Detecting Web Attacks with End-to-End Deep Learning

YAO PAN, FANGZHOU SUN, JULES WHITE, DOUGLAS C. SCHMIDT, Vanderbilt University, USA
JACOB STAPLES AND LEE KRAUSE, Securboration Inc., USA

Web applications are popular targets for cyber-attacks because they are network-accessible and often contain vulnerabilities. An intrusion detection system monitors web applications and issues alerts when an attack attempt is detected. Existing implementations of intrusion detection systems usually extract features from network packets or string characteristics of input that are *manually selected* as relevant to attack analysis. Manually selecting features, however, is time-consuming and requires in-depth security domain knowledge. Moreover, large amounts of *labeled* legitimate and attack request data are needed by supervised learning algorithms to classify normal and abnormal behaviors, which is often expensive and impractical to obtain for production web applications. This paper provides three contributions to the study of autonomic intrusion detection systems. First, we evaluate the feasibility of an unsupervised/semi-supervised approach for web attack detection based on the *Robust Software Modeling Tool* (RSMT), which autonomically monitors and characterizes the runtime behavior of web applications. Second, we describe how RSMT trains a stacked denoising autoencoder to encode and reconstruct the call graph for end-to-end deep learning, where a low-dimensional representation of the raw features with unlabeled request data is used to recognize anomalies by computing the reconstruction error of the request data. Third, we analyze the results of empirically testing RSMT on both synthetic datasets and production applications with intentional vulnerabilities. Our results show that the proposed approach can efficiently and accurately detect attacks, including SQL injection, cross-site scripting, and deserialization, with minimal domain knowledge and little labeled training data.

CCS Concepts: • **Security and privacy → Artificial immune systems**; *Web application security*;

Additional Key Words and Phrases: Web security, Deep learning, Application instrumentation

ACM Reference Format:
Yao Pan, Fangzhou Sun, Jules White, Douglas C. Schmidt and Jacob Staples and Lee Krause. 2019. Detecting Web Attacks with End-to-End Deep Learning. *ACM Trans. Autonom. Adapt. Syst.* 1, 1, Article 1 (January 2019), 28 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Emerging trends and challenges. Web applications are attractive targets for cyber attackers. SQL injection [Halfond et al. 2006], cross site scripting (XSS) [Wassermann and Su 2008] and remote code execution are common attacks that can disable web services, steal sensitive user information, and cause significant financial loss to both service providers and users. Protecting web applications from attack is hard. Even though developers and researchers have developed many counter-measures (such as firewalls, intrusion detection systems (IDSs) [Di Pietro and Mancini 2008] and defensive programming best practices [Qie et al. 2002]) to protect web applications, web attacks remain a major threat.

Authors' addresses: Yao Pan, Fangzhou Sun, Jules White, Douglas C. Schmidt, Vanderbilt University, 2201 West End Ave, Nashville, TN, 37235, USA, {yao.pan,fangzhou.sun,jules.white,d.schmidt}@vanderbilt.edu; Jacob Staples and Lee Krause, Securboration Inc. Melbourne, FL, USA, {jstaples,lkrause}@securboration.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
Manuscript submitted to ACM

For example, researchers found that more than half of web applications during a 2015-2016 scan contained significant security vulnerabilities, such as XSS or SQL Injection [cyberattack 2018]. Moreover, hacking attacks cost the average American firm $15.4 million per year [cyberattackloss 2018]. The Equifax data breach in 2017 [equifax 2018; equifax2 2018] (which exploited a vulnerability in Apache Struts) exposed over 143 million American consumers' sensitive personal information. Although the vulnerability was disclosed and patched in March 2017, Equifax took no action until four months later, which led to an estimated insured loss of over 125 million dollars.

Conventional intrusion detection systems do not work as well as expected for a number of reasons, including the following:

• **Workforce limitations**. In-depth domain knowledge of web security is needed for web developers and network operators to deploy these systems [Ben-Asher and Gonzalez 2015]. An experienced security expert is often needed to determine what features are relevant to extract from network packages, binaries, or other inputs for intrusion detection systems. Due to the large demand and relatively low barrier to entry into the software profession, however, many developers lack the necessary knowledge of secure coding practices.

• **Classification limitations**. Many intrusion detection systems rely on rule-based strategies or supervised machine learning algorithms to differentiate normal requests from attack requests, which requires large amounts of labeled training data to train the learning algorithms. It is hard and expensive, however, to obtain this training data for arbitrary custom applications. In addition, labeled training data is often heavily imbalanced since attack requests for custom systems are harder to get than normal requests, which poses challenges for classifiers [Japkowicz and Stephen 2002]. Moreover, although rule-based or supervised learning approaches can distinguish existing known attacks, new types of attacks and vulnerabilities emerge continuously, so they may be misclassified.

• **False positive limitations**. Although prior work has applied unsupervised learning algorithms (such as PCA [Liu et al. 2007] and SVM [Xu and Wang 2005]) to detect web attacks, these approaches require manual selection of attack-specific features. Moreover, while these approaches achieve acceptable performance they also incur false positive rates that are too high in practice, *e.g.*, a 1% increase in false positives may cause an intrusion detection system to incorrectly flag thousands of legitimate users [Pietraszek 2004]. It is therefore essential to reduce the false positive rate of these systems.

Given these challenges with using conventional intrusion detection systems, an infrastructure that requires less expertise and labeled training data is needed.

Solution approach ⇒ Applying end-to-end deep learning to detect cyber-attacks autonomically in real-time and adapt efficiently, scalably, and securely to thwart them. This paper explores the potential of end-to-end deep learning [Goodfellow et al. 2016] in intrusion detection systems. Our approach applies deep learning to the entire process from feature engineering to prediction, *i.e.*, raw input is fed into the network and high-level output is generated directly. There is thus no need for users to select features and construct large labeled training sets manually.

We empirically evaluate how well an unsupervised/semi-supervised learning approach based on end-to-end deep learning detects web attacks. Our work is motivated by the success deep learning has achieved in computer vision [Krizhevsky et al. 2012], speech recognition [Amodei et al. 2016], and natural language processing [Sutskever et al. 2014]. In particular, deep learning is not only capable of classification, but also automatically extracting features from high dimensional raw input.

Our deep learning approach is based on the *Robust Software Modeling Tool* (RSMT) [Sun et al. 2017], which is a late-stage (*i.e.*, post-compilation) instrumentation-based toolchain that target languages designed to run on the *Java Virtual Machine* (JVM). RSMT is a general-purpose tool that extracts arbitrarily fine-grained traces of program execution

from running software, which is applied in this paper to detect intrusions at runtime by extracting call traces in web applications. Our approach applies RSMT in the following steps:

1. During an unsupervised training epoch, traces generated by test suites are used to learn a model of correct program execution with a stacked denoising autoencoder, which is a symmetric deep neural network trained to have target value equal to a given input value [Vincent et al. 2010].

2. A small amount of labeled data is then used to calculate reconstruction error and establish a threshold to distinguish normal and abnormal behaviors.

3. During a subsequent validation epoch, traces extracted from a live application are classified using previously learned models to determine whether each trace is indicative of normal or abnormal behavior.

A key contribution of this paper is the integration of autonomic runtime behavior monitoring and characterization of web applications with end-to-end deep learning mechanisms, which generate high-level output directly from raw feature input.

The remainder of this paper is organized as follows: Section 2 summarizes the key research challenges we are addressing in our work; Section 3 describes the structure and functionality of the *Robust Software Modeling Tool* (RSMT); Section 4 explains our approach for web attack detection using unsupervised/semi-supervised end-to-end deep learning and the stacked denoising autoencoder; Section 5 empirically evaluates the performance of our RSMT-based intrusion detection system on representative web applications; Section 6 compares our work with related web attack detection techniques; and Section 7 presents concluding remarks.

2 RESEARCH CHALLENGES

This section describes the key research challenges we address and provides cross-references to later portions of the paper that show how we applied RSMT to detect web attacks by applying end-to-end deep learning.

Challenge 1: Attacks can have significantly different characteristics. Different types of web attacks, such as SQL injection, cross site scripting, remote code execution and file inclusion vulnerabilities, use different forms of attack vector and exploit different vulnerabilities inside web applications. These attacks therefore often exhibit completely different characteristics. For example, SQL injection targets databases, whereas remote code execution targets file systems. Conventional intrusion detection systems [Fu et al. 2007; Wassermann and Su 2008], however, are often designed to detect only one type of attack. For instance, a grammar-based analysis that works on SQL injection detection will not work on XSS. Section 3 describes how we applied RSMT to characterize the normal behaviors and detect different types of attacks comprehensively.

Challenge 2: Monitoring can have a significant performance cost. Static analysis approaches that analyze source code and search for potential flaws incur various drawbacks, including vulnerability to unknown attacks and the need for source code access. An alternative is to apply dynamic analysis by instrumenting applications. Instrumentation invariably incurs monitoring overhead [Waddington et al. 2009], however, which may degrade web application throughput and latency, as described in Section 5.3. Section 3.2 explores techniques applied by RSMT to minimize the overhead of monitoring and characterizing application runtime behavior.

Challenge 3: Collecting labeled attack training data. Machine learning-based intrusion detection systems rely on labeled training data to learn what should be considered normal and abnormal behaviors. Collecting this labeled training data can be hard and expensive in large-scale production web applications since labeling data requires extensive human effort and it is hard to cover all the possible cases. For example, normal request training data can be generated

with load testing tools, web crawlers, or unit tests. If the application has vulnerabilities, however, the generated data may also contain some abnormal requests, which can undermine the performance of supervised learning approaches.

Abnormal training data is even harder to obtain [Chandola et al. 2009], *e.g.*, it is hard to know what types of vulnerabilities a system has and what attacks it will face. Even manually creating attack requests targeted for a particular application may not cover all scenarios. Moreover, different types of attacks have different characteristics, which makes it hard for supervised learning methods to capture what attack requests should look like. Although supervised learning approaches often distinguish known attacks effectively, they may miss new attacks and vulnerabilities that emerge continuously, especially when web applications frequently depend on many third-party packages [equifax2 2018]. Section 4.3 describes how we applied an autoencoder-based unsupervised learning approach in RSMT to resolve the labeled training data problem.

Challenge 4: Developing intrusion detection systems without requiring users to have extensive web security domain knowledge. Traditional intrusion detection systems apply rule-based approach where users must have domain-specific knowledge in web security. Experienced security experts are thus needed to determine what feature(s) are relevant to extract from network packages, binaries, or other input for intrusion detection systems. This feature selection process can be tedious, error-prone, and time-consuming, such that even experienced engineers often rely on repetitive trial-and-error processes. Moreover, even web security experts may struggle to keep pace with the latest vulnerabilities due to quick technology refresh cycles and the continuous release of new tools and packages. Section 4.1 and 4.2 describe how we applied RSMT to build intrusion detection systems with "featureless" approaches that eliminated the feature engineering step and directly used high-dimensional request traces data as input.

