst time Uni-Burst count
Bi-Burst (Tx-Rx/Rx-Tx)	Bi-Burst size Bi-Burst time

from server to client). It uses distributions from individual burst sequences (*uni-bursts*) and sequences of two adjacent bursts (*bi-bursts*). To be robust against encrypted payloads, we limit feature extraction to packet headers.

Network packets flow between client (*Tx*) and server (*Rx*). Bi-Di constructs histograms using features extracted from packet lengths and directions. To overcome dimensionality issues associated with burst sizes, *bucketization* is applied to group bursts into correlation sets (e.g., based on frequency of occurrence). Table 1 summarizes the features used in our approach. It highlights new features proposed for uni- and bi-bursts as well as features proposed in prior works [1, 12, 15, 22].

Uni-burst features include burst *size*, *time*, and *count*—i.e., the sum of the sizes of all packets in the burst, the amount of time for the entire burst to be transmitted, and the number of packets it contains, respectively. Taking direction into consideration, one histogram for each is generated.

Bi-burst features include time and size attributes of *Tx-Rx-bursts* and *Rx-Tx-bursts*. Each is comprised of a consecutive pair of downlink and uplink bursts. The size and time of each are the sum of the sizes of the constituent bursts, and the sum of the times of the constituent bursts, respectively.

Bi-bursts capture dependencies between consecutive packet flows in a TCP connection. Based on connection characteristics, such as network congestion, the TCP protocol applies flow control mechanisms (e.g., window size and scaling, acknowledgement, sequence numbers) to ensure a level of consistency between Tx and Rx. This influences the size and time of transmitted packets in each direction. Each packet flow (uplink and downlink) thereby affects the next flow or burst until communicating parties finalize the connection.

5.2.2 System Call Analysis

The monitored data also includes system streams comprising a collection of OS events, where each event contains multiple fields including event type (e.g., *open*, *read*, *select*), process name, and direction. Our prototype implementation was developed for Linux x86_64 systems, which exhibit about 314 distinct possible system call events. Our framework builds histograms from these system calls using N-Gram—a system-level approach that extracts features from contiguous sequences of system calls.

Algorithm 1: *Ens-SVM*

Data: training data: $TrainX$, testing data: $TestX$
Result: a predicted label $\mathcal{L}_{\mathcal{I}}$ for each testing instance $\mathcal{I}$

1 **begin**
2 $\quad\mathbb{B} \leftarrow updateModel(\text{Bi-Di}, TrainX)$;
3 $\quad\mathbb{N} \leftarrow updateModel(\text{N-Gram}, TrainX)$;
4 $\quad$**for** *each* $\mathcal{I} \in TestX$ **do**
5 $\quad\quad\mathcal{L}_{\mathbb{B}} \leftarrow label(\mathbb{B}, \mathcal{I})$;
6 $\quad\quad\mathcal{L}_{\mathbb{N}} \leftarrow label(\mathbb{N}, \mathcal{I})$;
7 $\quad\quad$**if** $\mathcal{L}_{\mathbb{B}} == \mathcal{L}_{\mathbb{N}}$ **then**
8 $\quad\quad\quad\mathcal{L}_{\mathcal{I}} \leftarrow \mathcal{L}_{\mathbb{B}}$;
9 $\quad\quad$**else**
10 $\quad\quad\quad\mathcal{L}_{\mathcal{I}} \leftarrow label\left(\underset{c \in \{\mathbb{B},\mathbb{N}\}}{\arg\max}\, confidence(c, \mathcal{I}),\ \mathcal{I}\right)$;
11 $\quad\quad$**end**
12 $\quad$**end**
13 **end**

There are four feature types: *Uni-events* are system calls, and can be classified as enter or exit events. *Bi-events* are sequences of two consecutive events, where system calls in each bi-event constitute features. Similarly, *tri-* and *quad-events* are sequences of three and four consecutive events (respectively).

Bi-Di and N-Gram differ in feature granularity; the former uses coarser-grained bursting while the latter uses only individual system call co-occurrences.

5.3 Attack Classification

Bi-Di and N-Gram both use SVM for classification. Using a convex optimization approach and mapping non-linearly separated data to a higher dimensional linearly separated feature space, SVM separates positive (attack) and negative (benign) training instances by a hyperplane with the maximum gap possible. Prediction labels are assigned based on which side of the hyperplane each monitoring/testing instance belongs.

