A Feasibility Study of Autonomically Detecting In-process Cyber-Attacks

Fangzhou Sun, Peng Zhang, Jules White, and Douglas C. Schmidt
Department of EECS
Vanderbilt University, Nashville, TN, USA
{fangzhou.sun, peng.zhang, jules.white, d.schmidt}@vanderbilt.edu

Jacob Staples and Lee Krause
Securboration Inc.
Melbourne, FL, USA
{jstaples, lkrause}@securboration.com

***Abstract*—A cyber-attack detection system issues alerts when an attacker attempts to coerce a trusted software application to perform unsafe actions on the attacker's behalf. One way of issuing such alerts is to create an application-agnostic cyber-attack detection system that responds to prevalent software vulnerabilities. The creation of such an autonomic alert system, however, is impeded by the disparity between implementation language, function, quality-of-service (QoS) requirements, and architectural patterns present in applications, all of which contribute to the rapidly changing threat landscape presented by modern heterogeneous software systems.**

This paper evaluates the feasibility of creating an autonomic cyber-attack detection system and applying it to several exemplar web-based applications using program transformation and machine learning techniques. Specifically, we examine whether it is possible to detect cyber-attacks (1) *online*, *i.e.*, as they occur using lightweight structures derived from a call graph and (2) offline, *i.e.*, using machine learning techniques trained with features extracted from a trace of application execution. In both cases, we first characterize normal application behavior using supervised training with the test suites created for an application as part of the software development process. We then intentionally perturb our test applications so they are vulnerable to common attack vectors and then evaluate the effectiveness of various feature extraction and learning strategies on the perturbed applications. Our results show that both lightweight on-line models based on control flow of execution path and application specific off-line models can successfully and efficiently detect in-process cyber-attacks against web applications.

***Keywords*-cyber security; machine learning; application instrumentation; unit test;**

I. Introduction

Emerging trends and challenges. Cyber-attacks continue to grow in frequency and severity, *e.g.*, from 2014 to 2015 the average annualized loss from cyber-attacks increased \$8.96 million in financial services companies, \$6.08 million in technology companies, and \$6.44 in retail companies [1]. Various techniques have been developed to help prevent and defend against cyber-attacks. Manual approaches [2], [3], [4], such as defensive programming and code reviews, are widely applied to limit and correct mistakes made by software developers. Dynamic taint analysis techniques [5], [6] aid in detecting code vulnerabilities. Likewise, machine learning techniques [7], [8], [9] have been applied to detect cyber-attacks and identify vulnerabilities.

Although advances in information security enable more effective monitoring and threat detection, cybersecurity remains largely an art rather than a science or engineering discipline since it often requires domain-specific knowledge, *i.e.*, the capabilities of human analysts and decision makers remain indispensable [10]. For example, cybersecurity techniques often require highly proficient web security knowledge and skills from developers and network operators.

Open problem → Efficiently and accurately detecting in-process cyber-attacks in web applications. Due to the diversity of programming languages, functionality, quality-of-service (QoS) requirements, and architectural patterns, no single security approach can find and assess all security risks in heterogeneous production systems. An open challenge is therefore determining how to build a scalable and resilient cyber infrastructure that can *autonomically* detect in-process cyber-attacks and adapt efficiently and securely to thwart these attacks without requiring application developers to possess in-depth expertise in cybersecurity tactics, techniques, and procedures.

Contributions. This paper evaluates the feasibility of autonomically detecting in-process cyber-attacks against heterogeneous web-based applications running on a Java Virtual Machine (JVM) [11] using program transformation and machine learning. In particular, we explore the efficacy of program transformation techniques to extract execution features from applications and machine learning-based detection techniques that operate on these features. The paper also addresses whether test suites can be leveraged effectively to supervise training of machine learning models to detect attacks. The technical underpinning of this study is the *Robust Software Modeling Tool* (RSMT), which is a novel program transformation tool that can be attached to web-based applications and use to automatically extract execution features that detect in-process cyber-attacks without requiring obtrusive code modifications.

Paper organization. Section II describes the open issues investigated in this paper; Section III describes RSMT's online, instrumentation-derived model based upon application control flow; Section IV analyzes RSMT's offline machine learning detection model; Section V empirically evaluates the performance overhead of RSMT's JVM agent on applications when detecting a range of cyber-attacks;

Section VI compares our work with related cyber-attack detection techniques; and Section VII presents concluding remarks.

