Crook-sourced Intrusion Detection as a Service

Frederico Araujo[a,*], Gbadebo Ayoade[b], Khaled Al-Naami[b], Yang Gao[b], Kevin W. Hamlen[b] and Latifur Khan[b]

[a]IBM Research, Thomas J. Watson Research Center, Yorktown Heights, NY, 10598, USA
[b]The University of Texas at Dallas, Richardson, TX, 75080, USA

ARTICLE INFO

Keywords:
intrusion detection, datasets, neural networks, honeypots, cyberdeception, cloud computing, software-as-a-service

ABSTRACT

Most conventional cyber defenses strive to reject detected attacks as quickly and decisively as possible; however, this instinctive approach has the disadvantage of depriving intrusion detection systems (IDSes) of learning experiences and threat data that might otherwise be gleaned from deeper interactions with adversaries. For IDS technology to improve, a next-generation cyber defense is proposed in which cyber attacks are unconventionally reimagined as free sources of live IDS training data. Rather than aborting attacks against legitimate services, adversarial interactions are selectively prolonged to maximize the defender's harvest of useful threat intelligence. Enhancing web services with deceptive attack-responses in this way is shown to be a powerful and practical strategy for improved detection, addressing several perennial challenges for machine learning-based IDS in the literature, including scarcity of training data, the high labeling burden for (semi-)supervised learning, encryption opacity, and concept differences between honeypot attacks and those against genuine services. By reconceptualizing software security patches as feature extraction engines, the approach conscripts attackers as free penetration testers, and coordinates multiple levels of the software stack to achieve fast, automatic, and accurate labeling of live web streams.

Prototype implementations are showcased for two feature set models to extract security-relevant network- and system-level features from cloud services hosting enterprise-grade web applications. The evaluation demonstrates that the extracted data can be fed back into a network-level IDS for exceptionally accurate, yet lightweight attack detection.

1. Introduction

Cyber attackers breach computer networks using a myriad of techniques, with web application vulnerabilities corresponding to 25% of all exploitable attack vectors [51]. Detecting cyber attacks before they reach unpatched, vulnerable web servers (or afterward, for recovery purposes) has become a vital necessity for many organizations. In 2018 alone, the average window of exposure for critical web application vulnerabilities was 69 days, with a new vulnerability found *every hour*—an increase of 13% over the previous year's rate—and over 75% of all legitimate web sites have unpatched vulnerabilities, 20% of which afford attackers full control over victim systems [110, 51]. The cost of data breaches resulting from software exploits is expected to escalate to an unprecedented $2.5 trillion by 2022 [66].

Intrusion detection [47] is an important means of mitigating such threats, since it offers a means of automatically analyzing large, continuous data streams in which a relatively small number of threats may be concealed. IDSes capitalize 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 [103, 48, 63, 94]. 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. Specifically, machine learning-based IDSes alert administrators when deviations from a model of *normal* behavior are detected [53, 79, 123].

However, despite its great promise, the advancement of machine learning approaches for web intrusion detection have been hindered by a scarcity of realistic, current, publicly available cyber attack data sets, and by the difficulty of accurately and efficiently labeling such data sets, which are often prohibitively large and complex [115]. This has frustrated comprehensive, timely training of IDSes, and has resulted in an overreliance on unrealistic closed-world assumptions [107], thereby raising IDS false alarm rates and elevating their susceptibility to attacker evasion [17, 26, 55, 96, 107]. The general inadequacy of static attack datasets also introduces severe impediments to machine learning-based IDS deployment. Models trained with labeled data from a specific domain doesn't usually transfer, or generalize, to other domains. For example, data streams obtained from cloud-based Linux services cannot be used to predict cyber attacks against enterprise Windows endpoints, due to the intrinsic differences between the operating environments. This limitation impairs IDS model evolution and the adaptation of machine learning defenses against new and emergent attack techniques.

This paper proposes and examines a new deception-based approach to enhancing IDS data streams through *crook-sourcing*—the conscription and manipulation of attackers into performing free penetration testing for improved IDS model training and adaptation [7]. Unlike conventional intrusion detection, this deception-enhanced IDS incrementally builds attack detection models based on attacker behaviors collected from successful deceptions. The deceptions leverage user interactions at the network, endpoint, or application layers to solicit extra communication with adversaries and automatically label attack data. This augments the classifier with security-relevant feature extraction capabilities not available to typical network intrusion detectors.

*Corresponding author: frederico.araujo@ibm.com
ORCID(s): 0000-0001-5143-8318 (F. Araujo)

Deception has long been recognized as a key ingredient of effective cyber warfare (cf., [124]), but its applications to IDS have heretofore been limited to contexts where the deception is isolated and separate from the data stream in which intrusions must actually be detected. For example, dedicated *honeypots* collect attack-only data streams [116] but have limited IDS training value in that they can mistrain models to recognize only attacks against honeypots, including false positives from scans and accidental connections, or attacks by unsophisticated adversaries unable to identify and avoid honeypots. Attacks with substantial interactivity can be missed, since the honeypot offers no legitimate services, and therefore collects no data characterizing attacks against legitimate services.

Our approach overcomes this limitation by integrating deceptive attack response capabilities directly into live, production server software via *honey-patching* [9]. 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 observe software responses that resemble unpatched software, even though the vulnerability is actually patched. This allows the system to observe subsequent attack actions until the deception is eventually uncovered. Honey-patches offer equivalent security to conventional patches, but can potentially enhance IDS web data streams with a semantically rich stream of pre-labeled (attack-only) data for training purposes. These crook-sourced data streams thus provide IDSes with concept-relevant, current, feature-filled information with which to detect and prevent sophisticated, targeted attacks.

To enable the creation and maintenance of public and on-premise cloud environments for crook-sourcing, we propose container technologies as a foundation for practical honey-patching in service-oriented architectures. Our monitoring framework transparently collects network and system telemetry from application endpoints, and automates the extraction and labeling of crook-sourced data streams for timely IDS evolution. Such *deception-as-a-service* (ΔaaS) leverages the replication and virtualization capabilities of modern cloud computing architectures to create a "hall of mirrors" that attackers must navigate in order to distinguish valuable targets from traps.

We demonstrate the potential effectiveness of this new IDS approach through the design, implementation, and analysis of DEEPDIG (DEcEPtion DIGging), a framework for deception-enhanced web intrusion detection. Evaluation shows that extra information harvested through mini-deceptions (1) improves precision of anomaly-based IDSes by feeding back attack traces into the classifier, (2) provides feature-rich, multi-dimensional attack data for classification, and (3) can detect exploit variants previously unseen by defenders. Our goal is to assess whether successful deceptions are helpful for intrusion detection, and to what degree. Given the scarcity of good, current intrusion data sets and the costs of conducting large-scale empirical data collection, we believe that the approach's facility for generating richer, automatically-labeled, web attack data streams offers exceptional promise for future IDS research and deployments.

Our contributions can be summarized as follows:

- We propose a software patching methodology that facilitates semi-supervised learning for intrusion detection, in which deceptive security patches naturally modulate and automate the attack labeling and feature extraction.
- We present a feature-rich attack classification that more accurately characterizes malicious web activities.
- To harness training and test data, we present the design of a framework for the replay and generation of real web traffic, which statistically mutates and injects scripted attacks into the generated output streams.[1]
- We describe a service-oriented deployment model for transparently enabling and scaling crook-sourcing in public an on-premise cloud environments for attack data collection through honey-patching.
- We evaluate our approach on large-scale network and system events gathered through simulation and red team evaluations over a test bed built atop production web software deployed on a AWS EC2 cloud stack, including the Apache web server, OpenSSL, and PHP.

Section 2 outlines our approach and presents a system overview, followed by a more detailed architecture description in Section 3. Section 4 shows how our approach supports accurate characterization of attacks through decoy data. Implementation is summarized in Section 5, followed by evaluation methodology and results in Section 6. Finally, discussion and related work are presented in Sections 7 and 8 (respectively), and Section 9 concludes with outcomes and future directions.

2. Approach Overview

We first outline practical limitations of traditional machine learning techniques for intrusion detection, motivating our research. We then overview our approach for automatic attack labeling and feature extraction via honey-patching.

2.1. Intrusion Detection Challenges

Despite the increasing popularity of machine learning in intrusion detection applications, its success in operational environments has been hampered by specific challenges that arise in the cyber security domain. 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 [107].

Feature extraction [18] 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) often creates 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

[1] The implementation and datasets used in this paper are available in https://github.com/cyberdeception/deepdig.

challenging. Current approaches usually require manual analysis aided by expert knowledge [26, 17], which severely reduces model evolution and update capabilities to cope with attacker evasion strategies.

A third obstacle is analysis of encrypted streams, which are ubiquitously employed to prevent unauthorized users from accessing sensitive web data transmitted through network links or stored in file systems. Since network-level detectors typically discard cyphered data, their efficacy is greatly reduced by the widespread use of encryption [55]. In particular, attackers benefit from encrypting their malicious payloads, making it harder for standard classification strategies to distinguish attacks from normal activity.

High false positive rates are another practical challenge for adoption of machine learning approaches [96]. Raising too many alarms renders IDSes meaningless in most cases, since actual attacks are lost among the many alarms. Studies have shown that effective intrusion detection therefore demands very low false alarm rates [12].

These significant challenges call for the exploration and development of new, accurate anomaly detection schemes that lift together information from many different layers of the software stack. Toward this end, our work extends machine learning-based intrusion detection with the capability to effectively detect malicious activities bound to the application layer, affording detection approaches an inexpensive tool for automatically and continuously extracting security-relevant features for attack detection.

2.2. Cyberdeceptive Defenses

Cyber deception has become increasingly important for protecting organizational and national critical infrastructures from asymmetric cyber threats. Market forecasts predict an industry in excess of $2 billion for cyberdeceptive products by 2022 [39], including major product releases by Rapid7, TrapX, LogRhythm, Attivo, Illusive Networks, Cymmetria, Thinkst Canary, and many others in recent years [102].

These new defense layers are rising in importance because they enhance conventional defenses by shifting asymmetries that traditionally burden defenders back on attackers. For example, while conventional defenses invite adversaries to find just one critical vulnerability to successfully penetrate the network, deceptive defenses challenge adversaries to discern which vulnerabilities among a sea of apparent vulnerabilities (many of them traps) are real. As attacker-defender asymmetries increase with the increasing complexity of networks and software, deceptive strategies for leveling those asymmetries will become increasingly essential for scalable defense.

2.3. Digging Deception-Enhanced Threat Data

DEEPDIG is a new approach to enhance intrusion detection with threat data sourced from honey-patched [9] applications. Figure 1 shows an overview of the approach. Unlike conventional techniques, DEEPDIG incrementally builds a model M of *legitimate* and *malicious* behavior based on audit streams and attack traces A collected from successful deceptions, and uses this continuously updated model to detect attacks observed in the monitoring stream T. The deceptions leverage user interactions at the network, endpoint, or application layers to solicit

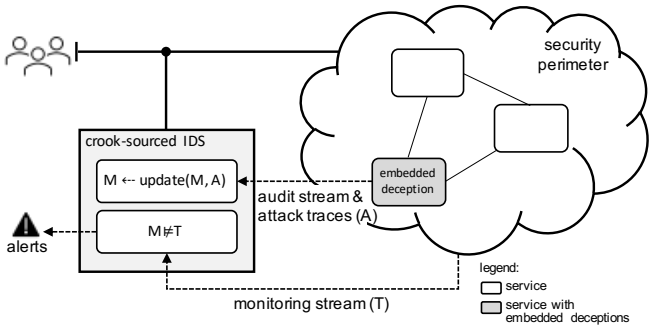


Figure 1: Crook-sourcing approach overview.

extra communication with adversaries and waste their resources, misdirect them, and gather intelligence. This augments the classifier with security-relevant feature extraction capabilities not available to typical network intrusion detectors. Specifically, *honey-patches* [9, 10, 34] introduce application layer deceptions by selectively replacing software security patches with decoy vulnerabilities. Attempted exploits transparently redirect the attacker's session to a decoy environment where the exploit is allowed to succeed. This allows the system to observe subsequent phases of the attacker's killchain without risk to genuine assets. Decoy environments can host genuinely unpatched software to make their responses precisely match adversarial expectations of vulnerable, compromised systems.

Our central enabling insight is that *software security patches can be repurposed as feature extractors for semi-supervised learning*. The maintenance of the feature extractors is crowd-sourced (by the software development community's ongoing discovery and creation of new security patches), and the data analyzed by the patches is crook-sourced (as attackers contribute their TTP patterns to the data streams processed by the embedded deceptions). Honey-patching transduces these two data sources into a highly accurate, rapidly co-evolving feature extraction module for an IDS. The extractor can effortlessly detect previously unseen payloads that exploit known vulnerabilities at the application layer, which can be prohibitively difficult to detect by a network-level IDS.