Ens-SVM: Bi-Di and N-Gram can be combined to obtain a better predictive model. A naïve approach concatenates features extracted by Bi-Di and N-Gram into a single feature vector and uses it as input to the classification algorithm. However, this approach has the drawback of introducing normalization issues. Alternatively, *ensemble methods* combine multiple classifiers to obtain a better classification outcome via majority voting techniques. For our purposes, we use an ensemble, *Ens-SVM*, which classifies new input data by weighing the classification outcomes of Bi-Di and N-Gram based on their individual accuracy indexes.

Algorithm 1 describes the voting approach for Ens-SVM. For each instance in the monitoring stream, if both Bi-Di and N-Gram agree on the predictive label (line 7), Ens-SVM takes the common classification as output (line 8). Otherwise, if the classifiers disagree, Ens-SVM takes the prediction with the highest SVM confidence (line 10). Confidence is rated using Platt scaling [17],

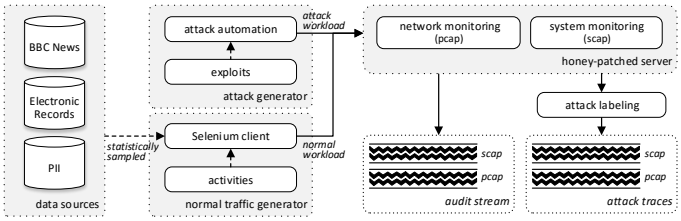


Fig. 3 Web traffic generation and testing harness

which uses the following sigmoid-like function to compute the classification confidence:

$$P(y = 1|x) = \frac{1}{1 + \exp(Af(x) + B)} \tag{1}$$

where y is the label, x is the testing vector, $f(x)$ is the SVM output, and A and B are scalar parameters learned using Maximum Likelihood Estimation (MLE). This yields a probability measure of how much a classifier is confident about assigning a label to a testing point.

6 Evaluation Testbed

Objective, scientific evaluation of cyber-deceptions is often very difficult, because evaluations on live attackers tend to be subjective (there is usually no way to know whether an anonymous attacker was genuinely deceived or just "playing along"), anecdotal (samples of hundreds or thousands of provably distinct attackers are required to draw quantifiable conclusions), and impossible to replicate. Much of the prior work in this space has been criticized on those grounds. Our work therefore offers a more rigorous evaluation methodology, which demonstrates that objectively quantifiable success metrics for IDSes significantly improve when exposed to deception-enhanced data, and the experimental results are reliably reproducible at large sample sizes.

6.1 Realistic Web Traffic Generation

To demonstrate the practical advantages and feasibility of deception-enhanced intrusion detection, we built a web traffic generator and test harness. Figure 3 shows an overview of our evaluation testbed, inspired by prior work [8]. It streams realistic *encrypted* legitimate and malicious workloads onto a honey-patched web server, resulting in labeled audit streams and attack traces (collected at decoys) for training set generation.

Legitimate data. Normal traffic is created by automating complex user actions on a typical web application as shown in table 3, leveraging *Selenium* to automate user interaction with a web browser (e.g., clicking buttons, filling out forms, navigating a web page). We generated web traffic for 12 different user

Table 2 Summary of synthetic data generation.

Normal workload summary		
Activity	**Application**	**Description**
Post	CGI web app	Posting blog on a guestbook CGI web application
Post	Wordpress	Posting blog on wordpress
Post	Wordpress buddypress plugin	Posting comment on social media web application
Registration	Wordpress woocommerce plugin	Product registration and product description
Ecommerce	Wordpress woocommerce plugin	Ordering of a product and checkout
Browse	Wordpress	Browsing through a blog post
Browse	Wordpress buddypress	Browsing through a social media page
Browse	Wordpress woocommerce plugin	Browsing product catalog
Registration	Wordpress	User registration
Registration	Wordpress woocommerce plugin	Coupon registration

activities (each repeated 200 times), including web page browsing, e-commerce website navigation, blog posting, and interacting with a social media web application. The setup included a CGI web application and a PHP-based Wordpress application hosted on a monitored Apache web server. To enrich the set of user activities, the Wordpress application was extended with *Buddypress* and *Woocommerce* plugins for social media and e-commerce web activities, respectively.