II. Open Issues for In-Process Cyber-Attack Detection

To motivate more effective in-process cyber-attack detection techniques, this section summarizes the following open issues associated with capturing execution behaviors from web-based applications via program transformations and detecting potential cyber-attacks at runtime using machine learning.

Issue 1: Capturing features representative of application behavior without creating a significant burden on developers: To augment human cybersecurity experts, automated mechanisms are needed to capture and analyze application execution information. A promising approach uses machine learning to build models of expected application execution behavior. We define execution behavior as the invocation of methods, the ordering of method invocation, and the inputs/outputs of method invocations. Key open issues facing researchers are (1) how to instrument an application unobtrusively and (2) how much knowledge of the underlying code base and access to the code is required to collect execution features and train accurate machine learning attack detection models.

To answer these questions, we conducted a study that compared the following application instrumentation approaches: (1) A bytecode transformation system for tracking control flow within a running application. Due to the performance penalty incurred when tracking control flow, we developed an instrumentation system using a JVM agent, which is a JAR file that utilizes Java's Instrumentation API to intercept classload events and invisibly transform classes before they are loaded by the JVM at run time. (2) AspectJ is a general-purpose "out-of-the-box" instrumentation tool. We use AspectJ's pointcut and join point capabilities to weave application-specific feature extraction behaviors into a small number of compiled classes for runtime data monitoring, permission checks, action interruption and resource management, among others. Results from our work comparing these two approaches are provided in Section III and III.

Issue 2: Determining the performance overhead of execution feature vector collection.: Monitoring instructions executed along the critical path of a program can degrade its performance. To evaluate this degradation, Section V-A describes the results of experiments that evaluated the average and worst-case performance overhead of RSMT program transformation in web-based applications. In particular, we identified trade-offs between lightweight online detection techniques that identify obviously dangerous behaviors versus fine-grained offline detection techniques that identify subtler adverse behaviors.

Issue 3: Characterizing the performance of various machine learning approaches for detecting cyber-attacks: Different machine learning algorithms have different problem domains, average predictive accuracy, training and validation speeds, and data requirements. A key issue facing researchers is how these different algorithms perform with respect to detecting different types of cyber-attacks. To address this issue, we employed three machine learning techniques using implementations from the Weka library [12] (naive Bayes, support vector machine, and random forest) on three common cyber-attacks (SQL injection, directory traversal, and cross-site scripting) from the OWASP "top ten" list [13]. We then compared the results of accuracy, precision, recall, and f-score [14]. Section V-B presents our results, which indicate that no single classifier is best at detecting all attack types, thereby motivating our further investigation of hybrid ensemble-style approaches.

Issue 4: Characterizing feature vector abilities to reflect application behaviors: Web-based applications produce copious amounts of data corresponding to their execution behaviors. This runtime data stream must be represented and stored in a manner that enables its effective utilization in analytics and classification models. A key open issue facing researchers is how to construct the data at runtime and how to store the data online and offline. To address this issue, we implemented and compared three feature representations in RSMT: (1) A call graph that is used to determine whether a transition is abnormal, (2) A call tree that is used to determine whether a sequence of transitions is abnormal, and (3) Feature attributes (*e.g.*, method execution time, whether parameter contains special characters, etc.) that are used to represent the features that cannot be reflected in a call graph/tree. Detailed analyzes of our comparisons are presented in Section IV-B and III-3.

III. Monitoring Program Behaviors with Java Instrumentation and Online Detection Using Call Graphs and Trees

This section describes the online (*i.e.*, in the critical path of execution) mechanism that RSMT employs to capture and validate program behaviors. Behavior capture and validation is achieved through a custom instrumentation system that enables the extraction of call graphs and call traces at runtime.

The functional components of this architecture are shown in Figure 1 and include: (1) a class transformation system that passes loaded Java class files to various class transformation components that are managed by a class transformer registry, (2) a runtime API that is invoked by instrumented code, (3) a filter that enables dynamic software probing, (4) a model builder that listens to runtime events and builds a model of system behavior, and (5) a model enforcer that compares a model representing correct system behavior to a snapshot of the currently observed behavior.