These 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. Traces extracted from decoys are always contexts of *true* malicious activity, yielding an effortless labeling of the data and higher-accuracy detection models.

By living inside web servers that offer legitimate services, our deception-enhanced IDS can target attackers who use one payload for reconnaissance but reserve another for their final attacks. Deceiving 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.

For example, consider a skilled adversary who knows that asset A is not a honeypot (e.g., because traffic analysis reveals it is delivering real services to real end-users), and attempts to exploit a known vulnerability V to inject beaconing malware M. If V is patched, the attack is rejected and the adversary

```
                                  1 if (i ≥ length(a))
                                  2     abort();
1 read a[i]                       3 read a[i]
```

```
1 if (i ≥ length(a))
2     fork_to_decoy();
3 read a[i]
```

Figure 2: Pseudo-code for a buffer overflow vulnerability (left), a patch (middle), and a honey-patch (right).

continues probing, eventually finding an unpatched vulnerability V' and exploiting it to upload M followed by a more destructive, previously unseen malware variant M'. However, if V is honey-patched then the attack appears to succeed, so the adversary exploits V to upload M', revealing M' to defenders before it is used in a successful attack. The defender's ability to thwart 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.

2.4. Honey-patching Approach

Prior work has observed that many vendor-released software security patches can be honeyed by replacing their attack-rejection responses with code that instead maintains and forks the attacker's connection to a confined, unpatched decoy [9, 10]. This approach retains the most complex part of the vendor patch (the security check) and replaces the remediation code with some boilerplate forking code [8], making it easy to implement.

Figure 2 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; its 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.

This integrated deception offers some important advantages over conventional honeypots. Most significantly, it observes attacks against the defender's genuine assets, not merely those directed at fake assets that offer no legitimate services; and it can observe attacks that are transparent to higher service layers, such as the system API, VM/OS, and network [58]. It 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 are detected. We next examine how deception-enhanced data harvested in this way can be of particular value to network-level defenses, such as firewalls armed with machine learning-based intrusion detection.

2.5. Deception as a Service

Cloud computing has attracted significant attention in recent years as a model for scalable service consumption and as a delivery platform for service-oriented computing. Revolutionary advances in hardware and virtualization technologies have elevated cloud computing to a thriving industry that affords enterprises the ability to shrink IT expenditures, adapt quickly to variable workloads, and reduce administration overhead. These successes have been achieved principally by reinventing a wide variety of traditionally on-site computing resources as deliverable services according to the mantra *Everything as a Service* (XaaS) [14]. Pillars of the XaaS mantra typically include Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

Our central observation is that the technological advances at the heart of the cloud computing movement have now converged to cultivate a remarkably fertile ground for mass-scale cyber deception as a defense. In particular, many foundational cloud technologies, including massive replication, high performance process migration and load balancing, hardware and software heterogeneity, aggressive multitenancy, and multi-layer virtualization, have led to computing environments ideal for assembling a "hall of mirrors" in which legitimate services are interlaced with deceptive computations, platforms, data, and software, all designed to misdirect attackers away from valuable targets. We refer to this vision as *Deception as a Service* (ΔaaS).

ΔaaS leverages rapid advances in cloud computing capabilities by introducing new, deception-powered defenses that leverage facilities and opportunities unique to cloud environments, and that cannot be realized as effectively on traditional, non-distributed computing platforms. This counterpoints public fears about the data security of cloud computing systems (cf., [22]) by championing clouds as a new cyber security *opportunity*, rather than merely a security resistant environment to which traditional defenses are transitioned. Our approach enables the automatic deployment and scaling of DEEPDIG for crook-sourcing conceptually-relevant threat data on hybrid cloud architectures and environments, affording cyber-defenders a new form of active response to attacks in commodity cloud and service-oriented infrastructures. It is therefore envisioned as a complement (not a replacement) to cloud defenses for computation integrity [69, 104, 31], data security and privacy [112, 20, 93, 68, 97], and non-deceptive cloud IDS (e.g., [70, 71]).

3. Architecture

DEEPDIG's architecture, depicted in Figure 3a, leverages application-level threat data gathered from attacker sessions redirected to decoys to train and adapt a network-level IDS live. Within this framework, honey-patches 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.

3.1. Monitoring & Threat Data Collection

The decoys into which attacker sessions are forked are managed as a pool of continuously monitored Linux containers. Each container follows the life cycle depicted in Fig. 3b. Upon attack detection, the honey-patching mechanism *acquires* the first available container from the pool. The acquired container holds an

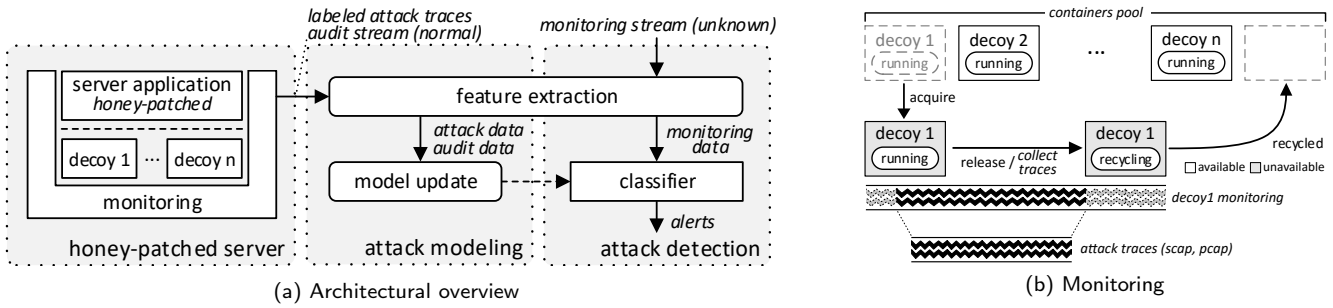


Figure 3: Overview of (a) deceptive IDS training and (b) decoy lifecycle management.

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 acquire and release), labels it, and stores it 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.

DEEPDIG distinguishes between three input data streams: (1) the *audit stream*, collected at the target honey-patched server; (2) *attack traces*, collected at decoys; and (3) the *monitoring stream*, the actual test stream collected from regular servers. Each of these streams contains network packets and OS events captured at each server environment. To minimize performance impact, we used two powerful and highly efficient software monitors: *sysdig* [111] (to track system calls and modifications made to the file system), and *libpcap* [114] (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.

Our monitoring and data collection solution is designed to scale for large, distributed on-premise and cloud deployments. The host-level telemetry leverages a mainstream kernel module that implements non-blocking event collection and memory-mapped event buffer handling for minimal computational overhead. This architecture allows system events to be safely collected (without system call interposition) and compressed by a containerized user space agent that is oblivious to other objects and resources located in the host environment. The event data streams originated from the monitored hosts are conveniently exported to a high-performance, distributed S3-compatible object storage server [91], designed for large-scale data infrastructures.

3.2. Attack Modeling & Detection

Using the continuous audit stream and incoming attack traces as labeled input data, DEEPDIG incrementally builds a machine learning model that captures legitimate and malicious behavior. The incremental updates accommodate the evolving TTPs of attackers and defenders as attack surfaces and adversarial experiences change over time. The raw training set (*viz.* the audit stream and attack traces) is piped into a feature extraction component that selects relevant, non-redundant features (see §4) 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 runtime *monitoring data*.

3.3. Deployment Automation

Modern computing environments typically require the configuration and orchestration of multiple services for applications to function. These can range from a few instances (e.g., a web server and a database), to very complex setups such as IaaS deployments requiring many components to be installed, configured, and interconnected (e.g., OpenStack). To ease the task of creating and maintaining such service-oriented environments, configuration management tools like Chef [29], Ansible [6], and Puppet [100], or even general-purpose scripting languages such as Python or Bash, automate the configuration of machines to a particular specification.

More recently, Juju [25] has been introduced as a model specification for service oriented architectures and deployments, enabling transparent and efficient management of cloud services on both public cloud infrastructures (e.g., Amazon EC2, Microsoft Azure, Joyent Triton) and private infrastructures (e.g., OpenStack, physical servers, containers). Juju abstracts and simplifies cloud deployment and scaling, and provides users with client-side command-line tools to uniformly manage locally and remotely deployed services. Application-specific knowledge such as dependencies, operational events like backups and upgrades, and integration options with other pieces of software are encapsulated in Juju's *charms*. A charm defines everything required to deploy a particular service, and is composed of user-implemented *hooks* which Juju invokes at different stages of the service's lifecycle.

A Service-Oriented Architecture for Crook-Sourcing. Using Juju as underlying framework, we implemented a charm that automates the deployment and scaling of DEEPDIG on top of IaaS environments, therefore augmenting cloud infrastructures with ΔaaS capabilities through honey-patching. Figure 4 shows

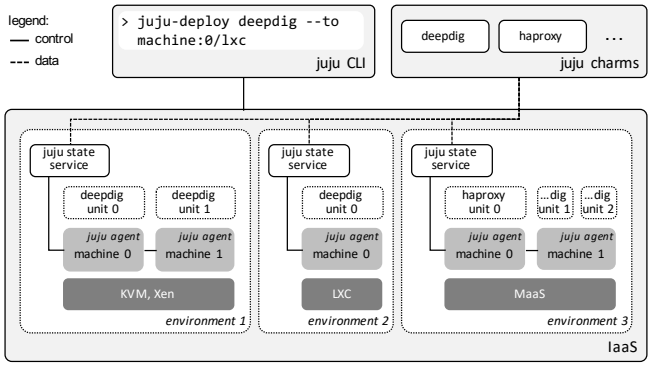


Figure 4: Overview of a ΔaaS architecture.

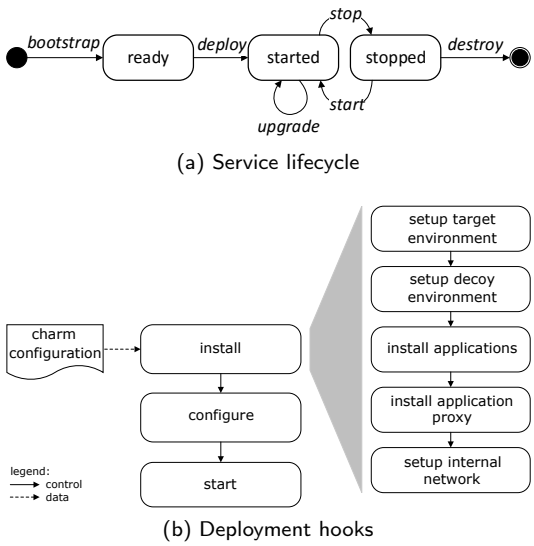


Figure 5: Overview of (a) service lifecycle and (b) deployment hooks and configuration.

an overview of the ΔaaS architecture, which outlines its main components and services. It builds atop of mainstream IaaS technologies, leveraging Juju to provision and orchestrate different ΔaaS deployment modes (e.g., load balanced, containerized, high-availability). Deployment packages (charms) are maintained and sourced from a centralized repository, and a remote command line interpreter is used to manage the deployments.

Users of our platform can easily deploy our crook-sourcing framework on a variety of environments including KVM, Xen, and LXC. Physical deployment is also supported through bare-metal containers and metal-as-a-service, which lets physical servers be treated like virtual machines in the cloud. For example, the command line instruction

```
juju-deploy deepdig –to machine:0/lxc
```

instructs the *juju state service* component to deploy a new unit of DEEPDIG on *environment 2*. This triggers the automatic instantiation of a new container (i.e., *machine 0*), and the *juju agent* running on the container is tasked with the execution of the charm specification.

Scalability. One of the main benefits of this service-oriented architecture is the simplicity of scaling services up and down. For example, to scale DEEPDIG up horizontally, users first instruct juju to add the desired number of units to the existent deployment (e.g., `juju add-unit deepdig –to machine:1/maas`), and then setup load balancing to distribute the work load among units. To achieve this, one option is to use the infrastructure's built-in load balancing capabilities. An alternative option is to deploy a load balancing service such as *HAProxy*:

```
juju-deploy haproxy –to machine:0/maas;
juju add-relation deepdig haproxy
```

Conversely, scaling down follows a similar procedure to remove deployed units.

Service Modeling. Each DEEPDIG service instance goes through a series of events during its lifecycle: *install*, *configure*, *start*, *upgrade*, and *stop*. Figure 5a depicts these events and the associated state transitions. Two special events, *bootstrap* and *destroy*, result in pre-defined actions executed by Juju, and correspond to the creation and destruction of the deployment environment, respectively. For each of the remaining events, Juju executes specific hooks specified in the deepdig charm. Hooks are executable scripts in a charm's hooks directory, and are invoked by the unit's juju agent at particular times in the service lifecycle. We designed deepdig hooks to be *idempotent*, meaning that there is no observable difference between running a hook once and running it multiple times.