To create realistic interactions with the web applications, our framework feeds from online data sources, such as the BBC text corpus, online text generators for personally identifiable information (e.g., usernames, passwords), and product names to populate web forms. To ensure diversity, we statistically sampled the data sources to obtain user input values and dynamically generated web content. For example, blog title and body is statistically sampled from the BBC text corpus, while product names are picked from the product names data source.

Attack data. Attack traffic is generated based on real vulnerabilities as shown in Table 3. For this evaluation, we selected 16 exploits for eight well-advertised, high-severity vulnerabilities. These include CVE-2014-0160 (Heartbleed), CVE-2014-6271 (Shellshock), CVE-2012-1823 (improper handling of query strings by PHP in CGI mode), CVE-2011-3368 (improper URL validation), CVE-2014-0224 (Change Cipher specification attack), CVE-2010-0740 (Malformed TLS record), CVE-2010-1452 (Apache mod_cache vulnerabilty), and CVE-2016-7054 (Buffer overflow in openssl with support for ChaCha20-Poly1305 cipher suite). In addition, nine attack variants exploiting CVE-2014-6271 (Shellshock) were created to carry out different malicious activities (i.e., different attack payloads), such as leaking password files and invoking bash shells on the remote web server. These vulnerabilities are important as attack vectors because they range from sensitive data exfiltration to complete control and remote code execution. To emulate realistic attack traffic, we interleaved attacks and normal traffic following the strategy of Wind Tunnel [8].

Dataset. The traffic generator is deployed on a separate host to avoid interference with the testbed server. To account for operational and environmental differences, our framework simulates different workload profiles (according to time of day), against various target configurations (including different back-

Table 3 Summary of attack workload

#	Attack Type	Description	Software
1	CVE-2014-0160	Information leak	Openssl
2	CVE-2012-1823	System remote hijack	PHP
3	CVE-2011-3368	Port scanning	Apache
4–10	CVE-2014-6271	System hijack (7 variants)	Bash
11	CVE-2014-6271	Remote Password file read	Bash
12	CVE-2014-6271	Remote root directory read	Bash
13	CVE-2014-0224	Session hijack and information leak	Openssl
14	CVE-2010-0740	DoS via NULL pointer dereference	Openssl
15	CVE-2010-1452	DoS via request that lacks a path	Apache
16	CVE-2016-7054	DoS via heap buffer overflow	Openssl

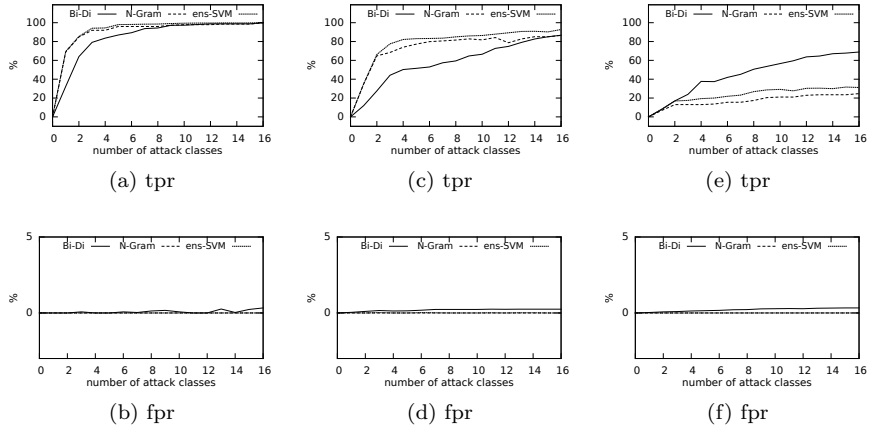
ground processes and server workloads), and network settings, such as TCP congestion controls. In total, we generated 12 GB of (uncompressed) network packets and system events over a period of three weeks. After feature extraction, the training data comprised 1200 normal instances and 1600 attack instances. Monitoring or testing data consisted of 2800 normal and attack instances gathered at unpatched web servers, where the distribution of normal and attack instances varies per experiment.