3 THE STRUCTURE AND FUNCTIONALITY OF THE ROBUST SOFTWARE MODELING TOOL (RSMT)

This section describes the structure and functionality of the *Robust Software Modeling Tool* (RSMT), which we developed to autonomically monitor and characterize the runtime behavior of web applications, as shown in Figure 1. This section

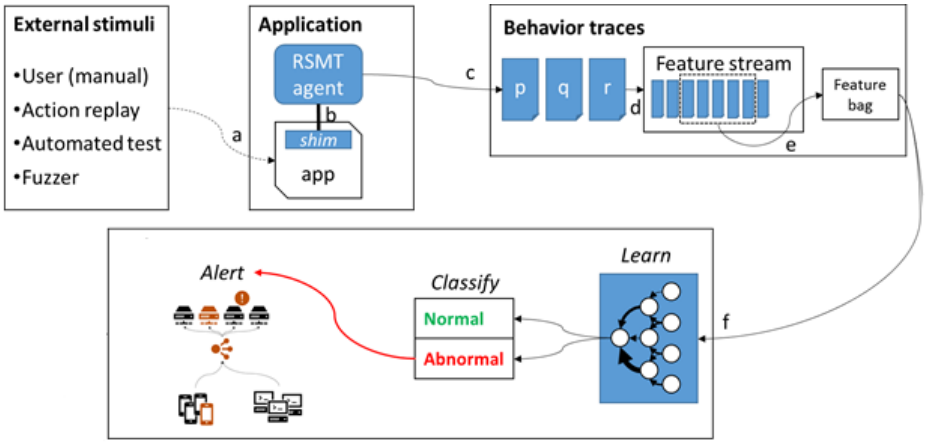

Fig. 1. The Workflow and Architecture of RSMT's Online Monitoring and Detection System

first gives an overview of RSMT, then focuses on RSMT's agent and agent server components, and finally explains how these components address *Challenge 1* detection different types of attacks) and *Challenge 2* (minimizing instrumentation

overhead) summarized in Section 2. Section 4 later describes RSMT's learning backend components and examines the challenges from Section 2 that they address.

3.1 Overview of RSMT

As discussed in Section 2, different attacks have different characteristics and traditional feature engineering approaches lack a unified solution for all types of attacks. RSMT bypasses these attack vectors and instead captures the low-level call graph. It assumes that no matter what the attack type is (1) some methods in the server that should not be accessed are invoked and/or (2) the access pattern is statistically different than the legitimate traffic.

RSMT operates as a late-stage (post-compilation) instrumentation-based toolchain targeting languages that run on the *Java Virtual Machine* (JVM). It extracts arbitrarily fine-grained traces of program execution from running software and constructs its models of behavior by first injecting lightweight shim instructions directly into an application binary or bytecode. These shim instructions enable the RSMT runtime to extract features representative of control and data flow from a program as it executes, but do not otherwise affect application functionality.

Figure 1 shows the high-level workflow of RSMT's web attack monitoring and detection system. This system is driven by one or more environmental stimuli **(a)**, which are actions transcending process boundaries that can be broadly categorized as either manual (*e.g.*, human interaction-driven) or automated (*e.g.*, test suites and fuzzers) inputs. The manifestation of one or more stimuli results in the execution of various application behaviors. RSMT attaches an agent and embeds lightweight shims into an application **(b)**. These shims do not affect the functionality of the software, but instead serve as probes that allow efficient examination of the inner workings of software applications. The events tracked by RSMT are typically control flow-oriented, though dataflow-based analysis is also possible.

As the stimuli drive the system, the RSMT agent intercepts event notifications issued by shim instructions. These notifications are used to construct traces of behavior that are subsequently transmitted to a separate trace management process **(c)**. This process aggregates traces over a sliding window of time **(d)** and converts these traces into "bags" of features **(e)**. RSMT uses feature bags to enact online strategies **(f)**, which involve the following two epochs:

- *during a training epoch*, where RSMT uses the traces generated by test suites to learn a model of correct program execution, and
- *during a subsequent validation epoch*, where RSMT classifies traces extracted from a live application using previously learned models to determine whether each trace is indicative of normal or abnormal behavior.

Figure 1 also shows the three core components of RSMT's architecture, which include (1) an *application*, to which the RSMT agent is attached, (2) an *agent server*, which is responsible for managing data gathered from various agents, and (3) a *machine learning backend*, which is used to train various machine learning models and validating traces. This architecture is scalable to accommodate arbitrarily large and complex applications, as shown in Figure 2. For example, a large web application may contain multiple components, where each component can be attached with a different agent. When the number of agents increases, a single agent server may be overwhelmed by requests from agents. Multiple agent servers can therefore be added and agent requests can then be directed to different agent servers using various partitioning rules.

It is also possible to scale the machine learning backend, *e.g.*, by deploying machine learning training and testing engine on multiple servers. An application generally comprises multiple tasks. For example, the tasks in a web forum service might be *init*, *registerNewUser*, *createThread*, and *createPost*. Machine learning models are built at the task granularity. Different machine learning backends store and process different models.

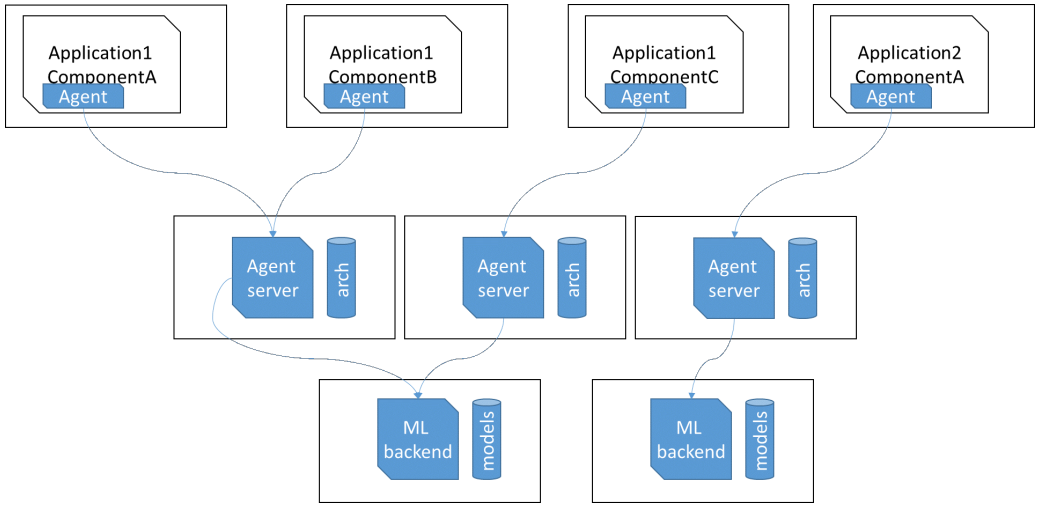

Fig. 2. The Scalability of RSMT

3.2 The RSMT Agent

Problem to resolve. To monitor and characterize web application runtime behavior, a plugin program is needed to instrument the web application and record necessary runtime information. This plugin program should require minimum human intervention to avoid burdening developers with low-level application behavior details. Likewise, instrumentation invariably incurs performance overhead that should be minimized to avoid unduly degrading web application throughput and latency.

Solution approach. To address the problem of instrumentation with minimum developer burden and performance overhead, the RSMT agent captures features that are representative of application behavior. This agent defines a class transformation system that creates events to generalize and characterize program behavior at runtime. This transformation system is plugin-based and thus extensible, *e.g.*, it includes a range of transformation plugins providing instrumentation support for extracting timing, coarse-grained (method) control flow, fine-grained (branch) control flow, exception flow, and annotation-driven information capture.

For example, a profiling transformer can inject ultra-lightweight instructions to store the timestamps when methods are invoked. A trace transformer could add **methodEnter()** and **methodExit()** calls to construct a control flow model. Each transformation plugin conforms to a common API. This common API can be used to determine whether the plugin can transform a given class, whether it can transform individual methods in that class, and whether it should actually perform those transformations if it is able.

We leverage RSMT's publish-subscribe (pub/sub) framework to (1) rapidly disseminate events by instrumented code and (2) subsequently capture these events via event listeners that can be registered dynamically at runtime. RSMT's pub-sub framework is exposed to instrumented bytecode via a proxy class that contains various static methods.[1] In turn, this proxy class calls various listeners that have been registered with it. The following event types are routed to event listeners:

• *Registration events* are typically executed once per method in each class as its $< clinit >$ (class initializer) method is executed. These events are typically consumed (not propagated) by the listener proxy.

[1]We use static methods since calling a Java static method is up to 2x faster than calling a Java instance method.

• *Control flow events* are issued just before or just after a program encounters various control flow structures. These events typically propagate through the entire listener delegation tree.

• *Annotation-driven events* are issued when annotated methods are executed. These events propagate to the offline event processing listener children.

The root listener proxy is called directly from instrumented bytecode and delegates event notifications to an error handler, which gracefully processes exceptions generated by child nodes. Specifically, the error handler ensures that all child nodes receive a notification regardless of whether that notification results in the generation of an exception (as is the case when a model validator detects unsafe behavior). The error handler delegates to the following model construction/validation subtrees:

- the *online model construction/validation subtree* performs model construction and verification in the current thread of execution *i.e.*, on the critical path, and
- the*offline model construction/validation subtree* converts events into a form that can be stored asynchronously with a (possibly remote) instance of Elasticsearch [elasticsearch 2018], which is an open-source search and analytics engine that provides a distributed real-time document store.

To address *Challenge 1* (minimizing the overhead of monitoring and charactering application runtime behavior) described in Section 2, RSMT provides a dynamic filtering mechanism. We analyzed the method call patterns and observed that most method calls are lightweight and occur in a small subset of nodes in the call graph. By identifying a method as being called frequently and having a significantly larger performance impact, we can disable events issued from it entirely or reduce the number of events it produces (thereby improving performance). These observations, along with a desire for improved performance, motivated the design of RSMT's dynamic filtering mechanism.

To enable filtering, each method in each class is associated with a new static field added to that class during the instrumentation process. The value of the field is an object used to filter methods before they make calls to the runtime trace API. This field is initialized in the constructor and is checked just before any event would normally be issued to determine if the event should actually occur.

