Automated Mining of Software Component Interactions for Self-Adaptation

Eric Yuan
Computer Science Dept.
George Mason University
eyuan@gmu.edu

Naeem Esfahani
Computer Science Dept.
George Mason University
nesfaha2@gmu.edu

Sam Malek
Computer Science Dept.
George Mason University
smalek@gmu.edu

ABSTRACT

A self-adaptive software system should be able to monitor and analyze its runtime behavior and make adaptation decisions accordingly to meet certain desirable objectives. Traditional software adaptation techniques and recent "models@runtime" approaches usually require an *a priori* model for a system's dynamic behavior. Oftentimes the model is difficult to define and labor-intensive to maintain, and tends to get out of date due to adaptation and architecture decay. We propose an alternative approach that does not require defining the system's behavior model beforehand, but instead involves mining software component interactions from system execution traces to build a probabilistic usage model, which is in turn used to analyze, plan, and execute adaptations. Our preliminary evaluation of the approach against an Emergency Deployment System shows that the associations mining model can be used to effectively address a variety of adaptation needs, including (1) safely applying dynamic changes to a running software system without creating inconsistencies, (2) identifying potentially malicious (abnormal) behavior for self-protection, and (3) our ongoing research on improving deployment of software components in a distributed setting for performance self-optimization.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Algorithms

Keywords

Data Mining, Self-Adaptation, Component-Based Software

1. INTRODUCTION

A self-adaptive software system is comprised of two conceptual parts, a *base-level subsystem* that provides the software system's application logic and domain functionalities and a *meta-level subsystem* that manages the behavior of the base-level subsystem to satisfy certain desirable objectives, e.g., performance, security, reliability, etc. Such management often takes the form of dynamically changing the structure of the base-level software, e.g., replacing software components at runtime.

To make proper decisions, the meta-level subsystem relies on an abstract representation of the software and the environment it executes. The collection of such models is often referred to as *models at runtime*, as they need to be kept in sync with the changes that unfold in a running system and its environment. An example of architectural models that is used extensively in the construction of adaptive software is *component interaction model*, which represents the behavior of the system's components in their interactions (e.g., message exchanges, interface invocations) with one another.

Component interaction models could be used for a variety of purposes in runtime management of software, including (1) determining the dependencies among the system's component to ensure their adaptation (e.g., replacement) does not leave the system in an inconsistent state [3], (2) detecting abnormal interactions among the system's components that are indicative of security attacks to enable self-protection capabilities, and (3) optimizing a software system's performance by collocating components that are highly interactive with one another [19]. The construction of such models, however, is a difficult task. First, in a complex software system, manually defining accurate models that truly represent all possible component interactions is time consuming. Second, it is not always possible to construct such models *a priori*, before the system's deployment. In service-oriented architectures (SOA) or peer-to-peer environments, for instance, component behavior may be user-driven and non-deterministic. Third, even when such models are built, it is a heavy burden to keep them in sync with the actual implementation of the software. Indeed, they are susceptible to the well-studied problem of architectural decay [29], which tends to occur when changes applied to the software are not reflected in its architecture models.

An approach toward addressing the above issues is to automatically mine such models form execution traces of the system, thus, alleviating the engineers from defining the models manually. Automated mining-based approaches also allow for their application throughout the system's execution, naturally enabling the refinement of models to changing behavior of the system and its environment.