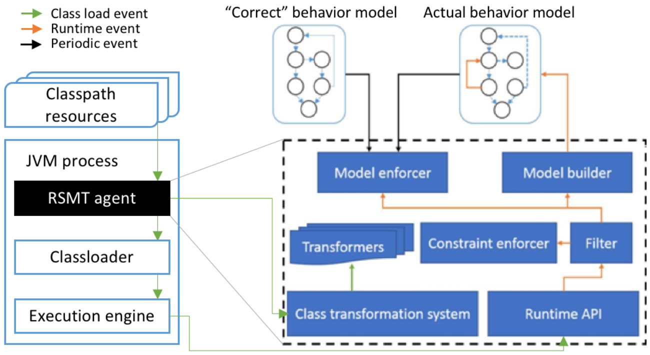

Figure 1. The Architecture of the RSMT Online Monitoring and Detection Model

1) Addressing issue 1 via class transformation and bytecode instrumentation: To address issue 1 in Section II (*i.e.*, how to capture features representative of application behavior without burdening developers), we developed a class transformation system in RSMT that creates events to generalize and characterize program behavior at runtime. The transformation system is plugin-based and thus extensible. In particular, 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 could 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) mechanism, which allows (1) the rapid dissemintation of events by instrumented code and (2) the subsequent capture by event listeners that can be registered dynamically at runtime. This pub-sub mechanism is exposed to instrumented bytecode via a proxy class that contains various static methods.[1] In turn, this proxy class is responsible for calling various listeners that have been registered to 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.
- *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.

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

- *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 handles exceptions generated in 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).
- The offline model construction/validation subtree converts events into a form can be stored asynchronously with a (possibly remote) instance of Elasticsearch [15], which is an open-source search and analytics engine that provides a distributed real-time document store.

2) Addressing issue 2 by improving performance via dynamic probes: To address issue 2 in Section II (*i.e.*, reduce the performance overhead of execution feature vector collection), we analyzed the method call patterns and observed that the majority of method calls are typically 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 (and therefore achieve improved performance). These observations, along with a desire for improved performance, yielded the creation of a dynamic filtering mechanism in RSMT.

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.

3) Addressing issue 4 via an online model builder and model validator: To address issue 4 in Section II (characterizing feature vector abilities to reflect application behaviors), we developed an online model builder and model validator. The model builder is responsible for constructing two views of software behavior: a call graph (used to quickly determine whether a transition is valid) and a call tree (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. Figures 3 and 4 demonstrate the complexity of the graphs we have seen. Each directed edge in a call graph connects a parent method (source) to a

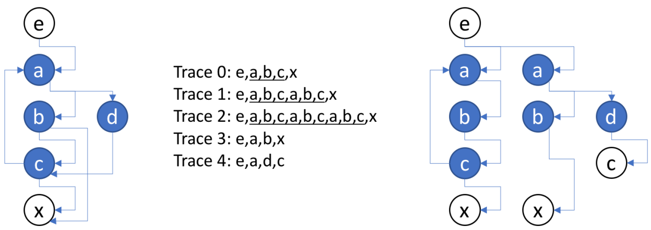

Figure 2. Call Graph (L) and Call Tree (R) Constructed for a Simple Series of Call Stack Traces

method called by the parent (destination). Call graph edges are not restricted wrt forming cycles. Suppose the graph in Figure 2 represented correct behavior. If we observed a call sequence *e,a,x* at runtime, we could easily tell that this was not a valid execution path because no *a,x* edge is present in the call graph.