6.2 Experimental Results

Using this dataset, we trained the classifiers presented in §5.2 and assessed their individual performance against test streams containing both normal and attack workloads. In the experiments, we measured the true positive rate (*tpr*) where true positive represents the number of actual attack instances that are classified as attacks, false positive rate (*fpr*) where false positive represents the number of actual benign instances classified as attacks, accuracy (*acc*), and F_2 score of the classifier, where the F_2 score is interpreted as the weighted average of the precision and recall, reaching its best value at 1 and worst at 0. An RBF kernel with $Cost = 1.3 \times 10^5$ and $\gamma = 1.9 \times 10^{-6}$ was used for SVM [15].

Detection accuracy. To evaluate the accuracy of intrusion detection, we tested each classifier after incrementally training it with increasing numbers of attack classes. Each class consists of 100 distinct variants of a single exploit, as described in §6.1, and an n-class model is one trained with up to n attack classes. For example, a 3-class model is trained with 300 instances from 3 different attack classes. In each run, the classifier is trained with 1200 normal instances and $100 * n$ attack instances where $n \in [1, 16]$ attack classes. In addition, in each run, we execute ten experiments where the attacks are shuffled in a cross-validation-like fashion and the average is reported. This ensures training is not biased toward any specific attacks.

Testing on decoy data. The first experiment measures the accuracy of each classifier against a test set composed of 1200 normal instances and 1600 uniformly distributed attack instances gathered at decoys. Figure 4(a)–(b) presents the results, which serve as a preliminary check that the classifiers



(a) tpr (c) tpr (e) tpr

(b) fpr (d) fpr (f) fpr

Fig. 4 Classification accuracy of Bi-Di, N-Gram, and Ens-SVM for 0–16 attack classes for (a)–(b) training and testing on decoy data, (c)–(d) training on decoy data and testing on unpatched server data, and (e)–(f) training on regular-patched server data and testing on unpatched server data.

can accurately detect attack instances resembling the ones comprised in their initial training set.

Testing on unpatched server data. The second experiment also measures each classifier's accuracy, but this time the test set was derived from monitoring streams collected at regular, *unpatched* servers, and having a uniform distribution of attacks. Figure 4(c)–(d) shows the results, which indicate that the detection models of each classifier generalize beyond data collected in decoys. This is critical because it demonstrates the classifier's ability to detect previously unseen attack variants. Our framework thus enables administrators to add an additional level of protection to their entire network, including hosts that cannot be promptly patched, via the adoption of a honey-patching methodology.

The results also show that as the number of training attack classes increases—which are proportional to the number of vulnerabilities honey-patched—a steep improvement in the true positive rate of both classifiers is observed, reaching an average *tpr* of above 92% for the compounded Ens-SVM, while average false positive rate in all experiments remained below 0.01%. This demonstrates the positive impact of the *feature-enhancing* capabilities of deceptive application-level attack responses like honey-patching.

Training on regular-patched server data. To compare our approach against analogous, standard IDSes that do not employ deception, we trained each classifier on data collected from non-deceptive, regular-patched servers, and tested them on the unpatched server data, using the same set of attacks. Figure 4(e)–(f) shows the results, which outline the inherent challenges of traditional intrusion detection models on obfuscated, unlabeled attack traces. Unlike honey-patches, which capture and label traces containing patterns of

Table 4 Base detection rates for approximate targeted attack scenario ($P_A \approx 1\%$) [11]

Classifier	***tpr***	***fpr***	***acc***	F_2	***bdr***
OneSVM-Bi-Di	55.56	13.17	68.96	59.69	4.09
OneSVM-N-Gram	84.77	0.52	91.07	87.09	62.22
Bi-Di	86.69	0.25	92.29	89.02	77.79
N-Gram	86.52	0.01	92.30	88.89	98.98
Ens-SVM	**92.76**	**0.01**	**95.86**	**94.12**	**99.05**

successful attacks, conventional security patches yield traces of failed attack attempts, making them unfit to reveal patterns of attacks against unpatched systems.

Baseline evaluation. This experiment compares the accuracy of our detection approach to the accuracy of an unsupervised outlier detection strategy, which is commonly employed in typical intrusion detection scenarios [9], where labeling attack data is not feasible or prohibitively expensive. For this purpose, we implemented two *One-class SVM* classifiers, *OneSVM-Bi-Di* with a polynomial kernel and $\nu = 0.1$ and *OneSVM-N-Gram* with a linear kernel and $\nu = 0.001$, using Bi-Di and N-Gram models for feature extraction, respectively. We fine tuned the One-class SVM parameters and performed a systematic grid search for the kernel and ν to get the best results.