This paper describes our experiences with an association

SEAMS '14, June 2-3, 2014, Hyderabad, India

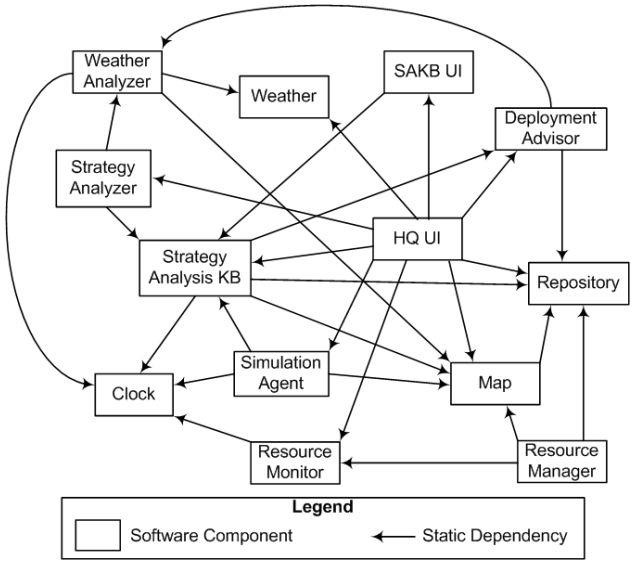

Figure 1: Subset of EDS Software Architecture

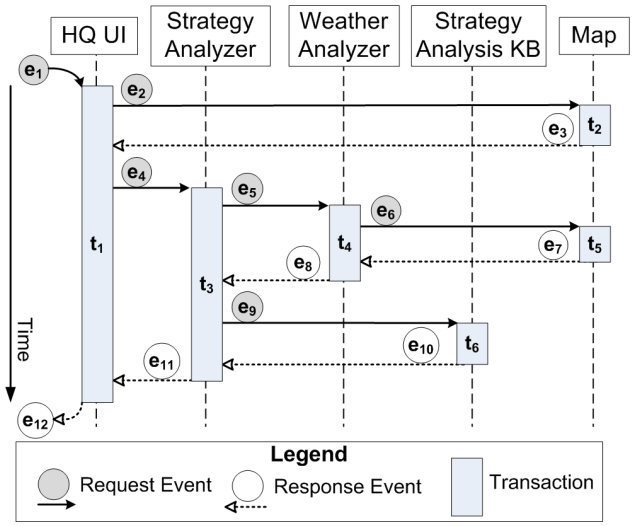

Figure 2: EDS Use Case Example

rule mining approach that can learn the component interaction model of a system by simply observing its behavior. The approach is comprised of three steps: (1) collect execution traces of the system at runtime, (2) use association rule mining to infer a probabilistic model of the component interactions from the collected execution traces, and (3) continuously monitor the accuracy of the inferred models, and upon detecting substantial variations, refine the models by mining the newly collected data.

The application of this overall approach to the construction of a self-adaptive emergency response system has shown to be quite promising. We describe how the component interaction models inferred using our approach can be effective in addressing a variety of adaptation needs. Although our experiences have been very promising, mining-based approaches, such as the one described in this paper, present their own unique challenges. Thus, we also provide an overview of these challenges to frame the future research.

The remainder of this paper is organized as follows. Section 2 introduces a distributed software system and its adaptation requirements to motivate the research. Section 3 provides an overview of our approach, while Section 4 describes the details. Sections 5 to 7 present applications of our approach in solving three types of concern in runtime management of software. Section 8 highlights related research. Finally, the paper concludes with a discussion of the remaining challenges and avenues of future research.

2. MOTIVATING EXAMPLE

We illustrate the concepts and evaluate the research using a software system, called Emergency Deployment System (EDS), which is intended for the deployment and management of personnel in emergency response scenarios. Figure 1 depicts a subset of EDS's software architecture, and in particular shows the dependency relationships among its components. EDS is used to accomplish four main tasks: (1) track the resources using Resource Monitor, (2) distribute resources to the rescue teams using Resource Manager, (3) analyze different deployment strategies using Strategy Analyzer, and finally (4) find the required steps toward a selected strategy using Deployment Advisor. EDS is representative of a large component-based software system, where the components communicate by exchanging messages (events). In the largest deployment of EDS to-date, it was deployed on 105 nodes and used by more than 100 users [20].

Like any software systems, the EDS functionality can be decomposed into a number of use cases. The sequence diagram for one such use case, conducting strategy analysis, is shown in Figure 2 as an example. We see that the execution of the use case involves a sequence of interactions among different software components. Note that a component here represents a coarsely grained software unit that deploys and runs independently from other components (in contrast to lower level entities such as a Java object or a code library). For instance, the "HQ UI" component is a web application that resides in a web server.

Once deployed and operational, a real-world system such as EDS needs to continually evolve to ensure quality, meet changing user requirements, and accommodate environment changes (such as hardware upgrades). At the same time, the system must also satisfy a number of architectural objectives such as availability, performance, reliability, and security.

Non-trivial system adaptations typically require an abstract representation of the components and their interactions at runtime, which can be used to formulate adaptation strategies and tactics [10]. In the case of EDS, a model such as Figure 2 could be used to reason about several adaptation concerns: (1) The model tells us when it is safe to adapt the components. For instance, as shown in [31], a model such as that of Figure 2 could be used to determine Strategy Analyzer can be safely adapted prior to event e_4 or after event e_{11}, but not in between, as its state is inconsistent. (2) The model could be used in the construction of self-protecting software to detect abnormal (malicious) behavior. For instance, assuming that the model of Figure 2 represents the only possible sequence of interaction among the components under this use case, one could determine a suspicious behavior when Strategy Analyzer interacts with a component it has not previously interacted with, such as Resource Manager. (3) The model could be used in the construction of self-optimizing software by changing the deployment of software, i.e., allocation of software components to the system's hardware nodes. For instance, as shown in [19], to reduce the

response time, components that interact frequently could be either collocated on the same hardware node or on nodes that have reliable and fast network connectivity.