Figure 5b details hooks associated with DEEPDIG's deployment. Hooks *setup target environment* and *setup decoy environment* pre-installs onto the deployment environment the target and decoy containers file systems according to the charm's configuration parameters (e.g., string prefix for decoy names, container pool size). Hook *install applications* fetches all applications specified in the configuration (e.g., a honey-patched Apache HTTP) and installs them into the target container. Finally, hooks *install reverse proxy* and *setup internal network* install the proxy on the unit and isolate the target container from the pool of decoys as a separate subnet, respectively.

4. Attack Detection

To assess our framework's ability to enhance IDS data streams, we have designed and implemented two familiar feature set models: (1) *Bi-Di* detects anomalies in security-relevant network streams, and (2) *N-Gram* finds anomalies in system call traces. Our approach is agnostic to the particular feature set model chosen; we choose these two models for evaluation purposes because they are simple and afford direct comparisons to non-deceptive prior works. The goal of the evaluation is hence to measure the utility of the deception for enhancing data streams for intrusion detection, not to assess the utility of novel feature sets.

4.1. Network Packet Analysis

Bi-Di (Bi-Directional) extracts features from sequences of packets and *bursts*—consecutive same-direction packets (*viz.*, uplinks from client *Tx*, or downlinks from server *Rx*) for network

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

behavior analysis. It uses distributions from individual burst sequences (*uni-bursts*) and sequences of two adjacent bursts (*bi-bursts*), constructing 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, including features from prior works [3, 50, 95, 119]. For robustness against encrypted payloads, we here limit feature extraction to packet headers.

Uni-burst features include burst *size* (the sum of the sizes of all packets in the burst), *time* (the duration for the entire burst to be transmitted), and *count* (the number of packets in the burst). 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 and times of the constituent bursts, respectively.

Bi-bursts capture dependencies between consecutive TCP packet flows. 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.

4.2. System Call Analysis

Monitored data also includes system streams comprised of OS events, each containing multiple fields, including event type (e.g., *open*, *read*, *select*), process name, and direction. Our prototype was developed for Linux x86_64 systems, which exhibit about 314 distinct system call events. We build histograms from these using N-Gram, which extracts features from event subsequences. Each feature type consists of between 1 (*uni-events*) and 4 (*quad-events*) consecutive events, with each event classified as an enter or exit.

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

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

4.3. Classification

We evaluate our approach's practicality using two supervised learning models: SVM [46] and deep learning [78]. Our main objective is to show that our deception-enhanced framework facilitates incremental supervised learning for intrusion detection.

Ens-SVM. This method builds SVM models for Bi-Di and N-Gram. Using convex optimization 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 resides.

We combine the two classifiers into an *ensemble* that classifies new input data by weighing the classification outcomes of Bi-Di and N-Gram based on their individual accuracy indexes. Ensemble methods tend to exhibit higher accuracy and avoid normalization issues raised by the alternative (brute force) approach of concatenating the dissimilar features into a single feature vector.

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 8), Ens-SVM takes the common classification as output (line 9). Otherwise, if the classifiers disagree, Ens-SVM takes the prediction with the highest SVM confidence (line 11). Confidence is rated using Platt scaling [98], which uses the following sigmoid-like function to estimate confidence:

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

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 a classifier's confidence in assigning a label to a testing point.

Metric Learning. To classify instances to classes, we use online adaptive metric learning (OAML) [54, 13]. OAML is better suited to our task than off-line approaches (e.g., k-nearest neighbors), which yield weak predictors when the separation between

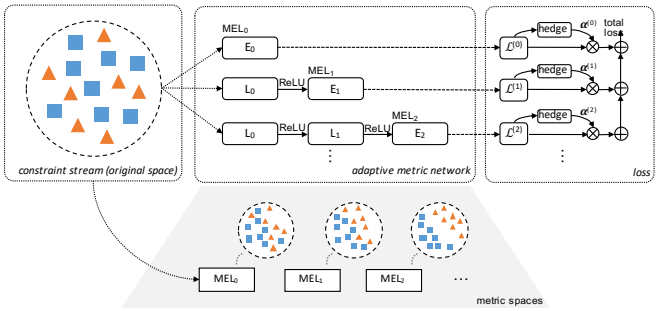


Figure 6: OAML network structure. Each layer L_i is a linear transformation output to a rectified linear unit (ReLU) activation. Embedding layers E_i connect to corresponding input or hidden layers. Linear model E_0 maps the input feature space to the embedding space.

different class instances is small. Online similarity metric learning (OML) [81, 28, 62, 64, 21] improves instance separation by finding a new latent space to project the original features, learning similarity from a stream of constraints. *Pairwise* and *triplet* constraints are typically employed: a pairwise constraint takes two dissimilar/similar instances, while a triplet constraint (A, B, C) combines similar instances A and B with a dissimilar instance C.

We choose adaptive OML since non-adaptive OML usually learns a pre-selected linear metric (e.g., Mahalanobis distance [122]) that lacks the complexity to learn non-linear semantic similarities among class instances, which are prevalent in intrusion detection scenarios. Moreover, it derives its metric model from well-defined input constraints, leading to bias towards the training data. OAML overcomes these disadvantages by adapting its complexity to accommodate more constraints in the observed data. Its metric function learns a dynamic latent space from the Bi-Di and N-Gram feature spaces, which can include both linear and highly non-linear functions.

OAML leverages artificial neural networks (ANNs) to learn a metric similarity function and can adapt its learning model based on the complexity of its input space. It modifies common deep learning architectures so that the output of every hidden layer flows to an independent metric-embedding layer (MEL). The MELs output an n-dimensional vector in an embedded space where similar instances are clustered and dissimilar instances are separated. Each MEL has an assigned metric weight to determine its importance for the models generated. The output of this embedding is used as input to a k-NN classifier. The approach is detailed below.

Problem Setting. Let $S = \{(\boldsymbol{x}_t, \boldsymbol{x}_t^+, \boldsymbol{x}_t^-)\}_{t=1}^T$ be a sequence of triplet constraints sampled from the data, where $\{\boldsymbol{x}_t, \boldsymbol{x}_t^+, \boldsymbol{x}_t^-\} \in \mathcal{R}^d$, and $\boldsymbol{x}_t$ (anchor) is similar to $\boldsymbol{x}_t^+$ (positive) but dissimilar to $\boldsymbol{x}_t^-$ (negative). The goal of online adaptive metric learning is to learn a model $\boldsymbol{F} : \mathcal{R}^d \mapsto \mathcal{R}^{d'}$ such that $||\boldsymbol{F}(\boldsymbol{x}_t) - \boldsymbol{F}(\boldsymbol{x}_t^+)||_2 \ll ||\boldsymbol{F}(\boldsymbol{x}_t) - \boldsymbol{F}(\boldsymbol{x}_t^-)||_2$. Given these parameters, the objective is to learn a metric model with adaptive complexity while satisfying the constraints. The complexity of $\boldsymbol{F}$ must be adaptive so that its hypothesis space is automatically modified.

Overview. Consider a neural network with L hidden layers, where the input layer and the hidden layer are connected to an independent MEL. Each embedding layer learns a latent space where similar instances are clustered and dissimilar instances are separated.

Figure 6 illustrates our ANN. Let $E_\ell \in \{E_0, E_1, \ldots, E_L\}$ denote the ℓ^{th} metric model in OAML (i.e., the network branch from the input layer to the ℓ^{th} MEL). The simplest OAML model E_0 represents a linear transformation from the input feature space to the metric embedding space. A weight $\alpha^{(\ell)} \in [0, 1]$ is assigned to E_ℓ, measuring its importance in OAML.

For a triplet constraint $(\boldsymbol{x}_t, \boldsymbol{x}_t^+, \boldsymbol{x}_t^-)$ that arrives at time t, its metric embedding $f^{(\ell)}(\boldsymbol{x}_t^*)$ generated by E_ℓ is

$$f^{(\ell)}(\boldsymbol{x}_t^*) = h^{(\ell)}\Theta^{(\ell)} \tag{2}$$

where $h^{(\ell)} = \sigma(W^{(\ell)}h^{(\ell-1)})$, with $\ell \geq 1$, $\ell \in \mathbb{N}$, and $h^{(0)} = \boldsymbol{x}_t^*$. Here $\boldsymbol{x}_t^*$ denotes any anchor ($\boldsymbol{x}_t$), positive ($\boldsymbol{x}_t^+$), or negative ($\boldsymbol{x}_t^-$) instance, and $h^{(\ell)}$ represents the activation of the ℓ^{th} hidden layer. Learned metric embedding $f^{(\ell)}(\boldsymbol{x}_t^*)$ is limited to a unit sphere (i.e., $||f^{(\ell)}(\boldsymbol{x}_t^*)||_2 = 1$) to reduce the search space and accelerate training.

For every triplet $(\boldsymbol{x}_t, \boldsymbol{x}_t^+, \boldsymbol{x}_t^-)$ arriving during the training phase, we first retrieve the metric embedding $f^{(\ell)}(\boldsymbol{x}_t^*)$ from the ℓ^{th} metric model using Eq. 2. A local loss $\mathcal{L}^{(\ell)}$ for E_ℓ is evaluated by calculating the similarity and dissimilarity errors based on $f^{(\ell)}(\boldsymbol{x}_t^*)$. Thus, the overall loss introduced by this triplet is given by

$$\mathcal{L}_{overall}(\boldsymbol{x}_t, \boldsymbol{x}_t^+, \boldsymbol{x}_t^-) = \sum_{\ell=0}^{L} \alpha^{(\ell)} \cdot \mathcal{L}^{(\ell)}(\boldsymbol{x}_t, \boldsymbol{x}_t^+, \boldsymbol{x}_t^-) \tag{3}$$

Parameters $\Theta^{(\ell)}$, $\alpha^{(\ell)}$, and $W^{(\ell)}$ are learned during the online learning phase. The final optimization problem to solve in OAML at time t is therefore:

$$\begin{aligned} \underset{\Theta^{(\ell)}, W^{(\ell)}, \alpha^{(\ell)}}{\text{minimize}} \quad & \mathcal{L}_{overall} \\ \text{subject to} \quad & ||f^{(\ell)}(\boldsymbol{x}_t^*)||_2 = 1, \forall \ell = 0, \ldots, L. \end{aligned} \tag{4}$$

We evaluate the similarity and dissimilarity errors using an *adaptive-bound triplet loss* (ABTL) constraint [54] to estimate $\mathcal{L}^{(\ell)}$ and update parameters $\Theta^{(\ell)}$, $W^{(\ell)}$ and $\alpha^{(\ell)}$.

Continual Learning. Novel classes may appear at any time in the monitoring streams (e.g., new attacks and new deceptions). To cope with such *concept-evolving* data streams, we include a novel class detector that extends traditional classifiers with automatic detection of novel classes before the true labels of the novel class instances arrive. Once a novel class is detected, the current batch of input data is used to incrementally retrain our supervised models. Such *continual learning* technique utilizes the production data streams to retrain the model based on the new activity, thus enabling DEEPDIG to continuously adapt to changes in the operating environment.

Data stream classification. Novel class detection observes that data points belonging to a common class are closer to each other (*cohesion*), yet far from data points belonging to other classes (*separation*). Building upon ECSMiner [88, 2], our approach segments data streams into equal, fixed-sized *chunks*, each containing a set of monitoring traces, efficiently buffering chunks

for online processing. When a buffer is examined for novel classes, the classification algorithm looks for strong cohesion among outliers in the buffer and large separation between outliers and training data. When strong cohesion and separation are found, the classifier declares a novel class.

Training & model update. A new classifier is trained on each chunk and added to a fixed-sized ensemble of M classifiers, leveraging audit and attack instances (traces). After each iteration, the set of $M + 1$ classifiers are ranked based on their prediction accuracies on the latest data chunk, and only the first M classifiers remain in the ensemble. The ensemble is continuously updated following this strategy and thus modulates the most recent concept in the incoming data stream, alleviating adaptability issues associated with concept drift [88]. Unlabeled instances are classified by majority vote of the ensemble's classifiers.

Classification model. Each classifier in the ensemble uses a k-NN classification, deriving its input features from Bi-Di and N-Gram feature set models. Rather than storing all data points of the training chunk in memory, which is prohibitively inefficient, we optimize space utilization and time performance by using a semi-supervised clustering technique based on Expectation Maximization (E-M) [89]. This minimizes both intra-cluster dispersion and cluster impurity, and caches a summary of each cluster (centroid and frequencies of data points belonging to each class), discarding the raw data points.