One-class SVM uses an unsupervised approach, where the classifier trains on one class and predicts whether a test instance belongs to that class, thereby detecting *outliers*—test instances outside the class. To perform this experiment, we incrementally trained each classifier with an increasing number of *normal* instances, and tested the classifiers after each iteration against the same unpatched server test set used in the previous experiments. The results presented in Table 4 highlight critical limitations of conventional outlier intrusion detection systems: reduced predictive power, lower tolerance to noise in the training set, and higher false positive rates.

In contrast, our supervised approach overcomes such disadvantages by automatically streaming onto the classifiers labeled security-relevant features, without any human intervention. This is possible because honey-patches identify security-relevant events at the point where such events are created, and not as a separate, *post-mortem* manual analysis of traces.

6.3 Discussion

Methodology. Our experiments show that just a few strategically chosen honey-patched vulnerabilities accompanied by an equally small number of honey-patched applications provide a machine learning-based IDS sufficient data to perform substantially more accurate intrusion detection, thereby enhancing the security of the entire network. Thus, we arrive at one of the first demonstrable measures of value for deception in the context of cyber security: its utility for enhancing IDS data streams.

Supervised learning. Our approach facilitates supervised learning, whose widespread use in the domain of intrusion detection has been impeded by many challenges involving the manual labeling of attacks and the extraction of security-relevant features [9]. Our results demonstrate that the language-based, active response capabilities provided via application-level honey-patches significantly ameliorates both of these challenges. The facility of deception for improving other machine learning-based security systems should therefore be investigated.

Intrusion detection datasets. One of the major challenges in evaluating intrusion detection systems is the dearth of publicly available datasets, which is often aggravated by privacy and intellectual property considerations. To mitigate this problem, security researchers often resort to synthetic dataset generation, which affords the opportunity to design test sets that validate a wide range of requirements. Nonetheless, a well-recognized challenge in custom dataset generation is how to capture the multitude of variations and features manifested in real-world scenarios [6]. Our evaluation approach builds on recent breakthroughs in dataset generation for IDS evaluation [8] to create statistically representative workloads that resemble realistic web traffic, thereby affording the ability to perform a meaningful evaluation of IDS frameworks.

7 Conclusion

This chapter outlined the implementation and evaluation of a new approach for enhancing web intrusion detection systems with threat data sourced from deceptive, application-layer, software traps. Unlike conventional machine learning-based detection approaches, our framework incrementally builds models of legitimate and malicious behavior based on audit streams and traces collected from these traps. This augments the IDS with inexpensive and automatic security-relevant feature extraction capabilities. These capabilities require no additional developer effort apart from routine patching activities. This results in an effortless labeling of the data and supports a new generation of higher-accuracy detection models.

8 Exercises

8.1 Software Engineering Exercises

∗ 1. Give an example of a high-profile software exploit cyberattack whose impact has been reported recently in the news, and for which the cyber-deceptive software techniques described in this chapter might have proved beneficial, if deployed. Based on any technical details available, advise how such a defense might have helped in that scenario, and discuss potential implementation issues or risks involved.

Table 5 Confusion Matrix

Total No of Instances: 160		Actual Classes	
		Attack	Benign
Predicted Classes	Attack	20	30
	Benign	10	100

** 2. For each of the following vulnerability types, find an example patch for one such vulnerability (e.g., from MITRE CWE), and then write code that reformulates the patch into a honey-patch. In your honey-patch, use the function call *fork_to_decoy()* to indicate where your code would fork the attacker's connection to a decoy environment. Remember, a good honey-patch implementation should not impact legitimate users!
   (a) buffer underflow/overflow (overwrite, overread, underwrite, or underread)
   (b) C format string vulnerability
   (c) TOCTOU (time-of-check / time-of-use) vulnerability
   (d) SQL injection
   (e) XSS (cross-site scripting)

*** 3. Install older (non-fully patched) versions of OpenSSL and Apache, and identify from a CVE list some of the unpatched vulnerabilities. Implement a honey-patch for any of the CVEs. Invite classmates to operate as a red team to penetrate your server. Were they able to distinguish the decoy environment from any successful compromise? Would any data collected from detected attacks potentially help your server resist subsequent exploit attempts?