Building and maintaining such a component interaction model, however, faces several difficult challenges, as outlined in Section 1. Our approach to addressing these challenges involves learning a *usage proximity* model of dynamic component interactions at runtime, without any pre-defined behavior specifications. Machine learning-based approaches alleviates engineers from maintaining the models manually, and also allow for their automatic adaptation and refinement to changing behavior of the system and its environment.

3. APPROACH OVERVIEW

We first start with some definitions and assumptions to frame the discussion of our mining framework. An event e is defined as a triple tuple $e =< src, dst, time >$, where *src* and *dst* are identifiers for the source and destination components, and *time* is the timestamp of its occurrence. Although an event is also likely to have a payload (e.g., a message in XML format), it is not relevant to this line of research, and thus not modeled. In the EDS example of Figure 2, 12 events (e_1-e_{12}) are depicted.

A transaction t is defined as a triple tuple $t =< start, end, R >$, where *start* and *end* respectively represent the events initiating and terminating the transaction t, while R is a set of transactions that subsequently occur as a result of t. $R \neq \emptyset$ when t is a dependent transaction (e.g., t_1, t_3, and t_4 in Figure 2), and $R = \emptyset$ when t is an independent transaction (e.g., t_2, t_5, and t_6 in Figure 2). For convenience sake, a transaction is also sometimes denoted $src \triangleright dst$ where *src* and *dst* are the source and destination components of t's initiating event.

A *top-level transaction* t is a kind of transaction where there is no other transaction x in the system such that $t \in x.R$. In other words, a transaction is top-level if its occurrence is not tied to other transactions in the system. A top-level transaction corresponds to the system's use cases (functional capabilities). For instance, t_1 in Figure 2 is a top-level transaction, initiated in response to e_1, which represents the user requesting a service from the system.

In this example we see that the components involved in a use case interact closely with one another. Given enough observations of the system at runtime, it is possible to infer the *stochastic component interaction model* of the system. Such a model not only infers the dynamic dependencies among the components (i.e., information equivalent to that captured in Figure 2), but it also provides a probabilistic measure of the certainty with which events and transactions may occur. Even though such a model is simplistic and by no means captures the complete and precise behavior of the system, it is surprisingly useful in addressing a number of adaptation objectives as we shall see in later sections.

To keep our approach widely applicable, we make minimal assumptions about the available information from the underlying system:

- *Black-Box Treatment*: We assume the software components' implementation is not available. This allows our approach to be applicable to systems that utilize services or COTS components, whose source code is not available. It also enables our approach to naturally support the evolution of software components.

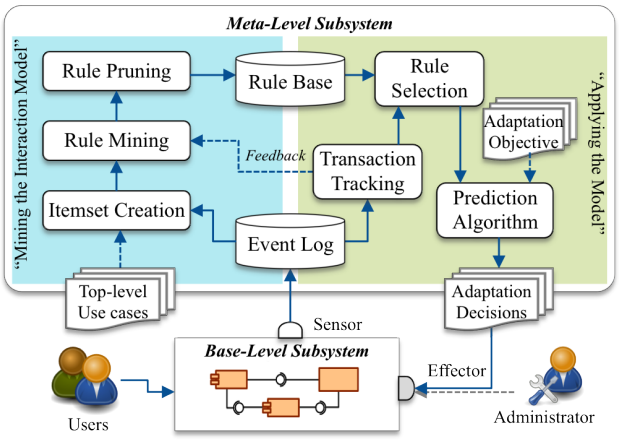

Figure 3: Overview of the mining framework

- *Observability of Event*: We assume that events marking the interactions among the system's components are observable. An event could be either a message exchange or a method call, which could be monitored via the middleware facilities that host the components or instrumentation of the communication links.
- *Observability of Transaction Duration*: We assume events *start* and *end*, which as you may recall indicate beginning and termination of a transaction, to be observable. This is a reasonable assumption consistent with several prior research approaches that have dealt with safely effecting runtime changes [14, 31, 18].
- *Use case-initiating events can be identified.* Here we assume that a number of "entry point" events exist that initiates top-level transactions. Such events typically represent the starting point of a system use case. An online banking system, for example, may have menu items such as "Withdrawal", "Deposit", or "Check Balance" that trigger different use cases. The EDS system, likewise, has client-server events (such as e_1 in Figure 2) that initiate different use cases.

With these assumptions, we proceed to define a novel approach for automatically deriving the stochastic component interaction model by mining the execution history of the software system. Figure 3 provides an overview of our approach, consisting of two complementary asynchronously running cycles: *Mining the Interaction Model* and *Applying the Model*.

