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

Fangzhou Sun, Peng Zhang, Jules White, and Douglas C. Schmidt
Institute of Software Integrated Systems
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, l krause}@securboration.com

Abstract—This paper evaluates the feasibility of creating an autonomic cyber-attack detection system and applying it to several exemplary 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, normal application behavior is first characterized using supervised training with the test suites created for an application as part of the software engineering process. We intentionally made our test applications vulnerable to several common attack vectors employed today and used them gauge the real-world effectiveness of various feature extraction and learning strategies. The results from our study demonstrate the feasibility of detecting in-process cyber-attacks using program transformation and machine learning models trained from test suites.

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 technologies 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 [5], [6] is another technique for detecting code vulnerabilities. Likewise, machine learning techniques [7], [8], [9] have been applied to detect cyber-attacks and predict vulnerability.

Although advances in information security enable more effective monitoring and threat detection, cybersecurity today remains more of an art than a science or an engineering discipline since it requires domain-specific knowledge and cognitive abilities, 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.

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 analyze 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 or not 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. The remainder of this paper is organized as follows: 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 empirically evaluates the performance overhead of RSMT’s JVM agent on applications when detecting a range of cyber-attacks; Section V compares our work with related cyber-attack detection techniques; and Section VI presents concluding remarks.

II. OPEN ISSUES FOR IN-PROCESS CYBER-ATTACK DETECTION

To motivate more effective in-process cyber-attack detection techniques, this section presents open issues associated with capturing execution behaviors from web-based applications with program transformations and detecting potential
cyber-attacks at run-time 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 execution information. A promising approach uses machine learning to learn 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 issue facing researchers is (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 implement an execution feature vector collection system that collects execution features and trains accurate machine learning attack detection models.

To address this issue, we implemented and evaluated a bytecode transformation system for tracking control flow within a running application. Due to the performance penalty incurred when tracking control flow, we opted to craft an in-house instrumentation system using a JVM agent: a JAR file that utilizes Java’s Instrumentation API to intercept class load events and invisibly transform classes before they are loaded by the JVM at run time. Results from our work with this system are presented in Section III.

**Issue 2: Determining the performance overhead of execution feature vector collection:** Any monitoring instructions executed along the critical path of a program can degrade its performance. To evaluate this degradation, Section IV-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 more subtle 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 answer this issue, we employed three machine learning techniques using implementations from the Weka library (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]. As shown in Section IV-B, our results indicate that no single classifier is best at detecting all attack types, so we also investigated various 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. Detailed analyzes of our comparisons are presented in Section III-4.

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

This section describes the online (i.e., on 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.

![Figure 1. The Architecture of the RSMT Online Monitoring and Detection Model](image)

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 run-time API that is invoked by instrumented code, (3) a filter that enables dynamic software probing, (4) a model builder that listens to run-time 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.

1) **Addressing Issue 1 via Class Transformation and Bytecode Instrumentation:** To answer Issue 1 in Section II of how to instrument an application, 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 method-Enter() and methodExit() calls to construct a control flow model. Each transformation plugin conforms to a common API that 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.

2) Addressing Issues 1 via Analyzing Code Through a Runtime API: A key component of the RSMT model is its publisher-subscriber mechanism that allows the rapid dissemination of events by instrumented code and subsequently captured by event listeners that can be registered dynamically at runtime. This API is exposed to instrumented bytecode via a proxy class that contains various static methods (calling a Java static method is up to 2x faster than calling a Java instance method). This proxy class, in turn, 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.
- **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 that 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.

3) Addressing Issue 2 via Improving Performance by Dynamic Probes: A key challenge facing researchers is to analyze the performance impact of instrumentation on applications and reduce overhead. To identify performance symptoms and solve bottlenecks, 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.

4) Addressing Issue 4 via an Online Model Builder and Model Validator: The model builder component 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.

![Call Graph (L) and Call Tree (R) Constructed for a Simple Series of Call Stack Traces](image)

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 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.

Although fast and simple to construct, the call graph 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, clearly 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, and an example is shown above in the right hand side of Figure 2.

Whereas the call graph falsely represents it as a valid sequence, we see that 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 collection approaches 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 small 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, though this startup cost is one-off 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 Apache 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 get and set calls), which are typically optimized away by the JVMs hotspot compiler and converted into direct variable accesses. The instrumentation performed by RSMT prevents this from occurring, however, since lightweight methods are wrapped in calls to the model construction and validation logic. As a result, our bz2 benchmark represents a worst case for RSMT performance.

Figure 5 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

![Figure 3. Call Tree Generated for a Simple SQL Statement Parse](image)

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

IV. EXPERIMENTAL EVALUATION

A. Overhead Observations

This section presents the results of experiments that evaluate the runtime overhead of the execution feature vector

![Figure 5. RSMT Performance Overhead Analysis](image)
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 getters and setters, however, it is possible to reduce this overhead by several orders of magnitude. We are continuing to investigate such techniques and present these overheads as a baseline for future enhancements.