Feature transformation. To make the learned representations robust to partial corruption of the input patterns and improve classification accuracy, abstract features are generated from the original feature space during training via a *stacked denoising autoencoder* (DAE) [117, 118] using the instances of the first few chunks in the data stream. Stacked DAE builds a deep neural network that aims to capture the statistical dependencies between the inputs by reconstructing a clean input from a corrupted version of it, thus forcing the hidden layers to discover more robust features (yielding better generalization) and prevent the classifier from learning the identity (while preserving the information about the input). Figure 7 illustrates our approach. The first step creates a corrupted version $\widetilde{x}$ of input $x \in \mathbb{R}^d$ using *additive Gaussian noise* [30]. In other words, a random value v_k is added to each feature in x: $\widetilde{x}_k = x_k + v_k$ where $k = [1 \dots d]$ and $v_k \sim \mathcal{N}(0, \sigma^2)$ (cf., [16]). The output of the training phase is a set of weights W and bias vectors b. We keep the learned weights and biases to transform the feature values of the subsequent instances of the stream. Finally, the transformed features are then used to retrain the classifier.

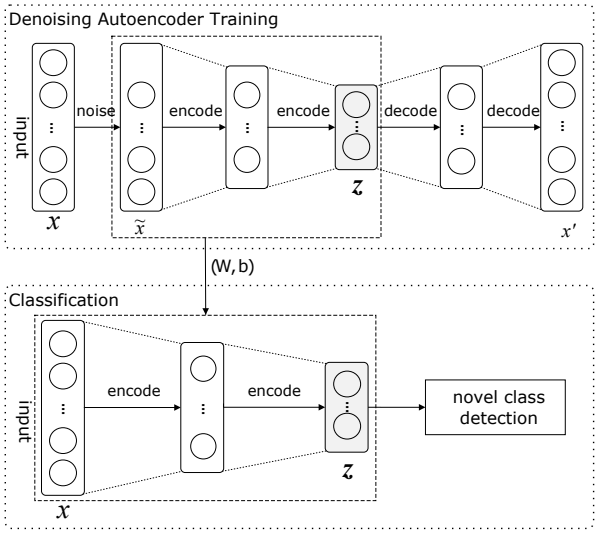


Figure 7: Overview of feature transformation.

5. Implementation

We developed an implementation of DEEPDIG for 64-bit Linux (kernel 3.19). It consists of two main components: (1) The monitoring controller performs server monitoring and attack trace extraction from decoys. It consists of about 350 lines of Node.js code, and leverages *tcpdump*, *editcap*, and *sysdig* for network and system call tracing and preprocessing. (2) The attack detection component is implemented as two Python modules: the feature extraction module, comprising about 1200 lines of code and feature generation; and the classifier component, comprising 230 lines of code that references the *Weka* [57] wrapper for LIBSVM [27]. The OAML components comprise about 500 lines of Python code referencing the PyTorch [101] library.

The source-code modifications required to honey-patch vulnerabilities in Apache HTTP, Bash, PHP, and OpenSSL consist of a mere 35 lines of C code added or changed in the original server code, showing that the required deceptive capabilities can be added to production-level web services with very little effort. (The forking framework [9, 8] is fixed, and thus not included in this count.)

To validate our ΔaaS architecture, we have implemented a Juju charm for DEEPDIG, and used it to deploy honey-patching as a service both on premise (on LXC containers running on top of a Linux VM) and remotely on Amazon EC2. To test our deployments, we streamed into each instance scripted attacks generated by our testing framework (see Section 6.1). Overall, our hooks consist of about 460 lines of Bash code, and expose a rich set of configuration parameters to ease deployment customization.

6. Evaluation

A central goal of our research is to quantitatively measure the impact of embedded deception on IDS accuracy. Our evaluation approach therefore differs from works that seek to measure absolute IDS accuracy, or that do not separate the impact of deception from the rest of the detection process. We first present our evaluation framework, which we harness to automatically generate training and test datasets from real web traffic for our experiments. Then we discuss our experimental setup and investigate the effects of different attack classes and varying numbers of attack instances on the predictive power and accuracy of the intrusion detection. Finally, we assess the performance impact of the deception monitoring mechanism that captures network packets and system events.

All experiments were performed on a 16-core hosts with 24 GB RAM running 64-bit Ubuntu 16.04. We used our ΔaaS automation to deploy regular and honey-patched servers as LXC containers [82] running atop the hosts using the official Ubuntu

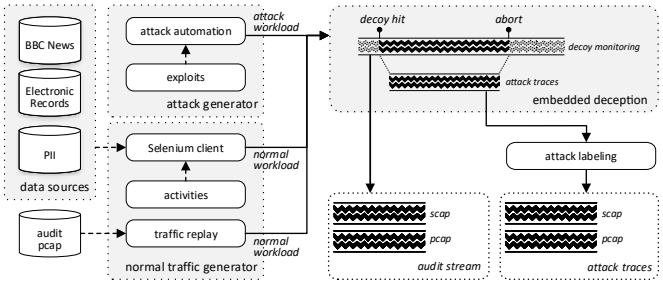


Figure 8: Overview of automated workload generation and testing harness.

container image. Red teaming validation was performed on a similar environment comprising of EC2 instances deployed on AWS (Amazon Web Services).

6.1. Experimental Framework

Figure 8 shows an overview of our evaluation framework, inspired by related work [19, 13]. It streams *encrypted* legitimate and malicious workloads (both simulated and real) simultaneously onto a honey-patched web server, resulting in labeled audit streams and attack traces (collected at decoys) for training set generation. This strategy facilitates the reproducibility of our experiments while allowing for the validation of our approach in a realistic setting. Table 2 summarizes the collected network traffic and systems events used in our experiments.

Legitimate workload. In order to collect normal data, we used both real user interactions with a web browser and automated simulation of various user actions on the browser. For the real user interaction, we monitored and recorded web traffic from users in a local area network (comprising 20 endpoints) over a two-day period, resulting in more than 30GB of *audit pcap* data. The recorded sessions are replayed by our framework and include users exhibiting normal browsing activities, such as accessing social media websites, search engines, online shopping websites, web email, video sharing, and news websites.

For the simulated interaction, normal traffic is created by automating complex user actions on a typical web application, leveraging *Selenium* [105] 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 activities (each repeated 200 times with varying data feeds) over a span of three weeks, 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 [56], online text generators [92] 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 are statistically sampled from the BBC text corpus, while product names are picked from the product names data source.

Table 2
Summary of collected network traffic and system events.

Type	Source	Description
Legitimate	Real users	30GB web traffic collected from a local network comprising 20 endpoints over a two-day period.
	Simulated	12GB web traffic and systems events generated using our automated testing framework for 12 different user activities types over a span of three weeks.
Attack	Red teaming	Web traffic and system events collected from a penetration testing exercise comprising 10 students with varied skill levels in offensive security perform an average of 45 minutes of penetration testing on their own time, over a span of three days.
	Simulated	Web traffic and system events generated from scripted remote attacks (cf. Table 3) and automatically interleaved to legitimate traffic (set at 0.5–1% of the overall traffic) using our testing framework.

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
17–22	CVE-2017-5941*	System hijack (6 variants)	Node.js

*used for testing only, as *n*-day vulnerability.

Attack workload. Attack traffic is generated based on real-world vulnerabilities, and corresponds to 0.5–1% of the overall network traffic, to approximate the results found in prior studies on targeted attacks [49]. Table 3 lists 22 exploits for nine 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), CVE2010-0740 (Malformed TLS record), CVE-2010-1452 (the Apache mod_cache vulnerabilty), CVE-2016-7054 (Buffer overflow in openssl with support for ChaCha20-Poly1305 cipher suite), and CVE-2017-5941 (Node.js

error handling vulnerability). 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, dropping malware, and invoking bash shells on the remote server. These vulnerabilities are important as attack vectors because they range from sensitive data exfiltration to complete control and remote code execution.

Similarly, we generated attack traffic using CVE-2017-5941 as an additional remote execution attack and executed 6 attack payloads. The post-infection payloads executed tasks such as tool acquisition from remote host, basic environment reconnaissance (e.g., active scanning with Nmap, passive inspection of system logs), remote password file access, root certificate exfiltration, and attempts at gaining access to other machines in the network.

Red teaming. To validate our results, we conducted a small-scale penetration testing experiment using a red team composed of graduate students in cyber security who were tasked with performing reconnaissance and attacking a collection of monitored web server instances deployed on AWS. The deployment comprised unpatched and honey-patched web servers exposing (real or apparent) Shellshock vulnerabilities. Special care was taken not to open unpatched software to the broad Internet. The attack team was composed by 10 students with basic (e.g., script-kidding) to advanced skills (e.g., penetration testing, reverse engineering) in offensive security. After a preliminary leveling and preparation, each student spent an average of 45 minutes completing the exercise on their own time. Each student were assigned separate, identical EC2 instances. Students were not permitted to repeat any exercise more than once. Network and system data was collected over a span of 3 days and used as testing and control data.

After successfully completing the initial infection, observed data show that students performed different malicious activities, such as searching for password files, printing user sessions on the system, listing network interfaces, injecting boot loaders, and installing root kits. For example, collected attack samples included modification of `rc.local` to launch a listening process in the background at boot time, network traffic redirection through iptables, file exfiltration over the network, file deletions and access permission modification, and cleanup of event logs and shell history to cover attack paths. No students were able to distinguish honey-patched from unpatched web servers during the experiment; they reported they had compromised genuine assets even when they had not.

Noise injection. Rather than testing with existing, publicly available intrusion datasets (which are inappropriate evaluations of DEEPDIG, since they lack concept-relevance for deception and are generally stripped of raw packet data), our evaluation interleaves attack and normal traffic following prior work on defense-in-depth [19], and injects benign payloads as data into attack packets to mimic evasive attack behavior. The generated traffic contains attack payloads against recent CVEs for which we created and tested realistic exploits, and our framework automatically extracts labeled features from the monitoring network and system traces to (re-)train the classifiers.

Dataset. Web traffic was generated from a separate host to avoid interference with the test bed server. To account for operational and environmental differences, our framework simulated different workload profiles (according to time of day), against various target configurations (including different background processes and server workloads), and network settings, such as TCP congestion controls. In total, we generated 42 GB of (uncompressed) network packets and system events over a period of three weeks. After feature extraction, the training data comprised 1800 normal instances and 1600 attack instances. Monitoring or testing data consisted of 3400 normal and attack instances gathered at unpatched web servers, where the distribution of normal and attack instances varies per experiment.

Detection accuracy. Using this dataset, we trained the classifiers presented in §4 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. We also calculated a base detection rate (*bdr*) to estimate the success of intrusion detection (§6.3). An RBF kernel with $Cost = 1.3 \times 10^5$ and $\gamma = 1.9 \times 10^{-6}$ was used for SVM [95]. OAML employed a ReLU network with n=200, L=1, and k=5 (defined in §4.3).

To evaluate the accuracy of intrusion detection, we verified 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 1800 normal instances and $100 * n$ attack instances with $n \in [1, 16]$ attack classes sourced from decoys. Each run executes 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.

6.2. Experimental Results

Table 4 measures the accuracy of classifiers that were trained using deceptive servers, and then tested on attacks against *unpatched* servers (to evaluate protection against patching lapses). Attacks are uniformly distributed across all synthetic attack classes and variants described in §6.1. Each result is compared (in parentheses) against the same experiment performed without deception. Leveraging deception yields an 8–22% increase in classification accuracy, with an 8–20% increase in true positives and a 5–41% reduction in false positives. Env-SVM achieves 97% accuracy with almost no false positives (0.01%).

These significant gains demonstrate that the detection models of each classifier learned from deception-enhanced data generalize beyond data collected in decoys. This showcases the classifier's ability to detect previously unseen attack variants. DEEPDIG 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.

Table 4
Detection rates (%) for scripted attack scenarios ($P_A \approx 1\%$) compared with results from non-deceptive training (parenthesized).

Classifier	*tpr*	*fpr*	*acc*	F_2	*bdr*
Bi-Di OML	91.00 (+13.2)	0.01 (-41.2)	91.14 (+22.2)	90.00 (+30.3)	98.92 (+97.1)
N-Gram OML	65.00 (-19.9)	0.01 (-5.1)	88.58 (+0.0)	80.00 (-8.4)	98.50 (+84.0)
Bi-Di SVM	79.00 (+1.2)	0.78 (-40.5)	89.88 (+20.9)	78.69 (+19.0)	50.57 (+36.1)
N-Gram SVM	92.42 (+7.5)	0.01 (-5.1)	96.89 (+8.3)	93.84 (+5.5)	99.05 (+84.6)
Ens-SVM	**93.63** (+8.8)	**0.01** (-5.1)	**97.00** (+8.4)	**94.89** (+6.5)	**99.06** (+84.6)

Table 5
Detection rates (%) for red team evaluation ($P_A \approx 1\%$) compared with results from non-deceptive training (parenthesized).