The *Mining the Interaction Model* cycle starts by processing the *Event Log* of the system to construct a large number of *Itemsets*. An itemset indicates the events that occur close in time. Itemsets are then passed through a data mining algorithm to derive *Transaction Association Rules (TARs)* relating the relationship between transactions that are occurring in the system and those that may happen in the future. Since mining may generate a large number of rules, some of which may be invalid and redundant, we *prune* the generated rules to arrive at a small number of useful rules that can be applied efficiently at runtime. Section 4 describes the details of the mining process.

The *Applying the Model* cycle starts with the *Transaction Tracking* activity that monitors the currently running transactions in the system. *Rule Selection* then uses the information about currently active transactions to pick a set of candidate TARs from the *Rule Base* for estimating the usage

probability of components. Depending on different adaptation objectives, a *Prediction Algorithm* will be applied to make the desired adaptation decisions. In sections 5 to 7 we will explore three such algorithms.

4. MINING COMPONENT INTERACTIONS

4.1 Itemset Creation

The first step to mining the relationship among the transactions is to construct itemsets. An *itemset*, as in the data mining literature for association rule mining, is a set of items that have occurred together. In the context of our research, an itemset I is a set of transactions that have occurred temporally close to one another at some particular point during the execution of the system: $I = \{t_1, t_2, ..., t_n\}$.

By reading the start and end timestamps of the top-level transactions, we can easily keep track of the set of "active baskets" in the system, and incoming events will fall into one or more of them. When a basket closes, an itemset is produced and stored. In reference to Figure 2, a new itemset would be created for t_1, as its beginning and end (determined by e_1 and e_{12}) do not fall within any other transactions. All the remaining transactions t_2, t_3, t_4, t_5, and t_6 are added to I_{t1} itemset as follows: $I_{t1} = \{t_1, t_2, t_3, t_4, t_5, t_6\}$.

Using this process, an entire segment of a software system's execution history can be transformed into a set of itemsets representing the occurrence of transactions together in time. Given a sufficiently large usage history, the approach compensates for concurrently running top-level transactions. Consider a version of the scenario depicted in Figure 2 in which a second top-level transaction t_7 overlapping partially in time with t_1 starts and itself initiates a transaction t_8 that falls wholly within the beginning and end times of both t_1 and t_7. The approach will include t_8 in both I_{t1} and I_{t7}. However, since transactions t_1 and t_7 are truly independent, the false placement of t_8 in I_{t1} is a random event that is not likely to occur in a significantly large number of itemsets, and thus safely ignored by the data-mining algorithm using minimum frequency thresholds.

4.2 Association Rules Mining

Associations Mining, also known as Frequent Pattern Mining plays an essential role in discovering associations, correlations, and multi-dimensional patterns and have been used in many practical application such as marketing analysis and query recommendations [16]. The next step of our approach is to use the itemsets to generate transaction association rules (TAR). Each TAR is of the form

$$X \Rightarrow Y :< s, c >$$

where X and Y are sets of transactions, e.g. $X = \{t_i, t_j\}$, $Y = \{t_k\}$ and s and c are the support and confidence level of the rule, respectively. Note that

$$s = \sigma(X \cup Y)/N \quad (1)$$

$$c = \sigma(X \cup Y)/\sigma(X) \quad (2)$$

where $\sigma(S)$ is the frequency count of the co-occurrence of the events in the itemset S, and N is the total number of itemsets. As a concrete example, here is a rule generated in one of the test runs for the EDS system:

$\{HQUI \triangleright StrategyAnalyzer,$
$StrategyAnalyzer \triangleright WeatherAnalyzer,$
$StrategyAnalyzer \triangleright StrategyKB\}$
$\Rightarrow \{WeatherAnalyzer \triangleright Map\} :< 0.45, 0.76 >$

It is important to note that a TAR of the form $\{t_i, t_j\} \Rightarrow \{t_k\}$ does not represent a temporal execution sequence. Rather, as an association rule it simply states the proximity or likelihood of transaction t_k occurring together with transactions t_i and t_j in one itemset (that is, in the same top level use case).

Several association mining algorithms exist such as Apriori [1] and FP-Growth [12]. In our evaluations we primarily used the Apriori algorithm for TAR mining due to the fact that its implementations are mature and widely available.

4.3 Rule Base Pruning

Minimum support and confidence levels are the two key input parameters for an associations mining algorithm. When generating candidate rules, any rules whose support and confidence values fall below these levels will be discarded. In order to build a model that captures the entire range of "normal" system use, we learned that they need to be set at very low levels, which tend to produce an excessively large number of rules. Further, many generated TARs are extraneous and unnecessary, in part due to the fact that we have chosen to ignore the temporal information associated with the event sequences and focus instead on the proximity of event co-occurrence. Therefore, pruning the rule base to reduce its size becomes a critical component of our approach.