To characterize feature vector abilities to reflect application behaviors, we added an online model builder and model validator to RSMT. The model builder constructs two views of software behavior: a *call graph*, which is used to quickly determine whether a transition is valid, and a *call tree*, which is used to determine whether a sequence of transitions is valid. The model validator is a closely related component that compares current system behavior to an instance of a model assumed to represent correct behavior.

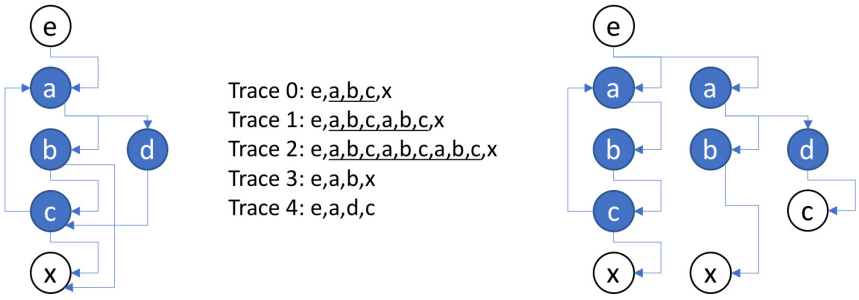

Fig. 3. Call Graph (L) and Call Tree (R) Constructed for a Simple Series of Call Stack Traces

Figures 4 and 5 demonstrate the complexity of the graphs we have created by applying RSMT on various SQL statements.

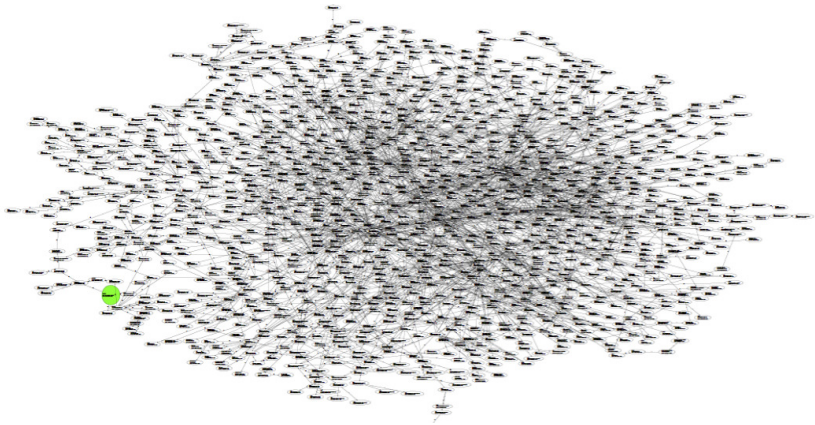

Fig. 4. Call Tree Generated for a Simple SQL Statement Parse

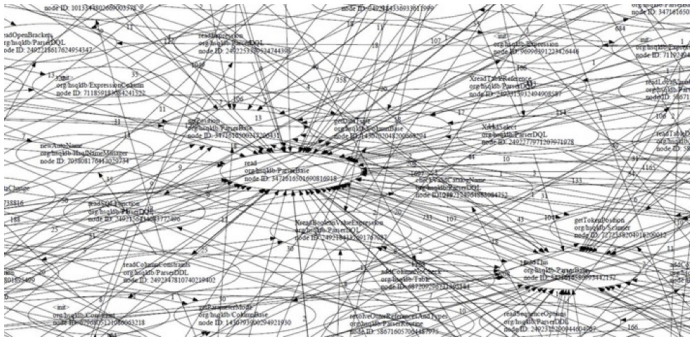

Fig. 5. Call Tree Generated for a Simple SQL Statement Parse (zoomed in on heavily visited nodes)

Each directed edge in a call graph connects a parent method (source) to a method called by the parent (destination). Call graph edges are not restricted with respect to forming cycles. Suppose the graph in Figure 3 represented correct behavior. If we observed a call sequence *e,a,x* at runtime, we could easily tell this was not a valid execution path since no *a,x* edge is present in the call graph.

Although the call graph is fast and simple to construct, it has shortcomings. For example, suppose a transition sequence *e,a,d,c,a* is observed. Using the call graph, none of these transition edges violated expected behavior. If we account for past behavior, however, there is no *c,a* transition occurring after *e,a,d*. To handle these complex cases, a more robust structure is needed. This structure is known as the *call tree*, as shown in the right-hand side of Figure 3.

Whereas the call graph falsely represents it as a valid sequence, there is no path along sequence *e,a,d,c,a* in the call tree (this requires two backtracking operations), so we determine that this behavior is incorrect. The call tree is not a

tree in the structural sense. Instead, it is a tree where each branch represents a possible execution path. If we follow the current execution trace to any node in the call tree, the current behavior matches the expectation.

Unlike a pure tree, the call tree does have self-referential edges (*e.g.*, the *c,a* edge in Figure 3) if recursion is observed. Using this structure is obviously more processing-intensive than tracking behavior using a call graph. Section 5.3 presents empirical evaluation of the performance overhead of the RSMT agent.

3.3 The RSMT Agent Server

Problem to resolve. A web application may comprise multiple components where multiple agents are attached. Likewise, multiple instances of the application may run on different physical hardware for scalability. It is important for agents to communicate effectively with our machine learning backend to process collected traces, which requires some means of mapping the task- and application-level abstractions onto physical computing resources.

Solution approach. RSMT defines an agent server component to address the problem of mapping task/application-level abstractions to physical computing resources. This component receives traces from various agents, aligns them to an application architecture, maps application components to models of behavior, and pushes the trace to the correct model in a remote machine learning system that is architecture agnostic. The agent server exposes three different REST APIs, which are described below:

- **A trace API** that RSMT agents use to transmit execution traces. This API allows an agent to (1) register a recently launched JVM as a component in a previously defined architecture and (2) push execution trace(s).
- **An application management API** for defining and maintaining applications by (1) defining/deleting/modifying an application, (2) retrieving a list of applications, and (3) transitioning components in an application from one state to another. This design affects how traces received from monitoring agents are handled, *e.g.*, in the *IDLE* state, traces are discarded whereas in the *TRAIN* state they are conveyed to a machine learning backend that applies them incrementally to build a model of expected behavior. In the *VALIDATE* state, traces are compared against existing models and classified as normal or abnormal.
- **A classification API** that monitors the health of applications. This API can be used to query the status of application components over a sliding window of time, whose width determines how far back in time traces are retrieved during the health check and which rolls up into a summary of all classified traces for an applicationâĂŹs operation. This API can also be used to retrieve a JSON representation of the current health of an application.

4 UNSUPERVISED WEB ATTACK DETECTION WITH END-TO-END DEEP LEARNING

This section describes how our unsupervised/semi-supervised web attack detection system augments the RSMT architectural components described in Section 3 with *end-to-end deep learning* mechanisms [Amodei et al. 2016; Graves and Jaitly 2014], which generate high-level output directly from raw feature input. The RSMT components covered in Section 3 provide feature input for the end-to-end deep learning mechanisms described in this section, whose output indicates whether a given web request is legitimate or an attack. This capability addresses *Challenge 4* (developing intrusion detection systems without domain knowledge) summarized in Section 2.

4.1 Traces Collection with Unit Tests

The RSMT agent is responsible for collecting application runtime traces, as described in Section 3.2. These collected traces include the program's execution path information, which is then used as the feature input for our end-to-end deep learning system. Below we discuss how the raw input data is represented.

When a client sends a request to a web application the RSMT agent records a *trace*, which is a histogram of directed f-calls-g edges observed beginning after the execution of a method. In particular, from a starting entry method A, we record call traces up to depth d. We record the number of times each trace triggers each method to fulfill a request from a client.

For example, A calls B one time and A calls B and B calls C one time will be represented as: A-B: 2; B-C: 1; A-B-C: 1. Each trace can be represented as a 1*N vector [2,1,1] where N is the number of different method call sequences. Unlike sequence-base approaches, we do not capture every order of method call, but instead use the frequency count as features. The order information, however, is still partially preserved by recording the frequency of call sequences.

We also pad the 1*N histogram feature with an additional dimension to represent un-seen method calls. If a test dataset contains method calls that never appear in the training dataset, its count will be recorded in this bit. Our goal is to determine if the request is an attack request when given the trace signature $T_i = \{c_1, c_2, ..., c_n\}$ produced in response to a client request P_i.

4.2 Anomaly Detection with Deep Learning

Machine learning approaches for detecting web attacks can be categorized into the following two types

- **Supervised learning** approaches (such as Naive Bayes [Russell et al. 1995] and SVM [Cortes and Vapnik 1995]) work by calibrating a classifier with a training dataset that consists of data labeled as either normal traffic or attack traffic. The classifier then classifies the incoming traffic as either normal data or an attack request. Two general types of problems arise when applying supervised approaches to detect web attacks: (1) classifiers cannot handle new types of attacks that are not included in the training dataset, as described in *Challenge 3* (hard to obtain labeled training data) in Section 2 and (2) it is hard to get a large amount of labeled training data, as described in *Challenge 3* in Section 2.
- **Unsupervised learning** approaches (such as Principal Component Analysis (PCA) [Wold et al. 1987] and autoencoder [Vincent et al. 2010]) do not require labeled training datasets. Instead, they rely on the assumption that data can be embedded into a lower dimensional subspace in which normal instances and anomalies appear significantly different. The idea is to apply dimension reduction techniques (such as PCA or autoencoders) for anomaly detection. PCA or autoencoders try to learn a function $h(X) = X$ that maps input to itself.

The input traces to web attack detection can have a very high dimension (thousands or more). If no constraint is enforced, an identity function will be learned, which is not useful. We therefore force some information loss during the process. For example, in PCA we only select a subset of eigenvalues. In autoencoder, the hidden layers will have smaller dimension than the input.

For PCA, the original input X will be projected to $Z = XV$. V contains the eigenvectors and we can choose k eigenvectors with the largest eigenvalues. To reconstruct the original input, $x = XVV^T$. If all eigenvectors are used, then VV^T is an identity matrix, no dimensionality reduction is performed, the reconstruction is perfect. If only a subset of eigenvectors are used, the reconstruction is not perfect, the reconstruction error is given by $E = ||x - X||^2$.

If a test input shares similar structure or characteristics with training data, the reconstruction error should be small. To apply the same principle to web attack detection, if a test trace is similar to the ones in the training set, the reconstruction error should be small and it is likely to be a legitimate request. If the reconstruction error is large, it implies the trace is statistically different, thereby suggesting it has a higher probability of being an attack request.

4.3 End-to-end Deep Learning with Stacked Denoising Autoencoders

The transformation performed by PCA is linear, so it cannot capture the true underlying input and output relationships if the modeled relationship is non-linear. *Deep neural networks* (DNNs) [LeCun et al. 2015] have achieved success in computer vision, speech recognition, natural language processing, etc. With non-linear activation functions and multiple hidden layers, DNNs can model complex non-linear functions.