8.2 Machine Learning Exercises

* 1. Given a set of data traces with packet data, what type of features can be extracted from packets?

* 2. Similarly, given a set of data traces with system calls, what type of features can be extracted to train a machine learning classifier?

* 3. Given the confusion matrix in Table 5, and defining *positives* to be alarms raised by the defense, calculate the following metrics: Accuracy, FPR, and TPR.

** 4. Why is false positive rate (FPR) important in evaluating machine learning based intrusion detection systems?

** 5. Implement an IDS using support vector machine that leverages packet data traces to classify and detect attack in collected data traces. For this exercise, you can follow the following steps:
   – *Extract packet Information:* Use the `dpkt` python toolkit to extract packet information, such as length, count, packet direction, and packet time.
   – Build a histogram of the packet length for each trace. Each trace will generate an instance to train your classifier.

   - After generating your data set, use the `Scklearn` python machine learning module to build an SVM classifier.

*** 6. Implement an ensemble classifier using support vector machine to leverage both packet data and system call data to classify attack traces. You can follow the steps described in previous question to complete this exercise.

** 7. Calculate the following metrics with the classifier you implemented in exercises 5 and 6: Accuracy and FPR. How do you explain the significance of the FPR compared to the accuracy?

*** 8. Run your algorithm on data collected from a honey-patched system (see software engineering exercises 2–3) and compare the performance to the data collected on a system with no honey-patch.

*** 9. Based on software engineering exercise 3, implement your own data collection mechanism that captures packet and system call level data. Apply your machine learning implementation from exercise 5 on the data traces collected. Compare your performance with the supplied data. To complete this exercise, you can use tcpdump (already installed on Linux systems) to collect packet trace data and sysdig[1] to collect system call data. To reduce noise in your data collection, run each attack independently and collect the associated traces. Remember to run each attack and trace collection multiple times to account for variations in system operation.

References

1. K. Alnaami, G. Ayoade, A. Siddiqui, N. Ruozzi, L. Khan, and B. Thuraisingham. P2V: Effective website fingerprinting using vector space representations. In *Proceedings of the IEEE Symposium on Computational Intelligence*, pages 59–66, 2015.
2. F. Araujo and K. W. Hamlen. Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception. In *Proceedings of the USENIX Security Symposium*, 2015.
3. F. Araujo and K. W. Hamlen. Embedded honeypotting. In S. Jajodia, V. Subrahmanian, V. Swarup, and C. Wang, editors, *Cyber Deception: Building the Scientific Foundation*, chapter 10, pages 195–225. Springer, 2016.
4. F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 942–953, 2014.
5. S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1–7, 1999.
6. M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita. Network anomaly detection: Methods, systems and tools. *IEEE Communications Surveys & Tutorials*, 16(1):303–336, 2014.
7. A. L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1):245–271, 1997.
8. N. Boggs, H. Zhao, S. Du, and S. J. Stolfo. Synthetic data generation and defense in depth measurement of web applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 234–254, 2014.
9. V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):15, 2009.

[1] https://sysdig.com/opensource/sysdig/install

10. D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.
11. D. Dudorov, D. Stupples, and M. Newby. Probability analysis of cyber attack paths against business and commercial enterprise systems. In *Proceedings of the IEEE European Intelligence and Security Informatics Conference*, pages 38–44, 2013.
12. K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 332–346, 2012.
13. P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1):18–28, 2009.
14. Juniper Research. The future of cybercrime and security: Financial and corporate threats and mitigation, 2015.
15. A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the Annual ACM Workshop on Privacy in the Electronic Society*, pages 103–114, 2011.
16. A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.
17. J. C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pages 61–74. MIT Press, 1999.
18. R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 305–316, 2010.
19. Symantec. Internet security threat report, vol. 21, 2016.
20. C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin. Intrusion detection by machine learning: A review. *Expert Systems with Applications*, 36(10):11994–12000, 2009.
21. E. Vasilomanolakis, S. Karuppayah, M. Mühlhäuser, and M. Fischer. Taxonomy and survey of collaborative intrusion detection. *ACM Computing Surveys*, 47(4), 2015.
22. T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the USENIX Security Symposium*, 2014.
23. J. Yuill, D. Denning, and F. Feer. Using deception to hide things from hackers: Processes, principles, and techniques. *Journal of Information Warfare*, 5(3):26–40, 2006.