Fortunately, depending on the particular adaptation scenario and architecture objective, a number of heuristics can often come to assistance to drastically reduce the rule base size. Interested readers may find some examples in [3]. The rule mining and pruning can be repeated for every N events in the event log to make sure the model is kept up to date with the latest system behavior.

Once the association rules are mined and pruned, we turn our attention to how they can be used to address self-adaption needs. The following sections demonstrates three application scenarios. Section 5 employs the model to dynamically predict what components can be safely adapted without compromising system consistency. Section 6 aims to enable self-protection of the system against malicious threats at the application level by detecting anomalous behavior patterns. Section 7 focuses on the self-optimization perspective and discusses how the rule base can better inform and adjust system deployment topology to reduce network latency and improve performance.

5. SAFE COMPONENT ADAPTATION

5.1 Background

Replacing a component in the middle of a transaction could place the system in an inconsistent state. Consider a situation in which *Strategy Analyzer* component of Figure 2 is replaced after sending request event e_5, but before receiving the response event e_8. Since the newly installed component does not have the same state as the old one, it may not be able to handle response e_8 and subsequently initiate transaction t_6 via event e_9, resulting in an inconsistency and potentially the system's failure. Three general approaches to this problem have been proposed: *quiescence*, *tranquility*, and *version-consistency*.

Quiescence [14] is the established approach for safe adaptation of a system. A component is in quiescence and can be

adapted if (1) it is not *active*, meaning it is not participating in any transaction, and (2) all of the components that may initiate transactions requiring services of that component are passivated. A component is *passive* if it continues to receive and process transactions, but does not initiate any new ones. At runtime, the decision about which part of the system should be *passivated* is made using a *static component interaction model*, such as that shown in Figure 1. For instance, to change the *Map* component, on top of passivating itself, *Weather Analyzer*, *Strategy Analysis KB*, *HQ UI*, *Simulation Agent*, and *Resource Manager* components need to be passivated as well, since those are the components that may initiate a transaction on *Map*.

While quiescence provides consistency guarantees, it is very pessimistic in its analysis and, therefore, sometimes very disruptive. Consider that the static interaction model includes all possible dependencies among the system's components, while at any point in the execution of a software system only some of those dependencies take effect. To address this issue, tranquility [31] proposes to use the *dynamic component interaction model* of a system in its analysis, an example of which is shown in Figure 2. Under tranquility *a component can be replaced within a transaction as long as it has not already participated in a transaction that it may participate in again.* For instance, under tranquility, *Map* could be replaced either before it receives event e_2 or after it sends event e_7, but not in between.

A shortcoming of tranquility, as realized in [31], was lack of support for handling dependent transactions. This issue was addressed in version-consistency [18], which guarantees a dependent transaction is served by either the old version or new version of a component that is being changed.

5.2 Applying Association Rules

Any observed event indicates the beginning and termination of a given transaction t_o (recall Section 3). Therefore, we can refer to these events as $t_o.start$ and $t_o.end$, respectively. We use a data structure, called *top-level tracker*, and represented as set *TLT*, to track each top-level transaction that is active (i.e., currently running) in the system. The purpose of TLTs is to keep account of the present transaction activity in the system. Upon observing $t_o.start$, the state of TLTs is updated as follows. If t_o is a top-level transaction, a new TLT is created. But if t_o is not a top-level transaction, its identifier is added to all open TLTs. This is done because there is no way of knowing which top-level transaction has actually initiated this transaction. Upon observing $t_o.end$, if t_o is not a top-level transaction, it is ignored. On the other hand, if t_o is a top-level transaction, then the TLT corresponding to t_o is closed.

The updated TLTs are used to determine what new predictions can be made about the probability with which components will be used. All predictions of the system activity are made by using the TARs stored in the Rule Base. We must determine what new TARs, if any, are implicated by the observation of $t_o.start$. A $tar \in RuleBase$ can only be implicated by the observation of $t_o.start$, if t_o is a member of set X of that tar. If this criterion is met, then we look to see if the tar is satisfied by any open top-level transaction as tracked by TLTs. For a tar to be satisfied, all transactions in X must have been observed during the processing of at least one TLT (*basis* of that tar). Furthermore, the tar's prediction (i.e., Y) should have new transactions other than the ones that have already occurred during the processing of the satisfying TLT. If both of these conditions are met, then the tar is added to the set set of all new TARs that are candidates for being applied at that given point in time.