The decision functions for anomaly detection in web attacks are often complex since no simple threshold can be used to determine if the request is an attack. Complicated interactions, such as co-occurrence and order of method calls, are all involved in the decision making. These complexities make DNNs ideal candidates for anomaly detection in web attacks. In particular, we use a special case of neural network called an *autoencoder* [Vincent et al. 2010], which is a neural network with a symmetric structure.

An autoencoder consists of two parts: (1) an encoder that maps the original input to a hidden layer h with an encoder function $h = f(x) = s(Wx + b)$, where s is the activation function and (2) a decoder that produce a reconstruction $r = g(h)$. The goal of normal neural networks is to learn a function $h(x) = y$ where the target variable y can be used for classification or regression. An autoencoder is trained to have target value equal to input value, *i.e.*, to minimize the difference between target value and input value, *e.g.*, $L(x, g(f(x))$ where L is the loss function. In this case, the autoencoder penalizes $g(f(x))$ for being dissimilar from x.

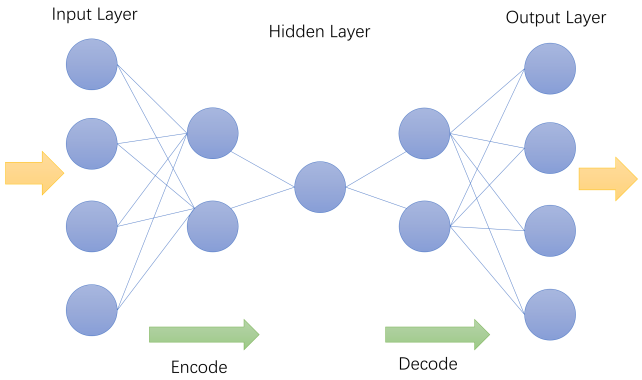

Fig. 6. Structure of Stacked Autoencoder.

If no constraint is enforced, an autoencoder will likely learn an identity function by just copying the input to the output, which is not useful. The hidden layers in autoencoders are therefore usually constrained to have smaller dimensions than the input x. This dimensionality constraint forces autoencoders to capture the underlying structure of the training data.

Figure 7 shows a visualization of normal and abnormal requests using the compressed representation learned from an autoencoder via a t-Distributed Stochastic Neighbor Embedding (t-SNE) [Maaten and Hinton 2008]. Blue dots in this figure represent normal requests and red dots represent abnormal requests, which can thus be easily distinguished in the low-dimensional subspace learned with the autoencoder.

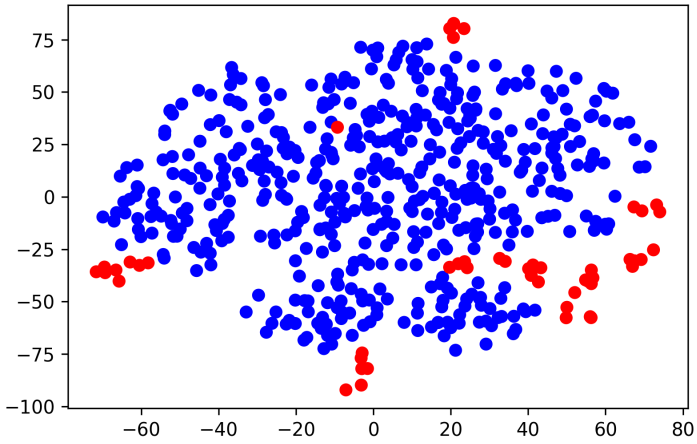

Fig. 7. t-SNE Visualization of Normal and Abnormal Requests.

To address *Challenge 2* (detecting different types of attacks) described in Section 2, the autoencoder performs feature extraction automatically. The input x is mapped to a low dimensional representation and reconstructed trying to restore input. When the reconstruction g(f(x)) is different from x, the reconstruction error $e = ||g(f(x) - x)||^2$ can be used as an indicator for abnormality.

If the training data share similar structure or characteristics, the reconstruction error should be small. An outlier is a datum that has significantly different underlying structure or characteristic. It is therefore hard to represent the outlier with the feature we extract. As a result, the reconstruction error will be larger. We can use the reconstruction error as a standard to distinguish abnormal traffic and legitimate traffic.

Compared to PCA, autoencoders are more powerful because the encoder and decoder functions can be chosen to be non-linear, thereby capturing non-linear manifolds. In contrast, PCA just does linear transformations, so it can only create linear decision boundaries, which may not work for complex attack detection problems. Moreover, non-linearity allows the network to stack to multiple layers, which increases the modeling capacity of the network. While the combination of multiple linear transformation is still one linear layer deep, it may lack sufficient capacity to model the attack detection decision.

Challenge 4 (developing intrusion detection systems without domain knowledge) in Section 2 is also addressed by applying the following two extensions to conventional autoencoders:

1. Stacked autoencoders, which may contain more than one hidden layer [Vincent et al. 2010]. Stacking increases the expressing capacity of the model, which enables the autoencoders to differentiate attacks and legitimate traffic from high dimensional input without web security domain knowledge. The output of each preceding layer is fed as the input to the successive layer. For the encoder: $h_1 = f(x)$, $h_i = f(h_{i-1})$, whereas for the decoder: $g_1 = g(h_i)$, $g_i = g(g_{i-1})$. Deep neural networks have shown promising applications in a variety of fields such as computer vision, natural language processing due to its representation power. These advantages also apply to deep autoencoders.

To train our stacked autoencoder we use a pretraining step involving greedy layer-wise training. The first layer of encoder is trained on raw input. After a set of parameters are obtained, this layer is used to transform the raw input to

a vector represented as the hidden units in the first layer. We then train the second layer on this vector to obtain the parameters of second layers. This process is repeated by training the parameters of each layer individually, while keep the parameters of other layers unchanged.

2. Denoising, which prevents the autoencoder from over-fitting. Our system must be able to generalize to cases that are not presented in the training set, rather than only memorizing the training data. Otherwise, our system would not work for unknown or new types of attacks. Denoising works by corrupting the original input with some form of noise. The autoencoder now needs to reconstruct the input from a corrupted version of it, which forces the hidden layer to capture the statistical dependencies between the inputs. More detailed explanation of why denoising autoencoder works can be found in [Vincent et al. 2008]. In our experiment (outlined here and described further in Section 5) we implemented the corruption process by randomly setting 20% of the entries for each input to 0.

We chose a denoising autoencoder with three hidden layers for our experiments in Section 5. The structure of the autoencoder is shown in Figure 6. The hidden layer contains n/2, n/4, n/2 dimensions respectively. Adding more hidden layers does not improve the performance and can easily overfit. Relu [Nair and Hinton 2010] was chosen as the non-linear activation function in the hidden layer. Section 5.5 presents the results of experiments that evaluate the performance of a stacked denoising autoencoder in web attack detection.

The architecture of our unsupervised/semi-supervised web attack detection system is shown in Figure 8 and described below (each numbered bullet corresponds to a numbered portion of the figure):

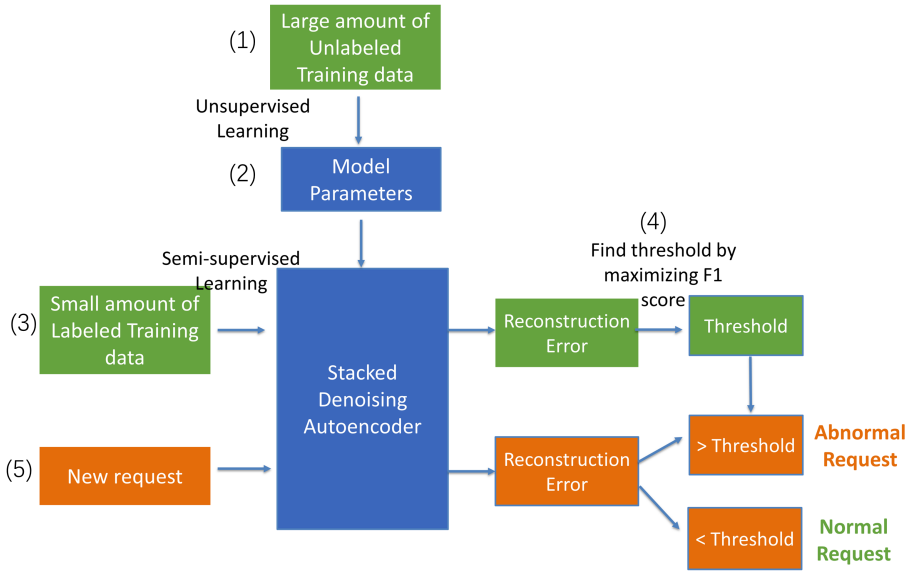

Fig. 8. The Architecture of the Unsupervised/Semi-supervised Web Attack Detection System.

1. RSMT collected a large number of unlabeled training traces by simulating normal user requests. These unlabeled training traces should contain mostly normal requests, although a few abnormal requests may slip in.

2. A stacked denoising autoencoder is used to train on the unlabeled training traces. By minimizing the reconstruction error, the autoencoder learns an embedded low dimensional subspace that can represent the normal requests with low reconstruction error.

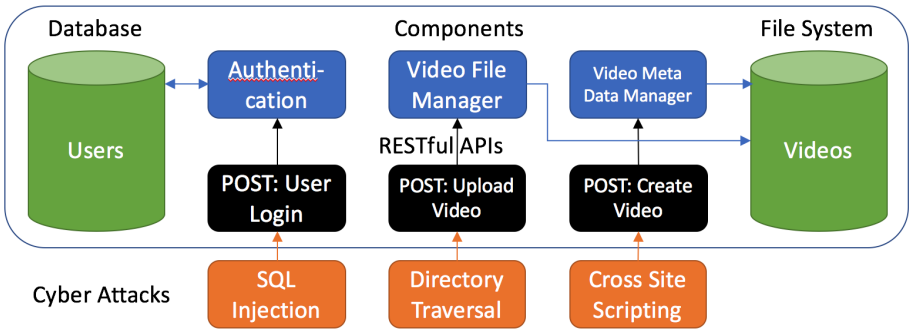

Fig. 9. Cyber-attacks Exploited in the Test Video Management Application

3. A semi-supervised learning step can optionally be performed, where a small amount of labeled normal and abnormal request data is collected. Normal request data can be collected by running repetitive unit tests or web traffic simulators, such as Apache JMeter [jmeter 2018]. Abnormal request data can be collected by manually creating attack requests, such as SQL injection and Cross-Site Scripting (XSS) attacks against the system. The transformation learned in unsupervised learning is applied to both normal and abnormal requests and their average reconstruction error is calculated respectively. A threshold for reconstruction error is chosen to maximize a metric, such as the F1 score, which measures the harmonic average of the precision and recall.