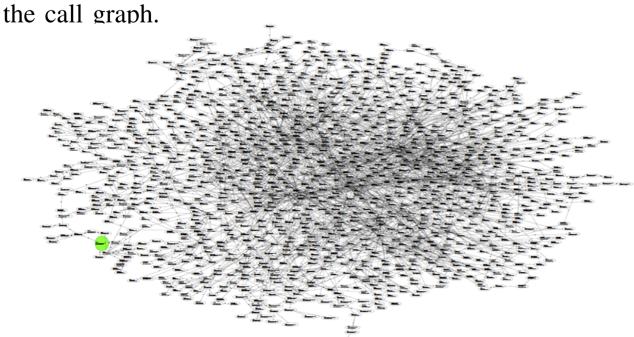

Figure 3. Call Tree Generated for a Simple SQL Statement Parse

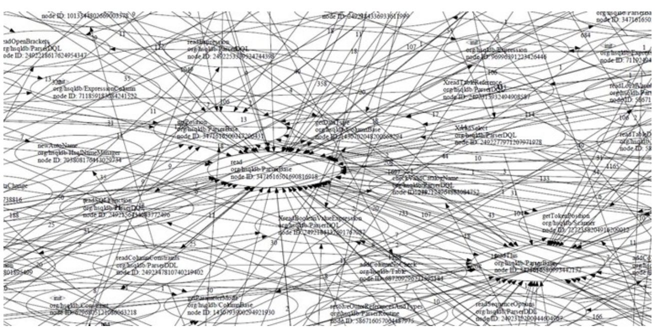

Figure 4. Call Tree Generated for a Simple SQL Statement Parse (zoomed in on heavily visited nodes)

Although the call graph is fast and simple to construct, it has shortcomings. For example, suppose we observed a transition sequence *e,a,d,c,a*. 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 more complex cases, a more robust structure is needed. We call such a structure the *call tree*, as shown in the right-hand side of Figure 2.

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. Rather, it is a tree in that 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 2) if recursion is observed. Using this structure is obviously more processor intensive than tracking behavior using a call graph.

IV. Monitoring with AOP and Offline Detection Using Machine Learning Techniques

To explore the feasibility of using execution traces, machine learning, and unit tests for cyber-attack detection, we created an offline monitoring and detection framework for Java web applications (its structure is shown in Figure 5). This section first presents our design of annotations and aspect-oriented programming techniques to capture the runtime data. It then describes how we create different feature vectors and employ machine learning algorithms for attack detection. The experimental results and analysis of detection performance are presented in Section V.

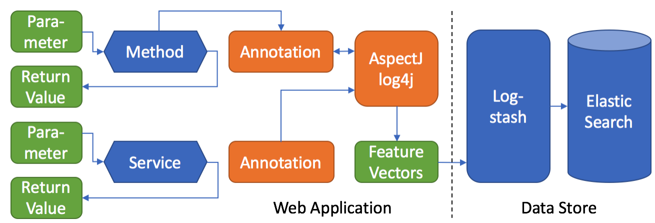

Figure 5. The Design of AspectJ Real-time Web Application Monitoring

A. Addressing issue 1 via capturing the runtime data using AspectJ and collecting and storing the data in a reliable data store

To address issue 1 in Section II (*i.e.*, capturing application behavior at runtime and transferring the data to detection models), we employ AspectJ [16], which provide aspect-oriented extensions to Java. When a web application is under attack, function parameters or return values may become abnormal and thus can be utilized to identify malicious behaviors. To monitor runtime behaviors of Java web applications without modifying the source code or altering its original functional behavior, we employed AspectJ's annotations. In particular, to access the method arguments and return value at runtime, we defined the following annotation types to use in the source code of web applications:

- `@Input (arguments)`. This annotation type can be used to acquire the arguments of the annotated method and other necessary data.
- `@Output (return values)`. This annotation type can be used to obtain the return value of the annotated method.

To process the annotations demonstrated above, we used AspectJ to implement a lightweight annotation processing engine that creates pointcuts around the executed method, captures and the data-like parameters, returns value as log messages, and then outputs the data as log messages via log4j.

To create a reliable mechanism to store and transfer the data captured by AspectJ to the detection models, Logstash and Elasticsearch were used to collect and manage the runtime data captured from multiple web applications. Logstash is a tool that collects and parses events and logs. We customized a simple Logstash plug-in that uses a grok filter [17] (a default way to filter and parse log messages for Logstash) to open and read each logging message from log4j, parses the log lines into required format, and then stores the processed data in Elasticsearch.

Elasticsearch is an open-source search and analytics engine built on top of Apache Lucene that provides a distributed real-time document store where all fields can be indexed and full text of unstructured data are search-able. We configured Elasticsearch to store and manage the captured runtime data from web applications. These data is later queried by our machine learning back end presented in Section IV via the APIs provided naively by Elasticsearch.