The next step is to apply the implicated TARs to update the *usage prediction registries*, represented as set UP. Each component q in the system has a register $u_q \in UP$. The value of u_q represents the probability of q being used or becoming active in near future. There are typically more than a single TAR predicting usage probability $u_q \in UP$ of any given component q. In other words, component q may appear in the predictions (i.e., Y) of several satisfied TARs. While some may be due to the new observation $t_o.start$, others may be due to the prior observations.

Recall from Section 5.1 that a component can be safely adapted as long as it has not already participated in a transaction that it may participate in again. When component q is already used in a transaction we don't consider any of the satisfied TARs and let $u_q = 1$, and hence, guarantee the consistency of adaptation. Otherwise, we combine the various confidence values from all of the satisfied TARs into a single prediction value u_q.

The insight guiding our research in this step is that each TLT can make a single prediction for a component. To that end, for each TLT, we select the TAR with the highest confidence value (i.e., c value of Equation 2) among all the TARs, which have that TLT as their basis. We then calculate u_q by combining the prediction values of all *selected* TARs for all active TLTs in the system. Since we have only selected a single TAR from each TLT, we can treat the predictions as independent probabilities when combining them to arrive at a final prediction value for the component:

$u_q = 1 - \textit{prob. } q \textit{ is not used} = 1 - \prod_{i=1}^{|\textit{selected TARs}|}(1 - c_i)$

The observation of $t_o.end$ can also cause an update. If $t_o.end$ is a top-level transaction, the TLT corresponding to $t_o.end$ is removed. As a result, all the TARs that have that TLT as their only basis should be removed as well. We simply recalculate u_q after removing those TARs from the set of *selected* TARs as we described earlier.

When we want to adapt component q, we refer to $u_q \in UP$. If u_q is below a certain threshold ϵ (where $\epsilon \leq 1$), we will allow component q to be adapted. We only use our predictions to reduce the disruptions experienced in the system before a component is used in a transaction (since $u_q = 1$, when q is already used, we will not adapt q). Of course, one could relax this constraints to reach faster adaptation times, while trading off the safety guarantees. Interested readers are referred to our recent publication [3] on how we select the right value of ϵ to balance between such trade offs.

5.3 Evaluation

We have developed a prototype of the approach using an implementation of Apriori provided in WEKA [11]. We performed experimentation on runtime adaptation of EDS (recall Section 2). To evaluate the approach, we used several versions of EDS with different concurrency levels [3]. We used a baseline version of EDS with a single user. We then repeated the evaluations on higher concurrency systems to evaluate the susceptibility of the approach to concurrency errors. The 80 and 137 experiments were simulated by using hyperactive dummy users, as EDS never naturally reached that level of concurrency error. Therefore, the values for users are merely projections, and the precise values for con-

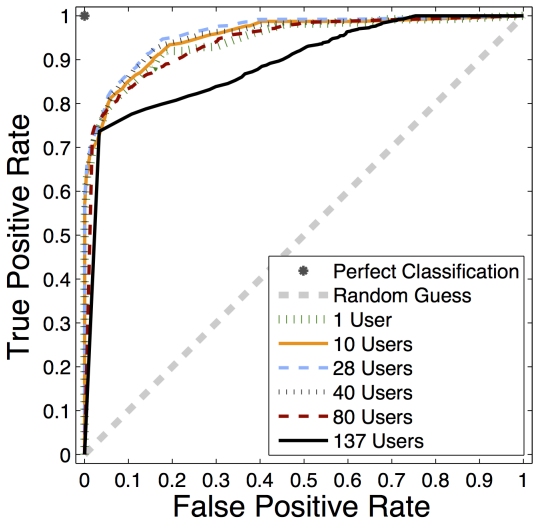

Figure 4: ROC Curves for Different Concurrency Settings in Adaptation Problem

currency error rate should receive primary focus.

The quality of differentiating active and inactive components can be viewed with a *receiver operating characteristic (ROC) curve*, often used to evaluate a binary classifier, as shown in Figure 4. In our case, the ROC curve depicts the change in the ratio of True Positive (TP) to False Positive (FP) as different ϵ thresholds are chosen. The extreme of $\epsilon = 1.0$ exists at the origin of the ROC plot, while the extreme of $\epsilon = 0.0$ exists at the point (1 , 1) of the ROC plot. Therefore, it can be seen how the TP and FP rates respond by moving the ϵ threshold. The ROC curve shows that the approach does an incredible job of achieving true positives despite changes in the ϵ threshold.