Classifier	*tpr*	*fpr*	*acc*	F_2	*bdr*
Bi-Di-OML	94.00 (-4.0)	0.39 (-52.6)	93.10 (+23.1)	94.00 (+7.0)	70.88 (+69.1)
Ngram-OML	99.00 (+1.0)	0.01 (-50.0)	99.90 (+26.9)	94.00 (+5.0)	99.01 (+97.1)
Bi-Di-SVM	99.56 (+1.6)	1.15 (-51.9)	99.19 (+29.2)	99.39 (+12.4)	46.65 (+44.8)
N-Gram-SVM	92.25 (-6.75)	0.01 (-50.0)	96.35 (+23.4)	93.70 (+4.7)	98.94 (+97.0)
Ens-SVM	**99.56** (+0.56)	**0.01** (-50.0)	**99.19** (+26.2)	**99.39** (+10.4)	**99.02** (+97.1)

Figure 9a shows that as the number of training attack classes (which are proportional to the number of vulnerabilities honey-patched) increases, a steep improvement in the true positive rate is observed, reaching an average above 93% for Ens-SVM, while average false positive rate in all experiments remains low ($< 1\%$). This demonstrates that deception has a *feature-enhancing* effect—the IDS learns from the prolonged adversarial interactions to detect more attacks.

Testing on an "unknown" vulnerability. We also measured our approach's ability to detect a *previously unseen*, unpatched remote code execution exploit (CVE-2017-5941) carrying attack payloads (classes 17–22) resembling the payloads that have been used to exploit honey-patched vulnerabilities (CVE-2014-6271). In this experiment, CVE-2017-5941 is used as an *n*-day vulnerability for which no patch has been applied. The resulting 98.6–99.8% *tpr* and 0.01–0.67% *fpr* show that crook-sourcing helps the classifier learn attack patterns unavailable at initial deployment, but revealed by deceived adversaries during decoy interactions, to learn exploits for which the classifier was not pre-trained.

Red teaming validation. Table 5 summarizes detection accuracy against the red team. We incrementally train our previously trained model with new attack instances collected from live decoys, and use it to detect human attacks against unpatched servers. The accuracy rates are much higher against human opponents than against the synthetic attacks, indicating that our synthetic data constitutes a challenging test. This may be in part because replicating the high diversity of the synthetic attacks would require an extremely large-scale human study.

False alarms. Figure 9b plots the false positive rates for classifiers that have undergone 30 incremental training iterations, each with 1–30 normal/attack instances per class. With just a few attack instances (≈ 5 per attack class), the false positive rates drop to almost zero, demonstrating that DEEPDIG's continuous feeding back of attack samples into classifiers greatly reduces false alarms.

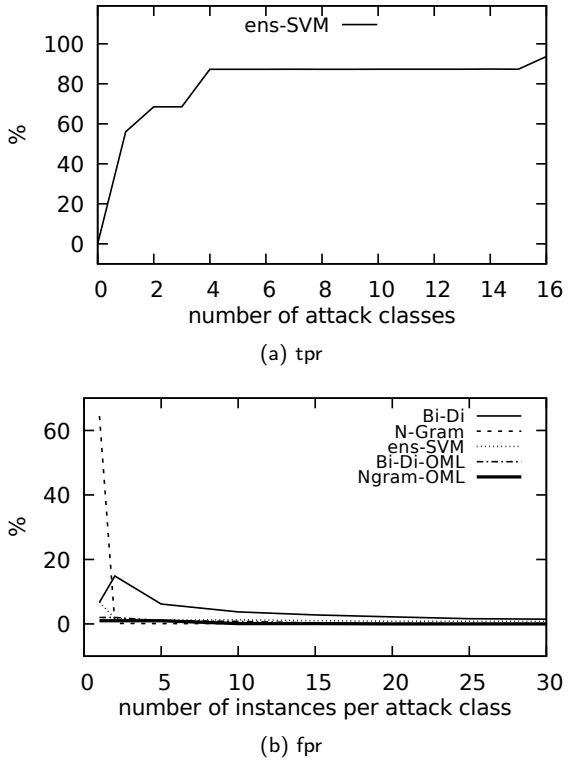


Figure 9: Experimental results showing (a) Ens-SVM classification *tpr* for 0–16 attack classes for training on decoy data and testing on unpatched server data; (b) False positive rates for various training set sizes.

6.3. Base Detection Analysis

In this section we measure the success of DEEPDIG in detecting intrusions in the realistic scenario where attacks are a small fraction of the interactions. Although risk-level attribution for cyber attacks is difficult to quantify in general, we use the results of a prior study [49] to approximate the probability of attack occurrence for the specific scenario of *targeted attacks against business and commercial organizations*. The study's model assumes a determined attacker leveraging one or more exploits of known vulnerabilities to penetrate a typical organization's internal network, and approximates the *prior* of a directed attack to $P_A = 1\%$ (based on real-world threat statistics).

To estimate the success of intrusion detection, we use a *base detection rate* (*bdr*) [65], expressed using the Bayes theorem:

$$P(A|D) = \frac{P(A)\ P(D|A)}{P(A)\ P(D|A) + P(\neg A)\ P(D|\neg A)]}, \qquad (5)$$

where A and D are random variables denoting the occurrence of a targeted attack and the detection of an attack by the classifier, respectively. We use *tpr* and *fpr* as approximations of $P(D|A)$ and $P(D|\neg A)$, respectively.

The final columns of Tables 4–5 present the *bdr* for each classifier, assuming $P(A) = P_A$. The parenthesized comparisons show how our approach overcomes a significant practical problem in intrusion detection research: Despite exhibiting high accuracy, typical IDSes are rendered ineffective when confronted with their extremely low base detection rates. This is in part due to their inability to eliminate false positives in operational contexts. In contrast, the *fpr*-reducing properties of deception-enhanced defense facilitate much more effective detection of intrusions in realistic settings, with *bdr* increases of up to 97%.

6.4. Resistance to Attack Evasion

To properly challenge deceptive defenses, it is essential to simulate adversaries who adapt and obfuscate their behaviors in response to observed responses to their attacks. Attackers employ various evasion techniques to bypass protections, including packet size padding, packet timing sequence morphing, and modifying data distributions to resemble legitimate traffic.

In our study, we considered three encrypted traffic evasion techniques published in the literature: *Pad-to-MTU* [50], *Direct Target Sampling* [121], and *Traffic Morphing* [121]. Pad-to-MTU (pMTU) adds extra bytes to each packet length until it reaches the Maximum Transmission Unit (1500 bytes in the TCP protocol). Direct Target Sampling (DTS) is a distribution-based technique that uses statistical random sampling from benign traffic followed by attack packet length padding. Traffic Morphing (TM) is similar to DTS but it uses a convex optimization methodology to minimize the overhead of padding. Each of these are represented using the traffic modeling approach detailed in §3 and analyzed using the machine learning approaches detailed in §4.

Table 6 shows the results of the deceptive defense against our evasive attack techniques compared with results when no evasion is attempted. In each experiment, the classifier is trained and tested with 1800 normal instances and 1600 *morphed* attack instances. Our evaluation shows that the tpr drops slightly and the fpr increases with the introduction of attacker evasion techniques. This shows that the system could resist some of the evasions but not all. Classifier retraining frequency might also need to be increased to accommodate the drop in performance. This might be a challenge as shorter time intervals result in fewer data points to retrain the classifier to maintain its detection performance.

Table 6
Detection performance in adversarial settings.

Evasion technique	*tpr*	*fpr*	*acc*	F_2
No evasion	**93.63**	**0.01**	**97.00**	**99.06**
pMTU	75.84	0.96	85.78	79.57
DTS	82.78	6.02	87.58	84.91
TM	79.29	6.17	85.52	81.91

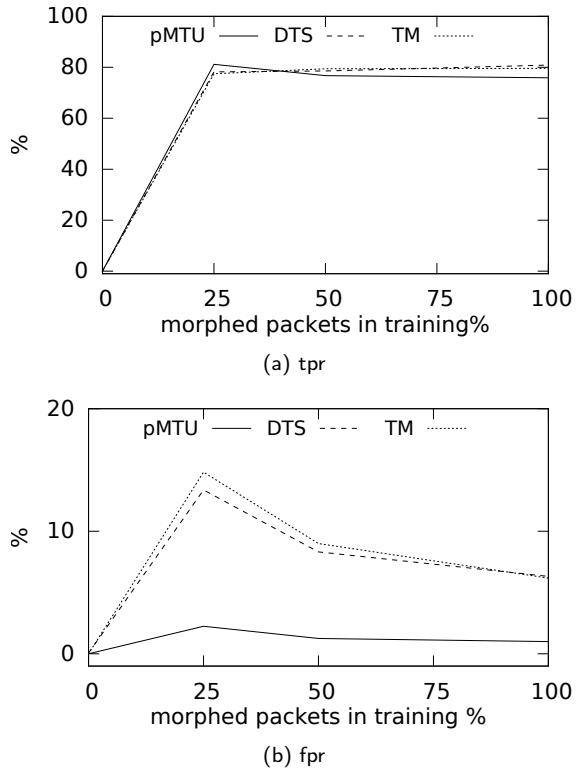
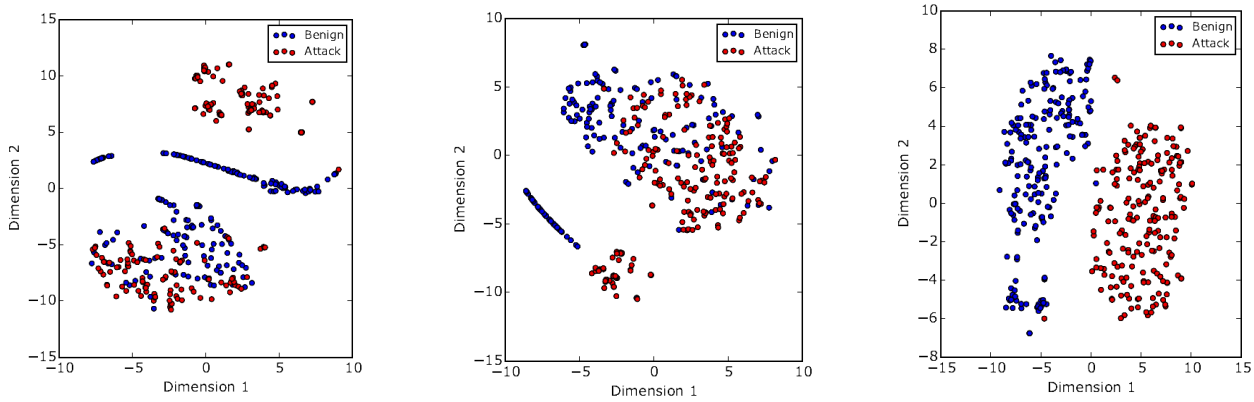


Figure 10: Accuracy when training the classifier with increasingly large proportions (0–100%) of morphed packets.

Analysis. Since our goal is to mine attack patterns from entire data streams rather than merely to classify individual packets as attacks, mixing benign with malicious activities in the decoy environment does not impair DEEPDIG's ability to learn attacker patterns, even in the presence of evasive behavior. In the above experiments, we trained the classifier in the presence of attacker evasion. This is practical and reveals that DEEPDIG captures the entirety of the attacker's activity, feeding it back to the classifier.

Figure 10(a)–(b) shows the measured *tpr* and *fpr* when gradually training the classifier with increasingly large proportions of morphed packets in the training set. The horizontal axis represents the percentage of the morphed packets in the training phase. For instance, 25% signifies that the classifier was trained with 1/4 of morphed packets and 3/4 of non-morphed packets.

Although tpr remains stable after 25% of morphed traffic, the results highlight an improvement for the false positive rate.



(a) *tm*=0% (b) *tm*=50% (c) *tm*=100%

Figure 11: High-dimensional visualization of decision boundary convergence in the presence of evasion, showing traffic morphing (*tm*) at 0%, 50%, and 100%. t-SNE transformation [83] was used to reduce the dimensionality to two dimensions for this visualization.

Table 7
Novel attack class detection performance.