B. Addressing issue 4 via machine learning cyber-attack detection techniques

To address issue 4 in Section II (*i.e.*, using proper feature vectors to represent runtime data stream), we developed two types of feature vectors and utilized three machine learning techniques to analyze and identify the possible cyber-attacks. Each of these techniques is described below.
Feature vectors Datasets and feature vectors are crucial for cyber-attack detection systems. Incorrect selection of properties to measure from a running system can yield both (1) unnecessary noise that reduces the performance of machine learning algorithms and (2) false positive and false negative predictions about system execution correctness. To study the feature vector abilities to reflect application behaviors, we designed and created the following feature attributes:

1) *Method execution time*. Attack behaviors can result in abnormal method execution times, *e.g.*, SQL injection attack might require less time to execute than normal database queries.
2) *User principle 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.
3) *Argument length*, which represents the number of characters of an argument, *e.g.*, XSS attacks might input some abnormal arguments that cause the overall argument length to be much larger than normal, such as: http://www.msn.es/usuario/guias123/default.asp?sec=-"></script><script>alert("Da-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*. Since many web browsers automatically ignore and correct duplicated characters, attackers can insert duplicated characters into requests to fool validators.

Our second feature vector was built using the n-gram [14] 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 used to make the feature vectors.

Machine learning detection models. We use machine learning algorithms from the Weka workbench, which provides a complete environment for classification, regression, and clustering. In the training stage, the machine learning system first queries the Elasticsearch engine and generates datasets in the Attribute-Relation File Format (ARFF) that are required by the Weka library. We used three original machine learning classifiers and two additional classifiers that utilize the results of three original classifiers, as follows:

1) *Naive Bayes*, which makes classification decisions by calculating the probabilities and costs for each decision. Bayes network classifiers are widely used in cyber-attack detection [18].
2) *Random forests*, which is an ensemble learning method for classification. Random forests train decision trees on sub-samples of the dataset and then use averaging to improve classification accuracy.
3) *Support vector machine (SVM)*, which is a supervised learning model that draws an optimal hyperplane in the feature space and divides separate categories as wide as possible. SVM is very efficient in classification problems. In our machine learning system, we use the Sequential Minimal Optimization (SMO) algorithm provided by Weka to train the SVM.
4) *Aggregate_vote*, which is is an aggregate classifier. If more than half of the classifiers claim to have detected attacks this classifier return ATTACKs, otherwise it returns NOT_ATTACK.
5) *Aggregate_any*. If any one of the classifiers claims to have detected attacks this classifier would return ATTACK, otherwise it returns NOT_ATTACK.

In the detection stage, the machine learning system provides a GET API for third-party applications to query the detection result.

V. Experimental Evaluation

This section presents the experimental evaluation of the RMST tool. We first evaluate the performance overhead of

RSMT's JVM agent on applications when detecting a range of cyber-attacks. We then describe the test environment created to evaluate the attack detection performance of the off-line machine learning models on three types of cyber-attacks: cross-site scripting, SQL injection, and directory traversal.

A. *Overhead Observations*

To address issue 2 in Section II (*i.e.*, to examine the performance overhead of execution feature vector collection), we conducted experiments that evaluated the runtime overhead in average cases and worst cases, as well as assess how "real-time" application execution monitoring and abnormal detection could be. In applications that are not computationally constrained, RSMT has low overhead. 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. However, this startup cost is amortized since class loading typically happens just once per class.

In addition to the startup delay, RSMT incurs some runtime performance overhead every time instrumented code is invoked. We tested several web services and found that RSMT had an overhead ranging from 5% to 20%. The factors that most strongly impact the overhead are number of methods called (more frequent invocation results in higher overhead) and 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 Apaches Commons-Compress library to bz2 compress randomly generated files of varying sizes ranging from 1x64B blocks to 1024x64B blocks, which is a control flow intensive task. Moreover, the Apache Commons implementation of bz2 is "method heavy" (*e.g.*, there are a significant number of setter and getter calls), which are typically optimized away by the JVMs hotspot compiler and converted into direct variable accesses. The instrumentation performed by RSMT prevents this optimization from occurring, however, since 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 6 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 results in a significant performance penalty, particularly the number of randomly generated blocks increases. Call graph tracking ranges from 1.5x to over 10x slower than the original application. Call tree tracking results in a 2-5x slowdown. Similarly, fine-grained control flow tracking results in a 4-6x slowdown.