4. If no semi-supervised learning is conducted, the highest reconstruction error for unlabeled training data is recorded and the threshold is set to a value that is higher than this maximum by a adjustable percentage.

5. When a new test request arrived, the trained autoencoder will encode and decode the request vector and calculate reconstruction error E. If E is larger than the learned threshold θ, it will be classified as attack request. If E is smaller than θ, it will be considered as a normal request.

5 ANALYSIS OF EXPERIMENTAL RESULTS

This section presents the results of experiments that empirically evaluate our deep learning-based intrusion detection system. We first describe the test environment and evaluation metrics. We then compare the performance of our end-to-end deep learning approach with alternative methods.

5.1 Testbed

We used the following two web applications as the basis for the testbed in our experiments: (1) a **video management application** built on Apache Spring framework using an embedded HSQL database and which handles HTTP requests for uploading, downloading, and viewing video files, and (2) a **compression service application** built upon the Apache Commons Compress library and which takes a file as input and outputs a compressed file in the chosen compression format. Figure 9 shows how the test video management application provides several RESTful APIs, including: (1) *user authentication*, where a GET API allows clients to send usernames and passwords to the server and then checks the SQL database in the back-end for authentication, (2) *video creation*, where a POST API allows clients to create or modify video metadata, and (3) *video uploading/downloading*, where POST/GET APIs allow users to upload or download videos from the server's back-end file system using the video IDs.

Our test web applications (webapps) were engineered in a manner that intentionally left them susceptible to several widely-exploited vulnerabilities. The test emulated the behavior of both normal (good) and abnormal (malicious) clients by issuing service requests directly to the test webapp's REST API. For example, the test harness might register a user

with the name "Alice" to emulate a good client's behavior or "Alice OR True" to emulate a malicious client attempting a SQL injection attack.

To evaluate the system's attack detection performance, we exploited three attacks from OWASP's top ten cybersecurity vulnerabilities list [owasp2013 2018a] and used them against the test webapp. These attacks included SQL injection, Cross-Site Scripting (XSS), and object deserialization vulnerabilities, as described below.

SQL injection. The SQL injection attack was constructed by creating queries with permutations/combinations of keywords INSERT, UPDATE, DELETE, UNION, WHERE, AND, OR, etc. The following types of SQL injections were examined:

- **Type1: Tautology-based**. Statements like OR '1' = '1' and OR '1' < '2' were added at the end of the query to make the preceding statement always true. For example, SELECT * FROM user WHERE username = 'user1' OR '1' = '1'.
- **Type2: Comment-based**. A comment was used to ignore the succeeding statements, *e.g.*, SELECT * FROM user WHERE username = 'user1' AND password = '123'.
- **Type3: Use semicolon to add additional statement**, *e.g.*, SELECT * FROM user WHERE username = 'user1'; DROP TABLE users; AND password = '123'.

Cross-Site Scripting (XSS). For the XSS attack, we added a new method with a @RequestMapping[2] in a controller that was never called in the "normal" set. We then called this method in the abnormal set to simulate an XSS attack that accessed code blocks a client should not be able to access. We also modified an existing controller method with @RequestMapping so a special value of one request path called a completely different code path to execute. This alternate code path was triggered only in the abnormal set.

Object deserialization. Object deserialization vulnerabilities [owasp2013 2018b] can be exploited by crafting serialized objects that will invoke reflective methods that result in unsafe behaviors during the deserialization process. For example, we could store ReflectionTransformer items in an ArrayList that result in Runtime.exec being reflectively invoked with arguments of our choice (effectively enabling us to execute arbitrary commands at the privilege level of the JVM process). To generate such serialized objects targeting the Commons-Collections library, we used the ysoserial tool [ysoserial [n. d.]].

We collected 1,000 traces for the compression service application. All runs compressed 64 MB of randomly generated data using a different method of random data generation for each run. For each of $x \in \{1, 2, 4, 8, 16, 32, ..., 1024, 2048, 4096\}$, a single chunk of size 64 MB/x was generated and duplicated times (with x = 4096 the data is repetitive, whereas with x = 1, the data is not repetitive at all). This test shows the input dependency of compression algorithm control flow, so it was not feasible to create inputs/test cases that would exercise all possible control flow paths.

5.2 Evaluation Metrics

An ideal intrusion detection system should classify the legitimate traffic as normal and classify attack traffic as abnormal. Two types of errors therefore exist: (1) **A false positive (FP)** or false alarm, which refers to classifying benign traffic as an attack, and (2) **A false negative (FN)**, which refers to classifying attack traffic as benign traffic. A key goal of an intrusion detection system is to minimize both the FP rate and FN rate. A tradeoff exists, however, since a more strict algorithm will tend to reduce the FN rate at the cost of classifying benign traffic as attack traffic.

Anomaly detection is an imbalanced classification problem, *i.e.*, the attack test cases appear much less frequently than the normal test cases. Accuracy is therefore not a good metric because simply predicting every request as normal will

[2] @RequestMapping is an annotation used in Spring framework for mapping web requests onto specific handler classes or handler methods.

give very high accuracy. To address this issue, we use the following metrics to evaluate our approaches: (1) **Precision = TP/(TP+FP)**, which penalizes false positives, (2) **Recall = TP/(TP+FN)**, which penalizes false negatives, and (3) **F1 score = 2*precision*recall/(precision+recall)**, which evenly weights precision and recall.

5.3 Overhead Observations

To examine the performance overhead of the RSMT agent described in Section 3, we conducted experiments that evaluated the runtime overhead in average cases and worst cases, as well as assessed how "real-time" application execution monitoring and abnormal detection could be. As discussed in Section 3, RSMT modifies bytecode and subsequently executes it, which incurs two key sources of overhead: (1) the cost of the instrumentation itself and (2) the performance cost of executing the new instructions injected into the original bytecode.

Such instruction-level tracing can significantly increase execution time in the worst case. For example, consider a while loop that iterates 100,000 times and contains 5 instructions. If a visitInstruction() method call is added to each static instruction in the loop, roughly 500,000 dynamic invocations of the visitInstruction() method will be incurred, which is a two-fold increase in the number of dynamic instructions encountered. Moreover, this overhead can be even greater when considering the number of instructions needed to initialize fields and make the appropriate calls to visitMethodEnter() or handle exceptions.

RSMT has low overhead for non-computationally constrained applications. For example, a Tomcat web server that starts up in 10 seconds takes roughly 20 seconds to start up with RSMT enabled. This startup delay is introduced since RSMT examines and instruments every class loaded by the JVM. This startup cost typically occurs only once, however, since class loading usually happens just once per class.

In addition to startup delays, RSMT incurs runtime overhead every time instrumented code is invoked. We tested several web services and found RSMT had an overhead ranging from 5% to 20%. The factors most strongly impacting its overhead are the number of methods called (more frequent invocation results in higher overhead) and the ratio of computation to communication (more computation per invocation results in lower overhead).

To evaluate worst-case performance, we used RSMT to monitor the execution of an application that uses ApacheâĂŹs Commons Compress library to "bz2 compress" randomly-generated files of varying sizes ranging from 1x64 byte blocks to 1024x64 byte blocks, which is a control-flow intensive task. Moreover, the Apache Commons implementation of bz2 is "method heavy," (*i6e.*., there are a significant number of setter and getter calls), which are typically optimized by the JVMâĂŹs hotspot compiler and converted into direct variable accesses. The instrumentation performed by RSMT prevents this optimization from occurring, however, since these lightweight methods are wrapped in calls to the model construction and validation logic. As a result, our bz2 benchmark represents the worst case for RSMT performance.

Figure 10 shows that registration adds a negligible overhead to performance (0.5 to 1%), which is expected since registration events only ever occur once per class, at class initialization. Adding call graph tracking incurs a significant performance penalty, particularly as the number of randomly generated blocks increases. Call graph tracking ranges from 1.5x to over 10x slower than the original application, whereas call tree tracking results in a 2-5x slowdown. Similarly, fine-grained control flow tracking results in a 4-6x slowdown. With full fine-grained tracking enabled, therefore, an application might run at 1% its original speed. By filtering getters and setters, however, it is possible to reduce this overhead by several orders of magnitude, as described later.

To further quantify RSMT's performance overhead, we used SPECjvm2008 [spec2008 2018], which is a suite comprising various integer and floating point benchmarks that quantitatively compare the performance of JVM implementations (*e.g.*, to determine whether one implementationâĂŹs JIT compiler is superior to another for a certain type of workload).

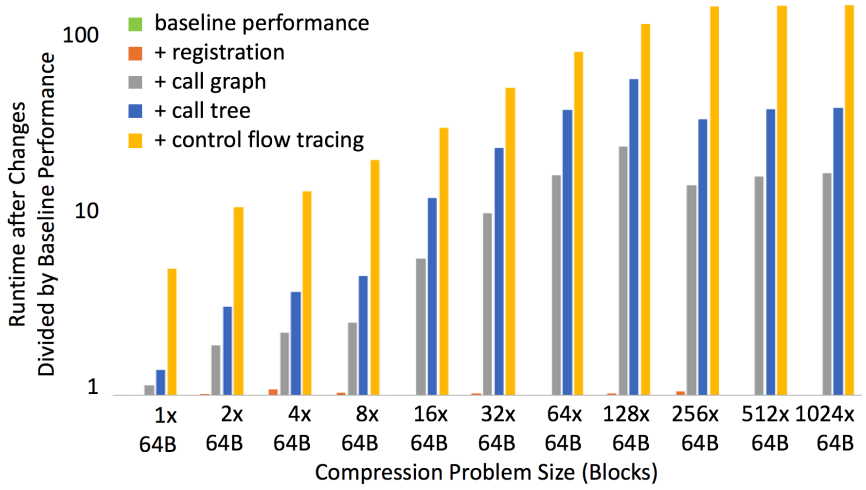

Fig. 10. Analysis of RSMT Performance Overhead

We used the same JVM implementation across our tests, but varied the configuration of our instrumentation agents to measure the performance tradeoffs.

We evaluated the following configurations: (1) no instrumentation (no RSMT features emitted), (2) reachability instrumentation only (disabled after first access to a code region), (3) call tracing but all events passed into a null implementation, and (4) reachability + call tracing (null). We executed each configuration on a virtualized Ubuntu 14 instance provisioned with two cores and 8 GB of memory. The results of this experiment are shown below in Figure 11. We would expect a properly tuned RSMT system to perform somewhere between configurations 3 and 4.