Features	Classifier	*tpr*	*fpr*
Bi-Di	OneSVM	44.06	31.88
	DAE & OneSVM	76.54	85.61
	ECSMiner	74.91	26.66
	DAE & ECSMiner	**84.73**	**0.01**
N-Gram	OneSVM	54.25	45.13
	DAE & OneSVM	80.09	71.49
	ECSMiner	76.36	34.89
	DAE & ECSMiner	**89.67**	**2.95**

This underscores the positive impact of honey-patching on overcoming adversarial behavior: The more morphed, evading samples attackers feed DEEPDIG, the better the IDS becomes in classifying future attack patterns. Thus, adversarial attempts to obfuscate attacks against honey-patched vulnerabilities are actually a gift to the defender, since they only help train the classifier to learn the attacker's obfuscation strategies. Figure 11 illustrates this by showing the convergence of the classifier's decision boundary as it observes morphed samples.

6.5. Novel Class Detection Accuracy

To test the ability of our novel class classifier (§4.3) to detect novel classes emerging in the monitoring stream, we split the input stream into equal-sized chunks. A chunk of 100 instances is classified at a time where one or more novel classes may appear along with existing classes. We measured the *tpr* (total incremental number of actual novel class instances classified as novel classes) and the *fpr* (total number of existing class instances misclassified as belonging to a novel class).

One-class SVM Ensemble. For our comparisons, we built an ensemble of one-class SVM classifiers. One-class SVM is an unsupervised learning method that learns the decision boundary of training instances and predicts whether an instance is inside it. We train one classifier for each class. For instance, if our training data consists of instances of k classes, our ensemble must contain k one-class SVM classifiers, each trained with one of the k class's instances. During classification, once a new unlabeled instance x emerges, we classify it using all the one-class SVM classifiers in the ensemble.

Analysis. Table 7 shows the results for OneSVM and ECSMiner. Here ECSMiner outperforms OneSVM in all measures. For example, for Bi-Di features, ECSMiner observes an *fpr* of 26.66% while OneSVM reports an *fpr* of 31.88%, showing that the binary-class nature of ECSMiner is capable of modeling the decision boundary better than OneSVM. To achieve better accuracy, we augmented ECSMiner with extracted deep abstract features using our stacked denoising autoencoder approach (DAE & ECSMiner). For DAE, we used two hidden layers (where the number of units in the first hidden layer is $2/3$ of the original features, and the number of units in the second hidden layer is $1/3$ of the first hidden layer units). For the additive Gaussian noise, which is used for data corruption, we assigned $\sigma = 1.1$. As a result, *fpr* reduced to a minimum (0.01%), showing a substantial improvement over ECSMiner. Notice that using the abstract features with OneSVM does not help as shown in the table.

These results show that our novel class detection technique is effective in detecting concept drifts, and can be used as a powerful heuristics for timely updating the IDS detection models with additional training data to adapt to the new concepts observed in the monitoring stream.

6.6. Monitoring Performance

To assess the performance overhead of DEEPDIG's monitoring capabilities, we used *ab* (Apache HTTP server benchmarking tool) to create a massive user workload (more than 5,000 requests in 10 threads) against two web server containers, one deployed with network and system call monitoring and another unmonitored.

Figure 12 shows the results, where web server response times are ordered ascendingly. Our measurements show average overheads of 0.2×, 0.4×, and 0.7× for the first 100, 250, and 500 requests, respectively, which is expected given the heavy

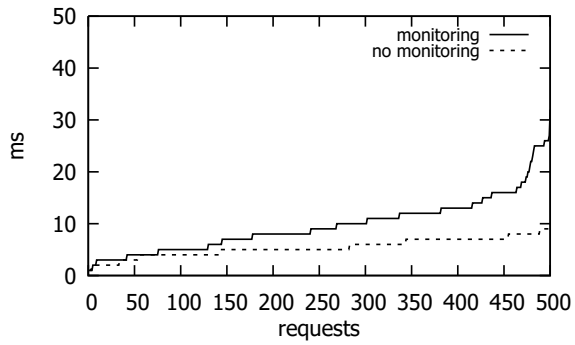


Figure 12: DEEPDIG performance overhead measured in average round-trip times (workload ≈ 500 req/s)

workload profile imposed on the server. Since server computation accounts for only about 10% of overall web site response delay in practice [108], this corresponds to observable overheads of about 2%, 4%, and 7% (respectively).

While such overhead characterizes feasibility, it is irrelevant to deception because unpatched, patched, and honey-patched servers are all slowed equally by the monitoring activity. The overhead therefore does not reveal which apparent vulnerabilities in a given server instance are genuine patching lapses and which are deceptions, and it does not distinguish honey-patched servers from servers that are slowed by any number of other factors (e.g., fewer computational resources).

7. Discussion

Role of deception. Leveraging advanced machine learning to effortlessly detect and pwn cyberattackers (similar to how ML now dominates other games, like chess) has long been a dream of defenders. Although the power of ML has grown exponentially thanks to Moore's Law, this dream has remained unrealized due to the scarcity of large, realistic, evolving, labeled streams of attack data needed to train ML models. 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 [26, 107]. Results demonstrate that even short-term deceptive responses to cyberattacks can significantly ameliorate both of these challenges. 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. This suggests that deception can and should play a more significant role in machine learning-based IDS deployments.

Generalization. The results presented in §6 show that our approach substantially improves the accuracy of intrusion detection, reducing false alarms to much more practical levels. This is established experimentally with moderate- to large-scale synthetic attacks and a small-scale red teaming study. Future work should explore larger numbers of attack classes and larger (e.g., industrial scale) datasets to simulate threats to high-profile targets. Due to the high-dimensional nature of the collected data, we chose OAML and SVM in Bi-Di and N-Gram. However, our approach is agnostic to the feature set and classification model; therefore, future work should study the effectiveness of deception for enhancing a variety of learning frameworks.

An avenue of future work is to leverage system call arguments in addition to the features we collected. A common technique is to use pairwise similarity between arguments (as sequences) of different streams [26], and then implement a k-NN (k-Nearest Neighbors) algorithm with longest common subsequence (LCS) as its distance metric. Generally, packet- and system-level data are very diverse and contain other discriminating features that should be explored.

Online training. The flood of data that is continuously streamed into a typical IDS demands methods that support fast, online classification. Prior approaches update the classification model incrementally using training batches consisting of one or more training instances. However, this strategy necessitates frequently re-training the classifier, and requires a significant number of instances per training. Future research should investigate more appropriate conditions for re-training the model. *Change point detection* (CPD) [59] is one promising approach to determine the optimal re-training predicate, based on a dynamic sliding window that tracks significant changes in the incoming data, and therefore resists concept-drift failures.

Class imbalance. Standard concept-learning IDSes are frequently challenged with imbalanced datasets [60]. Such class imbalance problems arise when benign and attack classes are not equally represented in the training data, since machine learning algorithms tend to misclassify minority classes. To mitigate the effects of class imbalance, sampling techniques have been proposed [32], but they often discard useful data (in the case of under-sampling), or lead to poor generalizations (in the case of oversampling). This scarcity of realistic, balanced datasets has hindered the applicability of machine learning approaches for web intrusion detection. By feeding back labeled attack traces into the classifier, DEEPDIG alleviates this data drought and enables the generation of adequate, balanced datasets for classification-based intrusion detection.

Intrusion detection datasets. One of the major challenges in evaluating intrusion detection systems is the dearth of publicly available datasets [40], 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 [17]. Our evaluation approach builds on recent breakthroughs in dataset generation for IDS evaluation [19] to create statistically representative workloads that resemble realistic web traffic, thereby affording the ability to perform a meaningful evaluation of IDS frameworks.

One of the major challenges for evaluation of deceptive IDSes is the general inadequacy of static attack datasets, which cannot react to deceptive interactions. Testing deceptive defenses with these datasets renders the deceptions useless, missing their

Table 8
Summary of related IDS techniques (cf. [55, 26, 96, 87, 90, 36, 44] for comprehensive surveys on intrusion detection).

			Features		Learning			Deception	
Type	**Technique**	**Source**	**Type**	**Self-labeling**	**Supervised**	**Model**	**Continual**	**Type**	**Site**
NIDS	Eskin et al. [52], Awad et al. [11]	Network traffic	Connection attributes	✗	✓	SVM	✗	✗	—
	Valdes and Skinner [42]	Network traffic	TCP bursts	✗	✗	Bayesian network	✗	✗	—
	Lee and Xiang [80]	Network traffic	Statistical features	✗	✗	Entropy	✗	✗	—
	Kruegel et al. [77, 75, 76]	Network traffic	Histogram	✗	✗	Statistical	✗	✗	—
HIDS	Marceau [85], Cohen [33], Warrender et al. [120]	System calls	Histogram	✗	✗	Cluster	✗	✗	—
	Shu et al. [106]	System calls	Event (co-) occurrence	✗	✗	Cluster	✗	✗	—
	Liu et al. [38]	Log entries	Graph embedding	✗	✗	Cluster	✗	✗	—
	Han et al. [37]	System calls	Graph embedding	✗	✗	Cluster	✗	✗	—
	Yuan et al. [43]	User activity	Temporal features	✗	✓	CNN	✗	✗	—
	Yuan et al. [45]	User activity	Temporal features	✗	✗	RNN-LSTM	✗	✗	—
Deceptive IDS	Portokalidis et al. [99], Tang and Chen [113], Kreibichi and Crowcroft [74], Anagnostakis et al. [5, 4]	Network traffic	—	✗	✗	Signature	✓*	Honeypot	Standalone
	DeepDig (ΔaaS)	Network traffic, System calls	Histogram	✓	✓	SVM OAML	✓	**Honey-patch**	**Embedded**

*attack signatures are generated from honeypots to update a signature-based IDS.

value against reactive threats. Establishing a straight comparison of our results to prior work is therefore frustrated by the fact that the majority of machine learning-based intrusion detection techniques are still tested on extremely old datasets [1, 107], and approaches that account for encrypted traffic are scarce [73]. For instance, recently-proposed SVM-based approaches for network intrusion detection have reported true positive rates in the order of 92% for the DARPA/KDD datasets, with false positive rates averaging 8.2% [84, 125]. Using the model discussed in §6.3, this corresponds to an approximate base detection rate of only 11%, in contrast to 99.06% estimated for our approach. However, the assumptions made by DARPA/KDD do not reflect the contemporary attack protocols and recent vulnerabilities targeted in our evaluation, so this might not be a fair comparison. Future work should consider reevaluating these prior approaches using updated datasets reflective of modern attacks (cf. DARPA TC [35], CSE-CIC-IDS2018 [41]), for reproducible comparisons.

Hybrid cloud. ΔaaS transparently enhances existing cloud infrastructures with crook-sourced data streams, enabling a new form of attack-defenses co-evolution in commodity cloud and service-oriented infrastructures. Assessing how the unique set of features and characteristics of each cloud provider affects deception delivery in public clouds is planned for future work. In particular, we plan to investigate architectural properties (e.g., container runtimes and orchestration, storage services, serverless computing) that will facilitate the implementation of multi-layer deception strategies across the deception stack.

8. Related Work

The scarcity of labeled, concept-relevant attack data introduces significant limitations in deploying and scaling conventional machine learning approaches to intrusion detection [55, 26, 96, 87, 90, 36, 44]. This data limitation hinders timely IDS model adaptation to cope with new and emergent attack patterns. Deceptive IDSes [5, 4, 99, 74, 113] overcome these disadvantages by building detection models with attack interactions collected from successful deceptions.

Table 8 summarizes these IDS techniques. Compared to prior intrusion detection approaches, DEEPDIG employs *embedded deceptions* (i.e., they integrate into the production service) to *continuously* train an IDS with *self*-labeled, concept-relevant attack data gathered from the deceptions. This enhances intrusion detection by enabling domain adaptation, and facilitates the implementation of more accurate *supervised* IDS approaches.

8.1. ML-based Intrusion Detection

Machine learning-based IDSes [55, 26, 96] find patterns that do not conform to expected system behavior, and are typically classified into *host-based* and *network-based* approaches.

Host-based detectors recognize intrusions in the form of anomalous system call trace sequences, in which co-occurrence of events is key to characterizing malicious behavior. For example, malware activity and privilege escalation often manifest specific system call patterns [26]. Seminal work in this area has analogized intrusion detection via statistical profiling of system events to the human immune system [53, 61]. This has

been followed by a number of related approaches using histograms to construct profiles of normal behavior [85]. Another frequently-used approach employs a *sliding window* classifier to map sequences of events into individual output values [120, 33], converting sequential learning into a classic machine learning problem. More recently, long call sequences have been studied to detect attacks buried in long execution paths [106].

Network-based approaches detect intrusions using network data. Since such systems are typically deployed at the network perimeter, they are designed to find patterns resulting from attacks launched by external threats, such as attempted disruption or unauthorized access [17]. Network intrusion detection has been extensively studied in the literature (cf., [17, 1]). Major approaches can be grouped into classification-based (e.g., SVM [52], [11], Bayesian network [42]), information-theoretic [80], and statistical [77, 75, 76] techniques.