As a result, with full, fine-grained tracking enabled, an application might run at 1% its original speed. By filtering

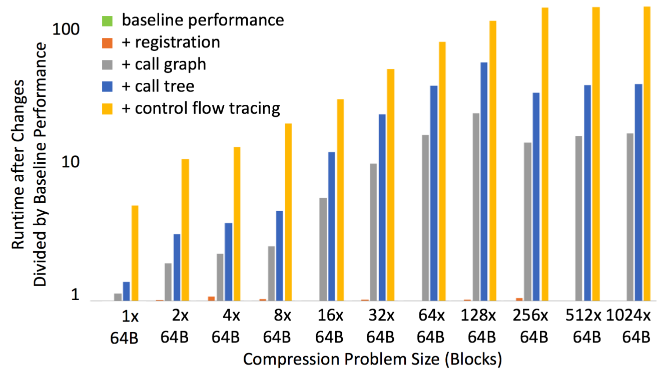

Figure 6. RSMT Performance Overhead Analysis

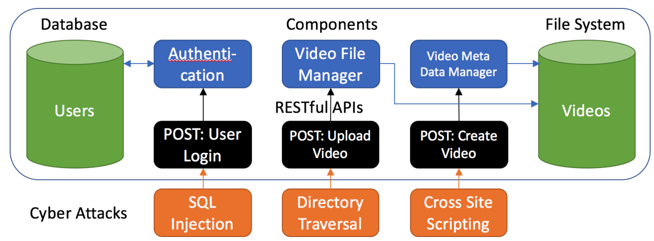

Figure 7. Several Cyber-attacks Are Exploited on the Test Video Management Application

getters and setters, however, it is possible to reduce this overhead by several orders of magnitude. We are investigating these types of optimization techniques and present the overhead reported here as a baseline for future enhancements.

B. *Detection Results*

To address issue 3 in Section II (*i.e.*, to examine the effectiveness of various machine learning techniques), we developed a Spring Boot [19] web application as the test environment, created a different test, and demonstrated exploits of some common cyber-attacks on it, as described below.

1) Setting Up the Test Environment: Figure 7 shows the test web application provides several RESTful APIs: (1) *user authentication*, where a GET API allows clients to send username and password to the server and then check the SQL database in the back-end for authentication, (2) *video creation*, where a POST API allows clients to create or modify video meta data, 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's ID.

To evaluate the system's attack detection performance, we exploit three attacks from OWASP's top ten cybersecurity vulnerabilities list [13] into the test application: (1) SQL injection, (2) directory traversal, and (3) cross-site scripting. We then investigated the overall accuracy, precision, recall, and f-score of the different machine learning models (naive Bayes, random forests and support vector machine [18]) in detecting and preventing attacks. We also use two additional aggregate models: (A) Aggregate_vote, where if more than half of the classifiers claim to have detected attacks this classifier return ATTACK, otherwise it returns NOT ATTACK and (2) Aggregate_any, where if any one of the classifiers claims to have detected attacks this classifier would return ATTACK, otherwise it returns NOT ATTACK.

2) Detecting the Directory Traversal Attacks: For the Directory Traversal Attacks, the training dataset contains 1,000 safe unit tests and 500 attack unit tests, while the validation dataset contains 250 safe unit tests and 125 attack unit tests (all attack samples are collected from OWASP website). Table I shows that the random forest classifier outperformed others, with highest accuracy, precision, recall, and f-score.

	accuracy	precision	recall	f-score
naive bayes	0.8044	0.7183	0.6800	0.6986
random forest	0.9333	0.8333	1.0000	0.9091
SVM	0.8400	0.6970	0.9200	0.7931
AGGREGATE_VOTE	0.8404	0.7019	0.9040	0.7902
AGGREGATE_ANY	0.8670	0.7193	0.9840	0.8311