B. Detection Results

To examine the effectiveness of various machine learning techniques and explore whether use unit tests can characterize normal application behaviors, we developed a Spring Boot [16] web application as the test environment, created different test, and demonstrated exploits of some common cyber attacks on it, as described below.

1) Setting Up the Test Environment: Figure 6 shows the test web application provides several RESTful APIs: (1) user authentication: a GET API that 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: a POST API that allows clients to create or modify video metadata; (3) video uploading/downloading: POST/GET APIs that 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 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 (naïve Bayes, random forests and support vector machine [17]) 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. Table I shows that the random forest classifier outperformed others, with highest accuracy, precision, recall, and f-score.

<table>
<thead>
<tr>
<th></th>
<th>accuracy</th>
<th>precision</th>
<th>recall</th>
<th>fscore</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve bayes</td>
<td>0.8044</td>
<td>0.7183</td>
<td>0.6800</td>
<td>0.6986</td>
</tr>
<tr>
<td>random forest</td>
<td>0.9333</td>
<td>0.8333</td>
<td>1.0000</td>
<td>0.9091</td>
</tr>
<tr>
<td>SVM</td>
<td>0.8400</td>
<td>0.6970</td>
<td>0.9200</td>
<td>0.7931</td>
</tr>
<tr>
<td>AGGREGATE_VOTE</td>
<td>0.8404</td>
<td>0.7019</td>
<td>0.9040</td>
<td>0.7902</td>
</tr>
<tr>
<td>AGGREGATE_ANY</td>
<td>0.8670</td>
<td>0.7193</td>
<td>0.9840</td>
<td>0.8311</td>
</tr>
</tbody>
</table>

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. All three classifiers show similar effectiveness in detecting XSS attacks.

<table>
<thead>
<tr>
<th></th>
<th>accuracy</th>
<th>precision</th>
<th>recall</th>
<th>fscore</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve bayes</td>
<td>0.8711</td>
<td>0.7211</td>
<td>1.0000</td>
<td>0.8380</td>
</tr>
<tr>
<td>random forest</td>
<td>0.8711</td>
<td>0.7211</td>
<td>1.0000</td>
<td>0.8380</td>
</tr>
<tr>
<td>SVM</td>
<td>0.8756</td>
<td>0.7282</td>
<td>1.0000</td>
<td>0.8427</td>
</tr>
<tr>
<td>AGGREGATE_VOTE</td>
<td>0.8722</td>
<td>0.7238</td>
<td>1.0000</td>
<td>0.8398</td>
</tr>
<tr>
<td>AGGREGATE_ANY</td>
<td>0.8634</td>
<td>0.7102</td>
<td>1.0000</td>
<td>0.8306</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th>accuracy</th>
<th>precision</th>
<th>recall</th>
<th>fscore</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve bayes</td>
<td>0.9167</td>
<td>0.9412</td>
<td>0.8000</td>
<td>0.8649</td>
</tr>
<tr>
<td>random forest</td>
<td>0.9333</td>
<td>1.0000</td>
<td>0.8000</td>
<td>0.8889</td>
</tr>
<tr>
<td>SVM</td>
<td>cell5</td>
<td>0.9333</td>
<td>1.0000</td>
<td>0.8889</td>
</tr>
<tr>
<td>AGGREGATE_VOTE</td>
<td>0.9333</td>
<td>1.0000</td>
<td>0.8000</td>
<td>0.8889</td>
</tr>
<tr>
<td>AGGREGATE_ANY</td>
<td>0.9167</td>
<td>0.9412</td>
<td>0.8000</td>
<td>0.8649</td>
</tr>
</tbody>
</table>

Table III

MACHINE LEARNING MODELS’ EXPERIMENTAL RESULTS FOR SQL INJECTION ATTACKS

V. 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. Livshits et al. [18] proposed a static analysis technique for detecting SQL
injection, cross-site scripting, and HTTP splitting attacks. Their system uses user-provided specifications of vulnerabilities to analyze code statically without code execution. Hafond et al. [19] combined conservative static analysis and run-time monitoring to detect SQL injection attacks.

Kemalis et al. [20] developed a prototype SQL injection detection system that utilizes specifications that define the intended syntactic structure of SQL queries and monitor Java-based applications and detect SQL injection attacks in real time. To detect and prevent SQL Injection attacks on web applications, Dharam et al. [21] presented and evaluated a run-time monitoring framework which 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. [22] applied machine learning techniques to learn the profiles of normal database access and detect SQL injection attacks. Sharma et al. [23] introduced 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.

VI. CONCLUDING REMARKS

This paper investigated the feasibility of creating an autonomous cyber-attack detection system capable of detecting attacks against modern applications running on the Java Virtual Machine. We described the Robust Software Modeling Tool (RSMT) that detects 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 and 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.

REFERENCES