Although we observed that the overhead incurred by naively instrumenting all control flows within an application could be quite large (see Figure 10), a well-configured agent should extract useful traces with overheads ranging from nearly 0% (for computation-bound applications) to 40% (for control-bound applications). Most production applications contain a blend of control-bound and computation-bound regions. Under this assumption we anticipate an overhead of 15-20% based on the composite score impact shown in Figure 11.

5.4 Supervised Attack Detection with Manually Extracted Features

Before evaluating the performance of our deep learning approach, we present several supervised learning methods as benchmarks for comparison. We also describe the manually extracted features we used.

5.4.1 Experiment Benchmarks. Datasets and feature vectors are crucial for cyber-attack detection systems. The following feature attributes were chosen as the input for our supervised learning algorithms:

(1) *Method execution time.* Attack behaviors can result in abnormal method execution times, *e.g.*, SQL injection attacks may execute faster than normal database queries.

(2) *User Principal Name (UPN).* UPN is the name of a system user in an e-mail format, such as my_name@my_domain_name. When attackers log into the test application using fake user principal names, the machine learning system can use this feature to detect it.

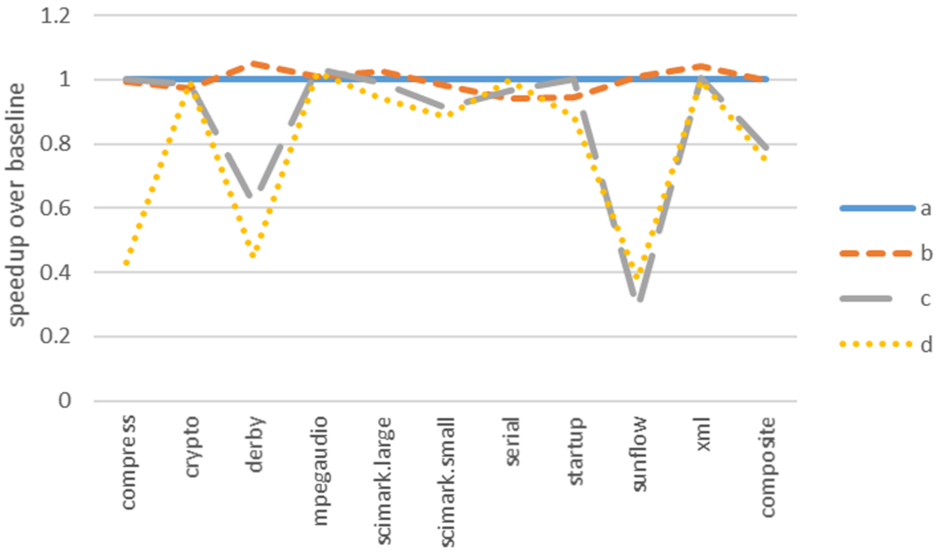

Fig. 11. SPECjvm2008 Performance Impact for Various Benchmarks and Test Configurations

(3) *The number of characters of an argument, e.g.*, XSS attacks might input some abnormally large argument lengths, such as http://www.msn.es/usuario/guias123/default.asp?sec=\discretionary{-}{}{}"></script><script>alert("Da\discretionary{-}{}{}iMon")</script>

(4) *Number of domains*, which is the number of domains found in the arguments. The arguments can be inserted with malicious URLs by attackers to redirect the client "victim" to access malicious web sources.

(5) *Duplicate special characters.* Many web browsers ignore and correct duplicated characters, so attackers can insert duplicated characters into requests to fool validators.

(6) *N-gram.* Feature vector was built using the n-gram [Powers 2011] model. The original contents of the arguments and return values are filtered by Weka's StringToWordVector tool (which converts plain word into a set of attributes representing word occurrence) and the results are then applied to make the feature vectors.

After instrumenting the runtime system to generate measurements of the system when it is executing correctly or incorrectly, supervised approaches use these measurements to build a training data set. In this data set the measurements are viewed as features that can characterize the correct and incorrect system operation. Machine learning algorithms use these features to derive models that classify the correctness of the execution state of the system based on a set of measurements of its execution. When new execution measurements are given to the machine-learned model, algorithms can be applied to predict whether the previously unseen trace represents a valid execution of the system.

To provide an environment for classification, regression, and clustering we used the following three supervised machine learning algorithms from the Weka workbench:

- *Naive Bayes*, whose classification decisions calculate the probabilities/costs for each decision and are widely used in cyber-attack detection [Buczak and Guven 2015].
- *Random forests*, which is an ensemble learning method for classification that train decision trees on sub-samples of the dataset and then improve classification accuracy via averaging. A key parameter for random forest is the number of attributes to consider in each split point, which are selected automatically by Weka.

- *Support vector machine (SVM)*, which is an efficient supervised learning model that draws an optimal hyperplane in the feature space and divides separate categories as widely as possible. RSMT uses Weka's *Sequential Minimal Optimization* algorithm to train the SVM.

Likewise, to reduce variance and avoid overfitting [Opitz and Maclin 1999], we also used the following two aggregate models:

- *Aggregate_vote*, which returns ATTACK if a *majority* of classifiers detect attacks and NOT_ATTACK otherwise.
- *Aggregate_any*, which returns attack if *any* classifier detects attacks and NOT_ATTACK otherwise.

5.4.2 Experiment Results. Table 1 and Table 2 show the performance comparison of different machine learning algorithms on testbed web applications. For the SQL injection attacks, the training dataset contains 160 safe unit tests and 80 attack unit tests, while the test dataset contains 40 safe unit tests and 20 attack unit tests. The SQL injection attack samples bypass the test applicationâĂŹs user authentication and include the most common SQL injection attack types.

Table 1. Machine Learning Models' Experimental Results for SQL Injection Attacks

	Precision	Recall	F-score
Naive bayes	0.941	0.800	0.865
Random forest	1.000	0.800	0.889
SVM	0.933	0.800	0.889
AGGREGATE_VOTE	1.000	0.800	0.889
AGGREGATE_ANY	0.941	0.800	0.865

The XSS training dataset contains 1,000 safe unit tests and 500 attack unit tests, while the test dataset contains 150 safe unit tests and 75 attack unit tests (XSS attack samples were obtained from `www.xssed.com`). All three classifiers are similar in detecting XSS attacks.

Table 2. Machine Learning Models' Experimental Results for Cross-site Scripting Attacks

	Precision	Recall	F-score
Naive bayes	0.721	1.000	0.838
Random forest	0.721	1.000	0.838
SVM	0.728	1.000	0.843
AGGREGATE_VOTE	0.724	1.000	0.840
AGGREGATE_ANY	0.710	1.000	0.831

5.5 Unsupervised Attack Detection with Deep Learning

5.5.1 Experiment benchmarks. Several techniques can be applied to differentiate benign traffic and attack traffic. The first is the naive approach, which learns a set of method calls from a training set (obtained by unit test or simulated legitimate requests). When a new trace is encountered, the naive approach checks if the trace contain any method call that is never seen from the training set. If there is such method, the trace will be treated as attack trace, otherwise it is considered safe.

The naive approach can detect attack traces easily since attack traces usually contains some dangerous method calls that will not be used in legitimate operation. The naive approach, however, also suffer from high false positive rate since it may not be possible to iterate through all the legitimate request scenarios. A legitimate request may thus contain some method call(s) that do not exist in the training set, which results in blocking benign traffic.

A more advanced technique is one-class SVM [Wang et al. 2004]. Traditional SVM solves the two or multi-class situation. While the goal of a one-class SVM is to test new data and find out whether it is similar to the training data or not. By just providing the normal training data, one-class classification creates a representational model of this data. If newly encountered data is too different (*e.g.*, outliers in the projected high-dimensional space), it is labeled as out-of-class.

5.5.2 Experiment Results. Table 3 and Table 4 compare the performance of different machine learning algorithms on our two testbed web applications. For the video upload application, the attack threat is SQL injection and XSS. The results in these tables show that autoencoder outperforms the other algorithms. For the compression application, we evaluate the detection performance in terms of a deserialization attack. Figure 12 plots the precision/recall/F-score

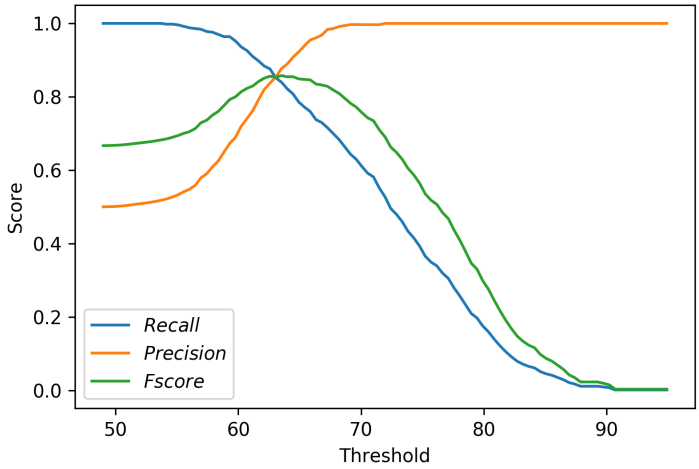

Fig. 12. Threshold is Chosen with Max F-score.

Table 3. Performance Comparison of Different Machine Learning Algorithms on Video Management Application

	Precision	Recall	F-score
Naive	0.722	0.985	0.831
PCA	0.827	0.926	0.874
One-class SVM	0.809	0.909	0.858
Autoencoder	0.898	0.942	0.914

Table 4. Performance Comparison of Different Machine Learning Algorithms on Compression Application

	Precision	Recall	F-score
Naive	0.421	1.000	0.596
PCA	0.737	0.856	0.796
One-class SVM	0.669	0.740	0.702
Autoencoder	0.906	0.928	0.918

curve along with threshold value. This figure shows a tradeoff between precision and recall. If a threshold is choosen that is too low, many normal request will be classified as abnormal, resulting in higher false negative and low recall score. In contrast, if a threshold is choosen that is too high, many abnormal requests will be classified as normal, leading to higher false positive and low precision score. To balance precision and recall in our experiements, we choose a threshold that maximizes the F-score in the labeled training data.