Table I
MACHINE LEARNING MODELS' EXPERIMENTAL RESULTS FOR DIRECTORY TRAVERSAL ATTACKS

3) Detecting the XSS Attacks: The XSS training dataset contains 1,000 safe unit tests and 500 attack unit tests, while the validation dataset contains 150 safe unit tests and 75 attack unit tests (XSS attack samples are URLs obtained `www.xssed.com`). All three classifiers show similar effectiveness in detecting XSS attacks.

	accuracy	precision	recall	f-score
naive bayes	0.8711	0.0.7211	1.0000	0.8380
random forest	0.8711	0.0.7211	1.0000	0.8380
SVM	0.8756	0.7282	1.0000	0.8427
AGGREGATE_VOTE	0.8722	0.7238	1.0000	0.8398
AGGREGATE_ANY	0.8634	0.7102	1.0000	0.8306

Table II
MACHINE LEARNING MODELS' EXPERIMENTAL RESULTS FOR CROSS-SITE SCRIPTING ATTACKS

4) Detecting the SQL Injection Attacks: For the SQL injection attacks, the training dataset contains 160 safe unit tests and 80 attack unit tests, while the validation dataset contains 40 safe unit tests and 20 attack unit tests. The SQL injection attack samples are made specifically to bypass the test applications user authentication, including the most common SQL injection attack types.

	accuracy	precision	recall	f-score
naive bayes	0.9167	0.0.9412	0.8000	0.8649
random forest	0.9333	1.0000	0.8000	0.8889
SVM	cell5	0.9333	1.0000	0.8889
AGGREGATE_VOTE	0.9333	1.0000	0.8000	0.8889
AGGREGATE_ANY	0.9167	0.9412	0.8000	0.8649

Table III
MACHINE LEARNING MODELS' EXPERIMENTAL RESULTS FOR SQL INJECTION ATTACKS

VI. RELATED WORK

This section compares our research on RSMT with related work.

Static analysis approaches read an application's source code and search for potential flaws in its construction and expected execution that could lead to attacks. For example, prior techniques have statically analyzed SQL queries and built grammars representing expected parameterization. These statically derived models are used at runtime to detect parameterizations of the SQL queries that do not fit the grammar and indicate possible attacks. Livshits et al. [20] propose a static analysis technique for detecting SQL injection, cross-site scripting, and HTTP splitting attacks. Their system applies user-provided specifications of vulnerabilities to analyze code statically without code execution. Huang et al. [21] present the WAVES web application security assessing tool and the NRE algorithm. Using web crawlers, they first identify all possible data entry points that are vulnerable for attacks. Then they attack the most vulnerable points determined by several malicious patterns. The resulting pages then would be analyzed using the NRE algorithm. Halfond et al. [22] combine conservative static analysis and runtime monitoring to detect SQL injection attacks.

Manual modeling approaches rely on system designers to annotate code or build auxiliary textual or graphical models to describe expected system behavior. For example, SysML 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. Kemalis et al. [23] describe a prototype SQL injection detection system that utilizes specifications that define the intended syntactic structure of SQL queires and monitor Java-based applications and detect SQL injection attacks in real time. Kosuga et al. [24] present a technique, Sania, to detect SQL injection in web applications during development and debugging phases. The parse tree of intended SQL query and the actual results are compared to assess the spot safety. To detect and prevent SQL Injection attacks on web applications, Dharam et al. [25] evaluate a runtime monitoring framework that leverages the knowledge gained from pre-deployment testing of web applications to identify valid/legal execution paths.

Cyber-attack detection system based on machine learning typically build models that learn normal behaviors from the application and use the model to detect anomaly activities. Unsupervised and supervised approaches are two common types of methods. Valeur et al. [26] apply machine learning techniques to learn the profiles of normal database access and detect SQL injection attacks. Sharma et al. [27] introduce a new K-means algorithm for anomaly detection.

While RSMT does perform some degree of static analysis, its primary characterization of program behavior is derived from monitoring software as it executes when driven by real-world parameters. Unlike manual modeling approaches, RSMT does not require the presence of a rigorous model of all system behaviors. Programmers can provide hints to RSMT about the nature of data being manipulated and assumptions about that data.