Network-based intrusion detection systems can monitor a large number of hosts at relatively low cost, but they are usually opaque to *local* or *encrypted* attacks. On the other hand, intrusion detection systems operating at the host level have complete visibility of malicious events, despite encrypted network payloads and obfuscation mechanisms [72]. Our approach therefore complements existing techniques and incorporates host- and network-based features to offer protective capabilities that can resist attacker evasion strategies and detect malicious activity bound to different layers of the software stack.

Another related area of research is *web-based* malware detection that identifies drive-by-download attacks using static analysis, dynamic analysis, and machine learning [67, 24]. In addition, other studies focus on *flow-based* malware detection by extracting features from proxy-logs and using machine learning [15].

8.2. Cyber-Deception in Intrusion Detection

Honeypots are information systems resources conceived to attract, detect, and gather attack information [109]. They are designed such that any interaction with a honeypot is likely to be malicious. Previous works have utilized honeypots for automating the generation of IDS signatures [99, 74, 113]. Shadow honeypots [4] are a hybrid approach in which a front-end anomaly detection system forwards suspicious requests to a back-end instrumented copy of the target application, which validates the anomaly prediction and improves the anomaly detector's heuristics through feedback. Although the target and instrumented programs may share similar states for detection purposes, shadow honeypots make no effort to deceive attackers into thinking the attack was successful—attack detection and the decision of decoying attacker sessions are driven solely by the network anomaly detection component.

8.3. Feature Extraction for Intrusion Detection

A variety of feature extraction and classification techniques have been proposed to perform host- and network-based anomaly detection [86]. Extracting features from encrypted network packets has been intensively studied in the domain of *website fingerprinting*, where attackers attempt to discern which websites are visited by victims. Users typically use anonymous networks, such as Tor, to hide their destination websites [119]. However, attackers can often predict destinations by training classifiers directly on encrypted packets (e.g., packet headers only). Relevant features typically include packet length and direction, summarized as a histogram feature vector. HTML markers, percentage of incoming and outgoing packets, bursts, bandwidth, and website upload time have also been used [95, 50]. Packet-word vector approaches additionally leverage natural language processing and vector space models to convert packets to word features for improved classification [3].

Bi-Di leverages packet and uni-burst data and introduces bi-directional bursting features for better classification of network streams. On unencrypted data, host-based systems have additionally extracted features from co-occurrences and sequences of system events, such as system calls [23, 85]. DEEPDIG uses a hybrid scheme that combines both host- and network-based approaches via a modified ensemble technique.

9. Conclusion

This paper introduced, implemented, and evaluated a new service-oriented security model for enhancing web intrusion detection systems with threat data sourced from deceptive, application-layer, software traps. Unlike conventional machine learning-based detection approaches, DEEPDIG 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.

Acknowledgement

The research reported herein was supported in part by ARO award W911NF2110032; ONR award N00014-17-1-2995; NSA award H98230-15-1-0271; AFOSR award FA9550-14-1-0173; an endowment from the Eugene McDermott family; NSF FAIN awards DGE-1931800, OAC-1828467, and DGE-1723602; NSF awards DMS-1737978 and MRI-1828467; an IBM faculty award (Research); and an HP grant. Any opinions, recommendations, or conclusions expressed are those of the authors and not necessarily of the aforementioned supporters.

References

[1] Ahmed, M., Mahmood, A.N., Hu, J., 2016. A survey of network anomaly detection techniques. Journal of Network and Computer Applications 60, 19–31.

[2] Al-Khateeb, T., Masud, M.M., Al-Naami, K.M., Seker, S.E., Mustafa, A.M., Khan, L., Trabelsi, Z., Aggarwal, C., Han, J., 2016. Recurring and novel class detection using class-based ensemble for evolving data stream. IEEE Transactions on Knowledge and Data Engineering 28, 2752–2764.

[3] Alnaami, K., Ayoade, G., Siddiqui, A., Ruozzi, N., Khan, L., Thuraisingham, B., 2015. P2V: Effective website fingerprinting using vector space representations, in: Proceedings of the IEEE Symposium on Computational Intelligence, pp. 59–66.

[4] Anagnostakis, K.G., Sidiroglou, S., Akritidis, P., Polychronakis, M., Keromytis, A.D., Markatos, E.P., 2010. Shadow honeypots. International Journal of Computer and Network Security (IJCNS) 2, 1–15.

[5] Anagnostakis, K.G., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E., Keromytis, A.D., 2005. Detecting targeted attacks using shadow honeypots, in: Proceedings of the 14th USENIX Security Symposium.

[6] Ansible, 2020. Red hat ansible. https://www.ansible.com. Accessed May 1, 2020.

[7] Araujo, F., Ayoade, G., Al-Naami, K., Gao, Y., Hamlen, K.W., Khan, L., 2019. Improving intrusion detectors by crook-sourcing, in: Procceedings of the 35th Annual Computer Security Applications Conference (ACSAC), pp. 245–256.

[8] Araujo, F., Hamlen, K.W., 2015. Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception, in: Proceedings of the 24th USENIX Security Symposium.

[9] Araujo, F., Hamlen, K.W., Biedermann, S., Katzenbeisser, S., 2014. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation, in: Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS), pp. 942–953.

[10] Araujo, F., Shapouri, M., Pandey, S., Hamlen, K., 2015. Experiences with honey-patching in active cyber security education, in: Proceedings of the 8th USENIX Conference on Cyber Security Experimentation and Test (CSET).

[11] Awad, M., Khan, L., Bastani, F., Yen, I.L., 2004. An effective support vector machines (SVMs) performance using hierarchical clustering, in: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), pp. 663–667.

[12] Axelsson, S., 1999. The base-rate fallacy and its implications for the difficulty of intrusion detection, in: Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS), pp. 1–7.

[13] Ayoade, G., Araujo, F., Al-Naami, K., Mustafa, A.M., Gao, Y., Hamlen, K.W., Khan, L., 2020. Automating cyberdeception evaluation with deep learning, in: Proceedings of the 53rd Hawaii International Conference on System Sciences (HICSS).

[14] Banerjee, P., Friedrich, R., Bash, C., Goldsack, P., Huberman, B., Manley, J., Patel, C., Ranganathan, P., Veitch, A., 2011. Everything as a Service: Powering the new information economy. IEEE Computer 44, 36–43.

[15] Bartos, K., Sofka, M., Franc, V., 2016. Optimized invariant representation of network traffic for detecting unseen malware variants, in: Proceedings of the 25th USENIX Security Symposium, pp. 807–822.

[16] Bengio, Y., 2009. Learning deep architectures for ai. Foundations and Trends® in Machine Learning 2, 1–127. URL: http://dx.doi.org/10.1561/2200000006.

[17] Bhuyan, M.H., Bhattacharyya, D.K., Kalita, J.K., 2014. Network anomaly detection: Methods, systems and tools. IEEE Communications Surveys & Tutorials 16, 303–336.

[18] Blum, A.L., Langley, P., 1997. Selection of relevant features and examples in machine learning. Artificial Intelligence 97, 245–271.

[19] Boggs, N., Zhao, H., Du, S., Stolfo, S.J., 2014. Synthetic data generation and defense in depth measurement of web applications, in: Proceedings of the 17th International Symposium on Recent Advances in Intrusion Detection (RAID), pp. 234–254.

[20] Bowers, K.D., Juels, A., Oprea, A., 2009. HAIL: A high-availability and integrity layer for cloud storage, in: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS), pp. 187–198.

[21] Breen, C., Khan, L., Ponnusamy, A., 2002. Image classification using neural networks and ontologies, in: Proceedings of the 13th International Workshop on Database and Expert Systems Applications, pp. 98–102.

[22] Burger, R.A., Cachin, C., Husmann, E., 2013. Cloud, Trust, Privacy: Trustworthy Cloud Computing Whitepaper. Technical Report. TClouds Project.

[23] Cabrera, J.B., Lewis, L., Mehra, R.K., 2001. Detection and classification of intrusions and faults using sequences of system calls. ACM SIGMOD Record 30, 25–34.

[24] Canali, D., Cova, M., Vigna, G., Kruegel, C., 2011. Prophiler: A fast filter for the large-scale detection of malicious web pages, in: Proceedings of the 20th International World Wide Web Conference (WWW), pp. 197–206.

[25] Canonical, 2021. Juju application and service modeling tool. https://jaas.ai. Accessed March 1, 2021.

[26] Chandola, V., Banerjee, A., Kumar, V., 2009. Anomaly detection: A survey. ACM Computing Surveys (CSUR) 41, 15.

[27] Chang, C.C., Lin, C.J., 2011. LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology (TIST) 2.

[28] Chechik, G., Sharma, V., Shalit, U., Bengio, S., 2010. Large scale online learning of image similarity through ranking. Journal of Machine Learning Research (JMLR) 11, 1109–1135.

[29] Chef, 2016. Chef configuration management tool. https://www.chef.io. Accessed May 1, 2020.

[30] Chen, M., Weinberger, K.Q., Sha, F., Bengio, Y., 2014. Marginalized denoising auto-encoders for nonlinear representations., in: ICML, pp. 1476–1484.

[31] Chow, R., Golle, P., Jakobsson, M., Shi, E., Staddon, J., Masuoka, R., Molina, J., 2009. Controlling data in the cloud: Outsourcing computation without outsourcing control, in: Proceedings of the ACM Workshop on Cloud Computing Security, pp. 85–90.

[32] Cieslak, D.A., Chawla, N.V., Striegel, A., 2006. Combating imbalance in network intrusion datasets, in: Proceedings of the IEEE International Conference on Granular Computing (GrC), pp. 732–737.

[33] Cohen, W.W., 1995. Fast effective rule induction, in: Proceedings of the 12th International Conference on Machine Learning, pp. 115–123.

[34] Araujo, F., Taylor, T., 2020. Improving cybersecurity hygiene through JIT patching, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 1421–1432.

[35] DARPA, 2020. Darpa transparent computing apt dataset. https://github.com/darpa-i2o/Transparent-Computing. Accessed March 10, 2021.

[36] Ferrag, M.A., Maglaras, L., Moschoyiannis, S., Janicke, H., 2020. Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study. Journal of Information Security and Applications 50.

[37] Han, X., Pasquier, T., Bates, A., Mickens, J., Seltzer, M., 2020. Unicorn: Runtime provenance-based detector for advanced persistent threats, in: Proceedings of the Network & Distributed System Security Symposium (NDSS).

[38] Liu, F., Wen, Y., Zhang, D., Jiang, X., Xing, X., Meng, D., 2019. Log2vec: a heterogeneous graph embedding based approach for detecting cyber threats within enterprise, in: Proceedings of the ACM Conference on Computer and Communications Security (CCS), pp. 1777–1794.

[39] Mordor Intelligence, 2018. Global Cyber Deception Market. Technical Report. Mordor Intelligence.

[40] Ring, M., Wunderlich, S., Scheuring, D., Landes, D., Hotho, A., 2019. A survey of network-based intrusion detection data sets. Computers & Security 86, 147–167.

[41] Sharafaldin, I., Lashkari, A.H., Ghorbani, A.A., 2018. Toward generating a new intrusion detection dataset and intrusion traffic characterization, in: Proceedings of the International Conference on Information Systems Security and Privacy, pp. 108–116.

[42] Valdes, A., Skinner, K., 2000. Adaptive, model-based monitoring for cyber attack detection, in: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID), Springer. pp. 80–93.

[43] Yuan, F., Cao, Y., Shang, Y., Liu, Y., Tan, J., Fang, B., 2018. Insider threat detection with deep neural network, in: International Conference on Computational Science, Springer. pp. 43–54.

[44] Yuan, S., Wu, X., 2021. Deep learning for insider threat detection: Review, challenges and opportunities. Computers & Security.

[45] Yuan, S., Zheng, P., Wu, X., Li, Q., 2019. Insider threat detection via hierarchical neural temporal point processes, in: IEEE International Conference on Big Data (Big Data), IEEE. pp. 1343–1350.

[46] Cortes, C., Vapnik, V., 1995. Support-vector networks. Machine Learning 20, 273–297.

[47] Denning, D.E., 1987. An intrusion-detection model. IEEE Transactions on Software Engineering (TSE) 13, 222–232.

[48] DiMaggio, J., 2015. The Black Vine cyberespionage group. Symantec Security Response.