To understand how various parameters (such as training data size, input feature dimension, and test coverage ratio) affect the performance of machine learning algorithms, we manually created a synthetic dataset to simulate web application requests. Figure 13 compares the performance of machine learning algorithms with different unlabeled

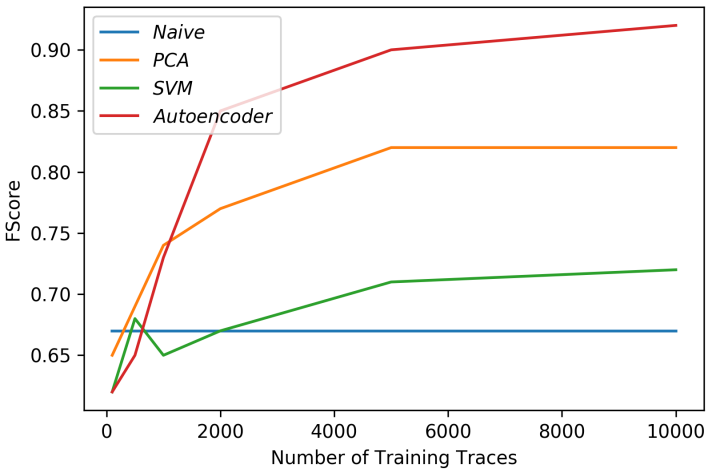

Fig. 13. Performance of Different Machine Learning Algorithm Under Different Unlabeled Training Data Size.

training data sizes. Since the test case contains method calls that were not presented in the training data, the naive approach simply treats every request as abnormal, resulting 100% recall, but 0% precision. Both PCA and autoencoder's performance improves since we have more training data.

PCA performs better, however, when there is limited training data (below 1,000). The autoencoder needs more training data to converge, but outperforms the other machine learning algorithms after it is given enough training data . Our results show the autoencoder generally needs 5,000 unlabeled training data to achieve good performance. Figure 14 shows the performance of machine learning algorithms under different test coverage ratios. The test coverage ratio is the percentage of method calls covered in the training dataset. For large-scale web applications, it is impossible to traverse every execution path and method calls due to the "path explosion problem" [Boonstoppel et al. 2008], where the number of feasible paths in a program grows exponentially with an increase in program size.

If only a subset of method calls are present in the training dataset, the naive approach or other supervised learning approaches may classify the legitimate test request with uncovered method calls as abnormal. In contrast, PCA and autoencoder algorithms can still learn a hidden manifold by finding the similarity in structure instead of exact method calls. They can thus perform well even given only a subset of coverage for all the method calls.

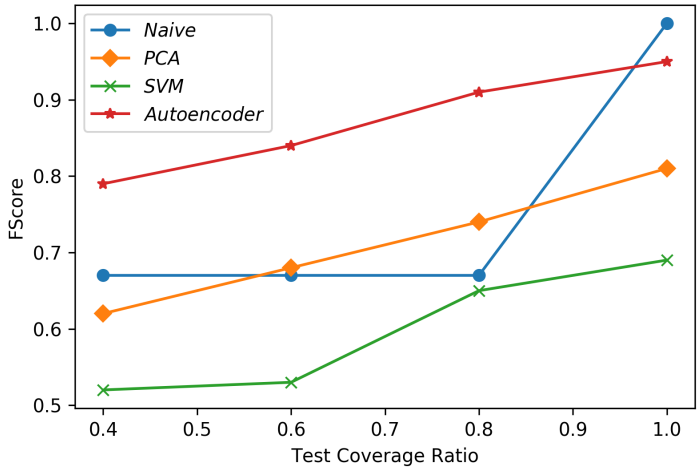

Fig. 14. Performance of Different Machine Learning Algorithms Under Different Test Coverage Ratios.

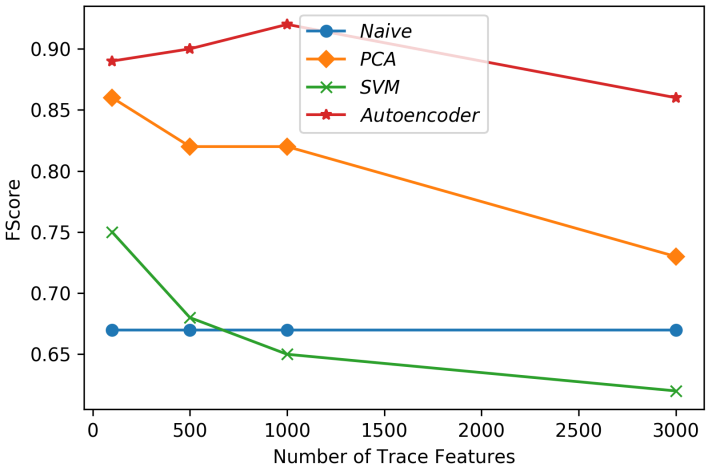

Fig. 15. Performance of Different Machine Learning Algorithms Under Different Input Feature Dimensions.

Figure 15 shows the performance of machine learning algorithms under different input feature dimensions (the unique feature ratio is kept constant). This figure shows the difference between the autoencoder and other approaches are not significant when the number of feature is small. As the number of feature increases, however, this gap becomes larger. The autoencoder shows robust performance even with complicated high dimension input data.

Figure 16 compares the performance of machine learning algorithms under different unique feature ratios. This figure shows that the performance of the machine learning algorithms improves as the unique feature ratio increases.

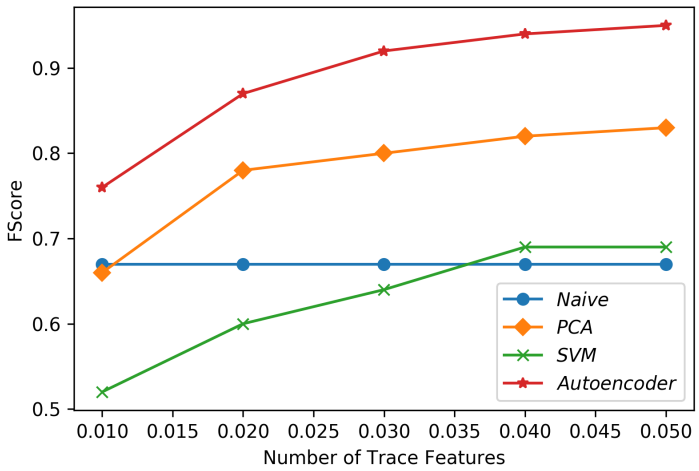

Fig. 16. Performance of Different Machine Learning Algorithm Under Different Unique Feature Ratios.

This result is not surprising because the statistical difference between normal and abnormal requests is larger and easier to capture. For the autoencoder algorithm at least 2% of unique features are needed in the abnormal requests for acceptable performance. The experiment was conducted on a desktop with Intel i5 3570 and GTX 960 GPU running Windows 10. The autoencoder was implemented using Keras 2.0 with a TensorFlow backend.

Table 5 compares the training/classification time for different algorithms. The training was performed with the same set of 5,000 traces with default parameters specified in Section 4.3. The classification time is the average time to classify one trace over 1,000 test traces.

The results in Table 5 show that the training time of the deep autoencoder is significant longer than other approaches. This training need not be performed frequently, however, and can also be done offline. Moreover, existing deep learning frameworks (such as TensorFlow) support powerful GPUs, which can also significantly accelerate training time.

For the classification time, all machine learning algorithms can perform classification in an acceptable short period of time with the trained model. Moreover, hardware advances (such as the Tensor Processing Unit [Schneider 2017]) are bringing high performance and low cost computing resources in the future. Computation cost should thus not be a bottleneck for future deployments of deep learning to detect web attacks.

Table 5. Comparison of Training/Classification Time for Different Algorithms.

	Training Time	Classification Time
Naive	51s	0.05s
PCA	2min 12s	0.2s
One-class SVM	2min 6s	0.2s
Autoencoder	8min 24s	0.4s

6 RELATED WORK

Intrusion detection systems monitor a network and/or system for malicious activity or policy violations [Di Pietro and Mancini 2008]. These types of systems have been studied extensively in the literature based on various approaches, including static analysis [Fu et al. 2007; Wassermann and Su 2008], sequence-based [Sekar et al. 2001; Wagner and Dean 2001], manual modeling [Friedenthal et al. 2014; Scott 2004], and machine learning [Farid et al. 2010; Zolotukhin et al. 2014]. This section describes prior work and compares/contrasts it to our research on RSMT presented in this paper.

6.1 Static Analysis

Static analysis approaches examine an applicationâĂŹs source code and search for potential flaws in its construction and expected execution that could lead to attack. For example, Fu et al. [Fu et al. 2007] statically analyzed SQL queries and built grammars representing expected parameterization. Wassermann et al. [Wassermann and Su 2008] presented a static analysis for detecting XSS vulnerabilities using tainted information flow with string analysis. Kolosnjaji et al. [Kolosnjaji et al. 2016] proposed an analysis on system call sequences for malware classification.

Statically derived models can also be used at runtime to detect parameterizations of the SQL queries that do not fit the grammar and indicate possible attack. Static analysis approaches, however, typically focus on specific types of attacks that are known as *a priori*. In contrast, RSMT bypasses these various attack vectors and captures the low-level call graph under the assumption that the access pattern of attack requests will be statistically different than legitimate requests, as shown in Section 3.

Moreover, many static analysis techniques require access to application source code, which may not be available for many production systems. Employing attack-specific detection approaches requires building a corpus of known attacks and combining detection techniques to secure an application. A significant drawback of this approach, however, is that it does not protect against unknown attacks for which no detection techniques have been defined. In contrast, RSMT models correct program execution behaviors and uses these models to detect abnormality, which works even if attacks are unknown, as shown in Section 4.

6.2 Sequence-based

Sequence-based anomaly detection approaches [Sekar et al. 2001; Wagner and Dean 2001] either try to model the call sequences as a finite-state automaton (FSA), Hidden Markov Models (HMM) [Warrender et al. 1999] or N-gram [Hofmeyr et al. 1998]. FSA can capture common program structures such as loops or branches and predict future behaviors from past behaviors. Although sequence-based approaches achieved early success, their time complexity is high.

It has also been shown that there is no polynomial time algorithm for learning an optimal FSA [Kearns and Valiant 1994]. N-gram [Hofmeyr et al. 1998] breaks a system call sequence into subsequences of fixed length N. The limitation of N-gram, however, is the number of N-gram grow exponentially with N. N must therefore be small, though a small N makes the algorithm ineffective at capturing long-term correlations. Moreover, the false alarm rate is high for N-gram because it cannot generalize to any N-gram that are not present in the training dataset.

6.3 Manual Modeling

Manual modeling relies on designers to annotate code or build auxiliary textual or graphical models to describe expected system behavior. For example, SysML [Friedenthal et al. 2014] is a language that allows users to define parametric constraint relationships between different parameters of the system to indicate how changes in one parameter should propagate or affect other parameters. Scott [Scott 2004] proposed a Bayesian model-based design for intrusion detection

systems. Ilgun et al. [Ilgun et al. 1995] used state transitions to model the intrusion process and build a rule-based intrusion detection system.