VII. Concluding Remarks

This paper investigated the feasibility of creating an autonomic cyber-attack detection system capable of detecting attacks against modern applications running on a Java Virtual Machine (JVM). We described the *Robust Software Modeling Tool* (RSMT), which we developed to detect attacks using both lightweight models based on control flow extracted on the critical path of execution and application-specific models of behavior that are validated offline. We showed that unit tests can build useful models for characterizing application behaviors observed at deploy time and that a variety of useful features indicative of program behavior can be readily extracted using existing instrumentation frameworks. To validate our findings, we created several test applications vulnerable to prevalent attack vectors. We then evaluated the performance of RSMT in detecting attacks conducted against these test applications. Our results indicate that RSMT is a viable tool for detecting in-process cyber-attacks against web applications. During the study we learned the following key lessons:

- **Advantages from dynamic probes**. If every method of every library during the application execution is instrumented, the number of events will be overwhelmed. For example, we observed that Apache's Commons Compress library produces tens of millions of method call events when compressing a 1MB file (an operation that typically takes several milliseconds). The overhead of simply making these calls and generating events increases execution time by thirty percent or more. RSMT's dynamic and adaptive filtering strategy show promise in reducing performance overhead.
- **Limitations of existing machine learning approaches**. The machine learning algorithms used by RSMT are largely off-the-shelf (*i.e.*, provided by Weka). While their performance is acceptable for carefully-crafted programmer-provided features, they are not acceptable without these human-provided inputs. Since a goal of the RSMT project is to minimize the invasiveness of developers, customized machine learning algorithms are needed to improve the off-line model's performance.

References

[1] "2014 cost of cyber crime study: United states," https://ssl.www8.hp.com/us/en/ssl/leadgen/document_download.html?objid=4AA5-5208ENW.

[2] R. A. Baker Jr, "Code reviews enhance software quality," in *Proceedings of the 19th international conference on Software engineering*. ACM, 1997, pp. 570–571.

[3] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages." in *USENIX Security*, vol. 6, 2006, pp. 179–192.

[4] V. Satyanarayana and M. Sekhar, "Static analysis tool for detecting web application vulnerabilities," *Int. Journal of Modern Engineering Research (IJMER)*, vol. 1, no. 1, pp. 127–133, 2011.

[5] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 249–260.

[6] M. Martin and M. S. Lam, "Automatic generation of xss and sql injection attacks with goal-directed model checking," in *Proceedings of the 17th conference on Security symposium*. USENIX Association, 2008, pp. 31–43.

[7] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 6, pp. 688–707, 2015.

[8] H. Lu, B. Cukic, and M. Culp, "Software defect prediction using semi-supervised learning with dimension reduction," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 314–317.

[9] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[10] N. Ben-Asher and C. Gonzalez, "Effects of cyber security knowledge on attack detection," *Computers in Human Behavior*, vol. 48, pp. 51–61, 2015.

[11] "Java virtual machine," https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-1.html#jvms-1.2.

[12] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[13] "2013 owasp top 10 most dangerous web vulnerabilities," https://www.owasp.org/index.php/Top_10_2013-Top_10.

[14] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," 2011.

[15] "Elasticsearch," https://www.elastic.co/products/elasticsearch.

[16] "Aspectj," https://eclipse.org/aspectj/.

[17] "grok," https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html.

[18] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2015.

[19] "Spring boot," http://projects.spring.io/spring-boot/.

[20] B. V. Livshits and M. S. Lam, "Finding security errors in java programs with static analysis (tech report)," 2005.

[21] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003, pp. 148–159.

[22] W. G. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter sql-injection attacks," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–7.

[23] K. Kemalis and T. Tzouramanis, "Sql-ids: a specification-based approach for sql-injection detection," in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 2153–2158.

[24] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama, and Y. Takahama, "Sania: Syntactic and semantic analysis for automated testing against sql injection," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 107–117.

[25] R. Dharam and S. G. Shiva, "Runtime monitoring framework for sql injection attacks," *International Journal of Engineering and Technology*, vol. 6, no. 5, p. 392, 2014.

[26] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of sql attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2005, pp. 123–140.

[27] M. Sharma and D. Toshniwal, "Pre-clustering algorithm for anomaly detection and clustering that uses variable size buckets," in *Recent Advances in Information Technology (RAIT), 2012 1st International Conference on*. IEEE, 2012, pp. 515–519.