[49] Dudorov, D., Stupples, D., Newby, M., 2013. Probability analysis of cyber attack paths against business and commercial enterprise systems, in: Proceedings of the IEEE European Intelligence and Security Informatics

Conference (EISIC), pp. 38–44.
[50] Dyer, K.P., Coull, S.E., Ristenpart, T., Shrimpton, T., 2012. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail, in: Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P), pp. 332–346.
[51] Edgescan, 2019. Vulnerability statistics report.
[52] Eskin, E., Arnold, A., Prerau, M., Portnoy, L., Stolfo, S., 2002. A geometric framework for unsupervised anomaly detection, in: Applications of Data Mining in Computer Security. Springer, pp. 77–101.
[53] Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A., 1996. A sense of self for Unix processes, in: Proceedings of the 17th IEEE Symposium on Security & Privacy (S&P), pp. 120–128.
[54] Gao, Y., Li, Y.F., Chandra, S., Khan, L., Thuraisingham, B., 2019. Towards self-adaptive metric learning on the fly, in: Proceedings of the 28th International World Wide Web Conference (WWW), pp. 503–513.
[55] Garcia-Teodoro, P., Diaz-Verdejo, J., Maciá-Fernández, G., Vázquez, E., 2009. Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers & Security 28, 18–28.
[56] Greene, D., Cunningham, P., 2006. Practical solutions to the problem of diagonal dominance in kernel document clustering, in: Proceedings of the 23rd International Conference on Machine learning (ICML), pp. 377–384.
[57] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The WEKA data mining software: An update. ACM SIGKDD Explorations Newsletter 11, 10–18.
[58] Hamlen, K.W., Morrisett, G., Schneider, F.B., 2006. Computability classes for enforcement mechanisms. ACM Transactions on Programming Languages and Systems (TOPLAS) 28, 175–205.
[59] Haque, A., Khan, L., Baron, M., 2016. SAND: Semi-supervised adaptive novel class detection and classification over data stream, in: Proceedings of the 30th Conference on Artificial Intelligence (AAAI), pp. 1652–1658.
[60] He, H., Garcia, E.A., 2009. Learning from imbalanced data. IEEE Transactions on Knowledge and Data Engineering (TKDE) 21, 1263–1284.
[61] Hofmeyr, S.A., Forrest, S., Somayaji, A., 1998. Intrusion detection using sequences of system calls. Journal of Computer Security 6, 151–180.
[62] Jain, P., Kulis, B., Dhillon, I.S., Grauman, K., 2008. Online metric learning and fast similarity search, in: Proceedings of the 21st International Conference on Neural Information Processing Systems (NIPS), pp. 761–768.
[63] Jeng, A., 2015. Minimizing damage from J.P. Morgan's data breach. InfoSec Reading Room .
[64] Jin, R., Wang, S., Zhou, Y., 2009. Regularized distance metric learning: Theory and algorithm, in: Proceedings of the 22nd International Conference on Neural Information Processing Systems (NIPS), pp. 862–870.
[65] Juarez, M., Afroz, S., Acar, G., Diaz, C., Greenstadt, R., 2014. A critical evaluation of website fingerprinting attacks, in: Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS), pp. 263–274.
[66] Juniper Research, 2017. The future of cybercrime and security: Key takeaways and juniper leaderboard.
[67] Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., Vigna, G., 2013. Revolver: An automated approach to the detection of evasive web-based malware, in: Proceedings of the 22nd USENIX Security Symposium, pp. 637–652.
[68] Khan, S.M., Hamlen, K.W., 2012a. AnonymousCloud: A data ownership privacy provider framework in cloud computing, in: Proceedings of the 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 170–176.
[69] Khan, S.M., Hamlen, K.W., 2012b. Hatman: Intra-cloud trust management for Hadoop, in: Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD), pp. 494–501.
[70] Khan, S.M., Hamlen, K.W., 2013. Computation certification as a service in the cloud, in: Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 434–441.
[71] Khan, S.M., Hamlen, K.W., Kantarcioglu, M., 2014. Silver lining: Enforcing secure information flow at the cloud edge, in: Proceedings of the 2nd IEEE International Conference on Cloud Engineering (IC2E), pp. 37–46.
[72] Kim, J., Bentley, P.J., Aickelin, U., Greensmith, J., Tedesco, G., Twycross, J., 2007. Immune system approaches to intrusion detection—a review. Natural Computing 6, 413–466.
[73] Kovanen, T., David, G., Hämäläinen, T., 2016. Survey: Intrusion detection systems in encrypted traffic, in: Proceedings of the 16th International Conference on Next Generation Wired/Wireless Networking (NEW^{2}AN), pp. 281–293.
[74] Kreibichi, C., Crowcroft, J., 2004. Honeycomb – creating intrusion detection signatures using honeypots. ACM SIGCOMM Computer Communication Review 34, 51–56.
[75] Kruegel, C., Vigna, G., 2003. Anomaly detection of web-based attacks, in: Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), pp. 251–261.
[76] Kruegel, C., Vigna, G., Robertson, W., 2005. A multi-model approach to the detection of web-based attacks. Computer Networks 48, 717–738.
[77] Krügel, C., Toth, T., Kirda, E., 2002. Service specific anomaly detection for network intrusion detection, in: Proceedings of the 17th ACM Symposium on Applied Computing (SAC), pp. 201–208.
[78] LeCun, Y., Bengio, Y., Hinton, G., 2015. Deep learning. Nature 521, 436–444.
[79] Lee, W., Stolfo, S.J., 1998. Data mining approaches for intrusion detection, in: Proceedings of the 7th USENIX Security Symposium, pp. 79–93.
[80] Lee, W., Xiang, D., 2001. Information-theoretic measures for anomaly detection, in: Proceedings of the 22nd IEEE Symposium on Security & Privacy (S&P), pp. 130–143.
[81] Li, W., Gao, Y., Wang, L., Zhou, L., Huo, J., Shi, Y., 2018. OPML: A one-pass closed-form solution for online metric learning. Pattern Recognition 75, 302–314.
[82] LXC, 2019. Linux containers. http://linuxcontainers.org.
[83] van der Maaten, L., Hinton, G.E., 2008. Visualizing high-dimensional data using t-sne. Journal of Machine Learning Research 9, 2579–2605.
[84] Manandhar, P., Aung, Z., 2014. Towards practical anomaly-based intrusion detection by outlier mining on TCP packets, in: Proceedings of the 25th International Conference on Database and Expert Systems Applications (DEXA), pp. 164–173.
[85] Marceau, C., 2001. Characterizing the behavior of a program using multiple-length n-grams, in: Proceedings of the New Security Paradigms Workshop (NSPW), pp. 101–110.
[86] Masud, M., Khan, L., Thuraisingham, B., 2011a. Data Mining Tools for Malware Detection. CRC Press.
[87] Masud, M.M., Al-Khateeb, T.M., Hamlen, K.W., Gao, J., Khan, L., Han, J., Thuraisingham, B., 2008a. Cloud-based malware detection for evolving data streams. ACM Transactions on Management Information Systems (TMIS) 2.
[88] Masud, M.M., Al-Khateeb, T.M., Khan, L., Aggarwal, C., Gao, J., Han, J., Thuraisingham, B., 2011b. Detecting recurring and novel classes in concept-drifting data streams, in: Proceedings of the 11th International IEEE Conference on Data Mining, pp. 1176–1181.
[89] Masud, M.M., Gao, J., Khan, L., Han, J., Thuraisingham, B., 2008b. A practical approach to classify evolving data streams: Training with limited amount of labeled data, in: Proceedings of the International Conference on Data Mining (ICDM), pp. 929–934.
[90] Masud, M.M., Gao, J., Khan, L., Han, J., Thuraisingham, B., 2010. Classification and novel class detection in data streams with active mining, in: Proceedings of the 14th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), pp. 311–324.
[91] MinIO, 2019. Minio object storage. https://min.io/.
[92] Mockaroo, 2018. Product data set. https://www.mockaroo.com.
[93] Nepal, S., Chen, S., Yao, J., Thilakanathan, D., 2011. DIaaS: Data integrity as a service in the cloud, in: Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD), pp. 308–315.
[94] Novetta Threat Research Group, 2016. Operation Blockbuster: Unraveling the long thread of the Sony attack.
[95] Panchenko, A., Niessen, L., Zinnen, A., Engel, T., 2011. Website fingerprinting in onion routing based anonymization networks, in: Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society (WPES), pp. 103–114.

[96] Patcha, A., Park, J.M., 2007. An overview of anomaly detection techniques: Existing solutions and latest technological trends. Computer Networks 51, 3448–3470.
[97] Pearson, S., 2009. Taking account of privacy when designing cloud computing services, in: Proceedings of the ICSE Workshop on Software Engineering Challenges of Cloud Computing, IEEE. pp. 44–52.
[98] Platt, J.C., 1999. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods, in: Advances in Large Margin Classifiers. MIT Press, pp. 61–74.
[99] Portokalidis, G., Slowinska, A., Bos, H., 2006. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. ACM SIGOPS Operating Systems Review 40, 15–27.
[100] Puppet, 2016. Puppet configuration management tool. https://www.puppet.com. Accessed May 1, 2020.
[101] PyTorch, 2019. Open source deep learning platform. https://pytorch.org.
[102] Gorka Sadowski, Kau, R., 2019. Improve Your Threat Detection Function With Deception Technologies. Technical Report G00382578. Gartner.
[103] Sager, T., 2014. Killing advanced threats in their tracks: An intelligent approach to attack prevention. InfoSec Reading Room .
[104] Santos, N., Gummadi, K.P., Rodrigues, R., 2009. Towards trusted cloud computing. Proceedings of the USENIX Workshop in Hot Topics in Cloud Computing (HotCloud) .
[105] Selenium, 2019. Selenium browser automation. http://www.seleniumhq.org.
[106] Shu, X., Yao, D., Ramakrishnan, N., 2015. Unearthing stealthy program attacks buried in extremely long execution paths, in: Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS), pp. 401–413.
[107] Sommer, R., Paxson, V., 2010. Outside the closed world: On using machine learning for network intrusion detection, in: Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P), pp. 305–316.
[108] Souders, S., 2007. High Performance Web Sites: Essential Knowledge for Front-End Engineers. O'Reilly.
[109] Spitzner, L., 2002. Honeypots: Tracking Hackers. Addison-Wesley.
[110] Symantec, 2018. Internet security threat report, vol. 23.
[111] Sysdig, 2021. Universal system visibility tool. https://github.com/draios/sysdig.
[112] Takabi, H., Joshi, J.B., Ahn, G.J., 2010. Security and privacy challenges in cloud computing environments. IEEE Security & Privacy 8, 24–31.
[113] Tang, Y., Chen, S., 2005. Defending against internet worms: A signature-based approach, in: Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), pp. 1384–1394.
[114] tcpdump, 2021. Tcpdump and libpcap. https://www.tcpdump.org/.
[115] Tsai, C.F., Hsu, Y.F., Lin, C.Y., Lin, W.Y., 2009. Intrusion detection by machine learning: A review. Expert Systems with Applications 36, 11994–12000.
[116] Vasilomanolakis, E., Karuppayah, S., Mühlhäuser, M., Fischer, M., 2015. Taxonomy and survey of collaborative intrusion detection. ACM Computing Surveys 47.
[117] Vincent, P., Larochelle, H., Bengio, Y., Manzagol, P.A., 2008. Extracting and composing robust features with denoising autoencoders, in: Proceedings of the 25th international conference on Machine learning, ACM. pp. 1096–1103.
[118] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P.A., 2010. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. Journal of Machine Learning Research 11, 3371–3408.
[119] Wang, T., Cai, X., Nithyanand, R., Johnson, R., Goldberg, I., 2014. Effective attacks and provable defenses for website fingerprinting, in: Proceedings of the 23rd USENIX Security Symposium.
[120] Warrender, C., Forrest, S., Pearlmutter, B., 1999. Detecting intrusions using system calls: Alternative data models, in: Proceedings of the 20th IEEE Symposium on Security & Privacy (S&P), pp. 133–145.
[121] Wright, C.V., Coull, S.E., Monrose, F., 2009. Traffic morphing: An efficient defense against statistical traffic analysis, in: Proceedings of the 16th IEEE Network and Distributed Security Symposium, pp. 237–250.
[122] Xiang, S., Nie, F., Zhang, C., 2008. Learning a mahalanobis distance metric for data clustering and classification. Pattern Recognition 41, 3600–3612.
[123] Yao, D., Shu, X., Cheng, L., Stolfo, S.J., Bertino, E., Sandhu, R., 2017. Anomaly Detection as a Service: Challenges, Advances, and Opportunities. Synthesis Lectures on Information Security, Privacy, and Trust, Morgan & Claypool Publishers.
[124] Yuill, J., Denning, D., Feer, F., 2006. Using deception to hide things from hackers: Processes, principles, and techniques. Journal of Information Warfare 5, 26–40.
[125] Zhang, M., Xu, B., Wang, D., 2015. An anomaly detection model for network intrusions using one-class SVM and scaling strategy, in: Proceedings of the 11th International Conference on Collaborative Computing: Networking, Applications, and Worksharing (CollaborateCom), pp. 267–278.