Manual modeling is highly effective when analysis can be performed on models to simulate or verify that error states are not reached. Although expert modelers can manually make errors, many errors can be detected via model simulation and verification. A key challenge of using manual modeling alone for detecting cyber-attacks, however, is that it may not fully express or capture all characteristics needed to identify the attacks. Since manual models typically use abstractions to simplify their usage and specification of system properties, these abstractions may not provide sufficient expressiveness to describe properties needed to detect unknown cyber-attacks. Our deep learning approach uses RSMT to analyze raw request trace data without making any assumption of the relationships or constraints of the system, thereby overcoming limitations with manual modeling, as shown in Section 3.

6.4 Machine Learning

Machine learning approaches require instrumenting a running system to measure various properties (such as execution time, resource consumption, and input characteristics) to determine when the system is executing correctly or incorrectly due to cyber-attacks, implementation bugs, or performance bottlenecks. For example, Farid et al. [Farid et al. 2010] proposed an adaptive intrusion detection system by combining naive bayes and decision tree. Zolotukhin et al. [Zolotukhin et al. 2014] analyzed HTTP request with PCA, SVDD, and DBSCAN for unsupervised anomaly detection. Likewise, Shar et al. [Shar et al. 2015] used random forest and co-forest on hybrid program features to predict web application vulnerabilities.

Anomaly detection is another machine learning [Chandola et al. 2009] application that addresses cases where traditional classification algorithms work poorly, such as when labeled training data is imbalanced. Common anomaly detection algorithms include mixture Gaussian models, support vector machines, and cluster-based models [Leung and Leckie 2005]. Likewise, autoencoder techniques have shown promising results in many anomaly detection tasks [Erfani et al. 2016; Sakurada and Yairi 2014; Xiong and Zuo 2016].

Our RSMT-baesd approach described in this paper uses a stacked autoencoder to build an end-to-end deep learning system for the intrusion detection domain. The accuracy of conventional machine learning algorithms [Farid et al. 2010; Shar et al. 2015] rely heavily on the quality of manually selected features, as well as the labeled training data. In contrast, our deep learning approach uses RSMT to extract features from high-dimensional raw input automatically without relying on domain knowledge, which enables it to achieve better detection accuracy with large training data, as shown in Section 5.5.

7 CONCLUSIONS

This paper describes the architecture and results of applying a unsupervised end-to-end deep learning approach to automatically detect attacks on web applications. We instrumented and analyzed web applications using the Robust Software Modeling Tool (RSMT), which autonomically monitors and characterizes the runtime behavior of web applications. We then applied a denoising autoencoder to learn a low-dimensional representation of the call traces extracted from application runtime. To validate our intrusion detection system, we created several test applications and synthetic trace datasets and then evaluated the performance of unsupervised learning against these datasets.

The following are key lessons learned from the work presented in this paper:

• **Autoencoders can learn descriptive representations from web application stack trace data**. Normal and anomalous requests are significantly different in terms of reconstruction error with representations learned by autoencoders. The learned representation reveals important features, but shields application developers from irrelevant details. The results of our experiments in Section 5.5 suggest the representation learned by our autoencoder is sufficiently descriptive to distinguish web request call traces.

• **Unsupervised deep learning can achieve over 0.91 F1-score in web attack detection without using domain knowledge**. By modeling the correct behavior of the web applications, unsupervised deep learning can detect different types of attacks, including SQL injection, XSS or deserialization with high precision and recall. Moreover, less expertise and effort is needed since the training requires minimum domain knowledge and labeled training data.

• **End-to-end deep learning can be applied to detect web attacks**. The accuracy of the end-to-end deep learning can usually outperform systems built with specific human knowledge. The results of our experiments in Section 5.5 suggest end-to-end deep learning can be successfully applied to detect web attacks. The end-to-end deep learning approach using autoencoders achieves better performance than supervised methods in web attack detection without requiring any application-specific prior knowledge.

In future work, we plan to investigate more complex network structures such as LSTM autoencoders or autoencoders with CNNs. We would like to see whether these structures can provide better accuracy for web attack detection tasks. Also, we plan to develop mechanisms to distribute the machine learning analysis workload across remote machines and support coordinated distributed detection across hosts.

REFERENCES

Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. 2016. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*. 173–182.

Noam Ben-Asher and Cleotilde Gonzalez. 2015. Effects of cyber security knowledge on attack detection. *Computers in Human Behavior* 48 (2015), 51–61.

Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking path explosion in constraint-based test generation. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 351–366.

Anna L Buczak and Erhan Guven. 2015. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys & Tutorials* 18, 2 (2015), 1153–1176.

Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009), 15.

Corinna Cortes and Vladimir Vapnik. 1995. Support vector machine. *Machine learning* 20, 3 (1995), 273–297.

cyberattack 2018. Acunetix. https://www.acunetix.com/acunetix-web-application-vulnerability-report-2016/.

cyberattackloss 2018. Cyber attack loss. http://money.cnn.com/2015/10/08/technology/cybercrime-cost-business/index.html.

Roberto Di Pietro and Luigi V Mancini. 2008. *Intrusion detection systems*. Vol. 38. Springer Science & Business Media.

elasticsearch 2018. Elasticsearch. https://www.elastic.co/products/elasticsearch.

equifax 2018. Equifax Data Breach. https://www.consumer.ftc.gov/blog/2017/09/equifax-data-breach-what-do.

equifax2 2018. News. https://theconversation.com/why-dont-big-companies-keep-their-computer-systems-up-to-date-84250.

Sarah M Erfani, Sutharshan Rajasegarar, Shanika Karunasekera, and Christopher Leckie. 2016. High-dimensional and large-scale anomaly detection using a linear one-class SVM with deep learning. *Pattern Recognition* 58 (2016), 121–134.

Dewan Md Farid, Nouria Harbi, and Mohammad Zahidur Rahman. 2010. Combining naive bayes and decision tree for adaptive intrusion detection. *arXiv preprint arXiv:1005.4496* (2010).

Sanford Friedenthal, Alan Moore, and Rick Steiner. 2014. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann.

Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. 2007. A static analysis framework for detecting SQL injection vulnerabilities. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, Vol. 1. IEEE, 87–96.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

Alex Graves and Navdeep Jaitly. 2014. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. 1764–1772.

William G Halfond, Jeremy Viegas, and Alessandro Orso. 2006. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Vol. 1. IEEE, 13–15.

Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. 1998. Intrusion detection using sequences of system calls. *Journal of computer security* 6, 3 (1998), 151–180.

Koral Ilgun, Richard A Kemmerer, and Phillip A Porras. 1995. State transition analysis: A rule-based intrusion detection approach. *IEEE transactions on software engineering* 21, 3 (1995), 181–199.

Nathalie Japkowicz and Shaju Stephen. 2002. The class imbalance problem: A systematic study. *Intelligent data analysis* 6, 5 (2002), 429–449.

jmeter 2018. JMeter. http://jmeter.apache.org/.

Michael Kearns and Leslie Valiant. 1994. Cryptographic limitations on learning Boolean formulae and finite automata. *Journal of the ACM (JACM)* 41, 1 (1994), 67–95.

Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. 2016. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*. Springer, 137–149.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

Kingsly Leung and Christopher Leckie. 2005. Unsupervised anomaly detection in network intrusion detection using clusters. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*. Australian Computer Society, Inc., 333–342.

Guisong Liu, Zhang Yi, and Shangming Yang. 2007. A hierarchical intrusion detection model based on the PCA neural networks. *Neurocomputing* 70, 7 (2007), 1561–1568.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9, Nov (2008), 2579–2605.

Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

David W Opitz and Richard Maclin. 1999. Popular ensemble methods: An empirical study. *J. Artif. Intell. Res.(JAIR)* 11 (1999), 169–198.

owasp2013 2018a. 2013 OWASP Top 10 Most Dangerous Web Vulnerabilities. https://www.owasp.org/index.php/Top_10_2013-Top_10.

owasp2013 2018b. Deserialization attack.

Tadeusz Pietraszek. 2004. Using adaptive alert classification to reduce false positives in intrusion detection. In *Recent Advances in Intrusion Detection*. Springer, 102–124.

David Martin Powers. 2011. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. (2011).

Xiaohu Qie, Ruoming Pang, and Larry Peterson. 2002. Defensive programming: Using an annotation toolkit to build DoS-resistant software. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 45–60.

Stuart Russell, Peter Norvig, and Artificial Intelligence. 1995. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995), 27.

Mayu Sakurada and Takehisa Yairi. 2014. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*. ACM, 4.

David Schneider. 2017. Deeper and cheaper machine learning [top tech 2017]. *IEEE Spectrum* 54, 1 (2017), 42–43.

Steven L Scott. 2004. A Bayesian paradigm for designing intrusion detection systems. *Computational statistics & data analysis* 45, 1 (2004), 69–83.

R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 144–155.

Lwin Khin Shar, Lionel C Briand, and Hee Beng Kuan Tan. 2015. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Transactions on Dependable and Secure Computing* 12, 6 (2015), 688–707.

spec2008 2018. SPECjvm2008. https://www.spec.org/jvm2008/.

Fangzhou Sun, Peng Zhang, Jules White, Douglas Schmidt, Jacob Staples, and Lee Krause. 2017. A Feasibility Study of Autonomically Detecting In-process Cyber-Attacks. In *Cybernetics (CYBCON), 2017 3rd IEEE International Conference on*. IEEE, 1–8.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*. ACM, 1096–1103.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. 2010. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research* 11, Dec (2010), 3371–3408.

Daniel G Waddington, Nilabja Roy, and Douglas C Schmidt. 2009. Dynamic analysis and profiling of multi-threaded systems. *IGI Global* (2009).

David Wagner and R Dean. 2001. Intrusion detection via static analysis. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 156–168.

Yanxin Wang, Johnny Wong, and Andrew Miner. 2004. Anomaly intrusion detection using one class SVM. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*. IEEE, 358–364.

Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. 1999. Detecting intrusions using system calls: Alternative data models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*. IEEE, 133–145.

Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*. ACM, 171–180.

Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.

Yihui Xiong and Renguang Zuo. 2016. Recognition of geochemical anomalies using a deep autoencoder network. *Computers & Geosciences* 86 (2016), 75–82.

Xin Xu and Xuening Wang. 2005. An adaptive network intrusion detection method based on PCA and support vector machines. *Advanced Data Mining and Applications* (2005), 731–731.

ysoserial [n. d.].

Mikhail Zolotukhin, Timo Hämäläinen, Tero Kokkonen, and Jarmo Siltanen. 2014. Analysis of http requests for anomaly detection of web attacks. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*. IEEE, 406–411.