The comparison of the different experiments also shows the effect of concurrency on the approach. With many users in the system, there are many more observations that allow the approach to predict usage of a component, when the component is actually used. Therefore, as concurrency increases, the approach keeps the high quality in differentiating active and inactive components. However, when we approach 137 users, the concurrency error rate is roughly 60% and active components are constantly at $u = 1.0$ until the transactions they participate in subside. This concurrency rate forces $\epsilon = 1.0$ to avoid high disruption in the system. At this point, ϵ reaches its maximum value, and hence, cannot compensate for the increase in FP rate by moving to a higher value.

6. DETECTING ANOMALOUS BEHAVIOR FOR SELF-PROTECTION

6.1 Background

As modern software systems become increasingly modular, distributed and interactive, they are also facing unprecedented security challenges, especially under new computing paradigms such as mobile and cloud computing. prone to cyber attacks. Conventional techniques for securing software systems, often manually developed and statically employed, are therefore no longer sufficient. This has motivated active research in dynamic and adaptive security approaches [5]. In particular, active research has focused on self-protecting software systems, a class of systems capable of autonomously defending itself against security threats at runtime [35].

The first step towards autonomic and responsive security is the timely and accurate detection of security compromises and software vulnerabilities at runtime, which is a daunting task in its own right. Data mining techniques have been widely applied in this regard, however most security-oriented data mining research to-date has focused on "lower layers" of a software system in an architectural sense, that is, mining data at network, host machine, or source code levels. As a result, such approaches mainly address specific types of threats that are tactical in nature, but the "big picture" understanding of attacker strategy and intent, as well as overall security posture of the system appears to be lacking. Furthermore, these approaches typically can do very little to address the growing concern of insider threats, where attackers use the system with legitimate credentials instead of external intrusions [26].

In contrast, our research has focused on developing a threat detection approach based on *software component interactions* as opposed to mining data collected from network traffic or source code. Our underlying insight is that many cyber attacks *misuse* the system in a way that deviates from normal system behavior. Take the EDS system for instance, since it is an online system that manages sensitive information such as personnel records and locations, it may be subject to various intrusions and exploits including SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF), just to name a few [24]. As a concrete example, suppose an attacker obtains a valid user login and hijacks the Strategy Analyzer component through the Headquarters User Interface (HQUI) component. Instead of calling *Weather Analyzer* and *Strategy Analysis KB* components as prescribed in the use case in Figure 2, the attacker sends requests from Strategy Analyzer to the *Resource Manager* and *Repository* components to retrieve sensitive information about all deployed resources, as shown in Figure 5. Such a violation of system usage occurs at the application level and is therefore much harder to detect and thwart using conventional firewalls and intrusion detection devices, which are primarily concerned with ports and protocols.

To be able to effectively detect potentially malicious behavior at the application level, there are two main approaches, *signature-based* or *anomaly-based*. Signature-based techniques attempt to capture the signatures or specifications of attacks as the basis for detection, which are usually very accurate and efficient but require constant maintenance as attack strategies and tactics evolve ever so rapidly. Nor can this approach detect unknown threats. Anomaly-based approach, on the other hand, seeks to build a "normal" system usage model as the basis for threat detection. Our research shows that the mining framework introduced in Section 3 can be used as such a model that is efficient and effective against both known and unknown threats; as shown in preliminary evaluation results.

6.2 Anomaly Detection Based on Association Rules

After association rules are generated from the mining phase of the framework as described in Section 4, we are not quite ready to apply them directly to detect anomalous behavior from the system's event execution streams. In fact, this problem scenario is the opposite of typical uses

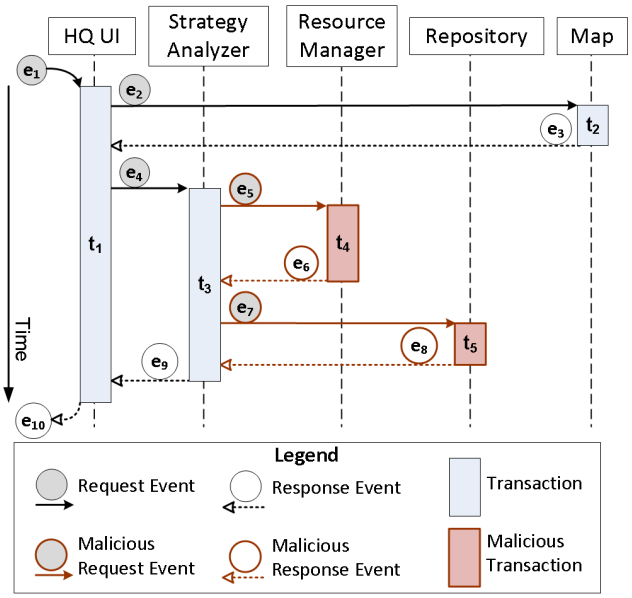

Figure 5: Example attack scenario

of association rules – instead of using them to predict what item(s) are frequently associated with a given set of items, the question we ask here is "is this transaction *infrequently* associated with the given itemset of transactions, so much as to warrant an alarm?" To this end, we developed an effective detection algorithm that is roughly outlined as follows:

For an itemset containing a top level transaction T and a set of child transactions $t_1, t_2, ..., t_n$, by observing the start and end times of the messages, we can find all the Enclosing Transaction Sequences (ETS) $< x_1, ..., x_{j-1}, x_j >$ such that each transaction is the child of the preceding transaction. In Figure 2, for example, $< t_1, t_2 >$, $< t_1, t_3, t_4, t_5 >$ and $< t_1, t_3, t_6 >$ are such sequences. For normal use cases, all such sequences should have been captured in the rule base after we have observed enough number of transactions from the event log. For malicious uses of the system, however, some of the ETS such as $< t_1, t_3, t_4 >$ and $< t_1, t_3, t_5 >$ in Figure 5 occur at a much lower frequency and thus not found in the rule base. Each not-found ETS will be marked as a violation, and we can mark the itemset as anomalous when the number of violations reach a certain threshold.

6.3 Evaluation

Our experimentation environment involves a customized instance of the original EDS system in a similar setup as introduced in the previous section. In addition to the normal system use cases, we injected the attack scenario outlined in Figure 5 to the simulation runs according to a predefined anomaly probability, which is set at $\sim 0.3\%$ ($3\,\sigma$ or standard deviations of a normal distribution) under the assumption that covert malicious attacks are rare events.

We use the same Apriori implementation from WEKA for association rule generation. Both simulation and data analysis are run on a quad-core Mac OS X machine. Table 1 shows our experimental results with different number of users and with the detection algorithm configured to run with different support and confidence levels.

Even though there is still room for improvement, these results show that our approach can help detect anomalous use of the system in an automated, unattended fashion, with high recall and reasonable precision. In the 5-user scenario, for example, our framework detects 88% of the anomalous events with a 50% precision. Instead of manually inspecting 25,000 user sessions with roughly 114,000 total transactions, a security administrator only needs to inspect the 127 alarms, 64 of which are true anomalies that need to be addressed. This demonstrates that our approach can be used as an effective mechanism to enhance both overall system security and security administrator productivity. It is worth noting that the concurrent users in our simulation runs are "intense" users used to generate a heavy load on the system, therefore *they actually represent a much larger number of human users in a real-world system.*

Table 1: Detection Performance for 1-User, 5-User, and 10-User Scenarios

#Concurrent Users	1	5	10
Min. Support Level	0.025	0.01	0.025
Min. Confidence Level	0.2	0.15	0.1
TP Count	16	64	130
FP Count	0	63	1136
FN Count	0	9	27
TN Count	4,984	24,926	48,707
TP Rate (TPR)	1.0	0.877	0.828
FP Rate (FPR)	0.0	0.003	0.023
Precision	1.0	0.504	0.103
Recall	1.0	0.877	0.828
F-Measure	1.0	0.64	0.183

$TPR = TP/(TP + FN); FPR = FP/(FP + TN)$
$Precision = TP/(TP + FP); Recall = TPR$
$F\text{-}Measure = 2TP/(2TP + FP + FN)$

(5000 itemsets / use cases per user)

The results also show, however, that detection accuracy (esp. precision) is impacted by system concurrency. From the TPR-FPR plots (a.k.a., ROC curves) shown in Figure 6, we see a reduced Area Under the Curve (AUC) as the number of concurrent users increases. To address this issue, we have developed a concurrency measure γ as the average number of active top-level transactions under which a transaction t falls, which is easily measurable from system event log and is more objective than the number of active users. We are in the process of devising a "floating" detection threshold based on γ to mitigate the concurrency effects.

7. SELF-OPTIMIZATION OF DEPLOYMENT TOPOLOGY

7.1 Background

The design and development of large-scale, component-based software systems today are influenced by modern software engineering practices as embodied by architecture styles (e.g., pipe and filter), design patterns (e.g. proxies), and coding paradigms (e.g. aspect-orientation). A direct consequence is that the deployment of such systems becomes more complex and fluid, with hundreds or perhaps even thousands of options and parameters to consider, along dimensions such as location, capacity, timing, sequencing, service levels, security, etc. Many of them may be interdependent and possibly conflicting. Due to the combinatorially large problem space, the values of these parameters are usually set and fine-tuned manually by experts, based on rules

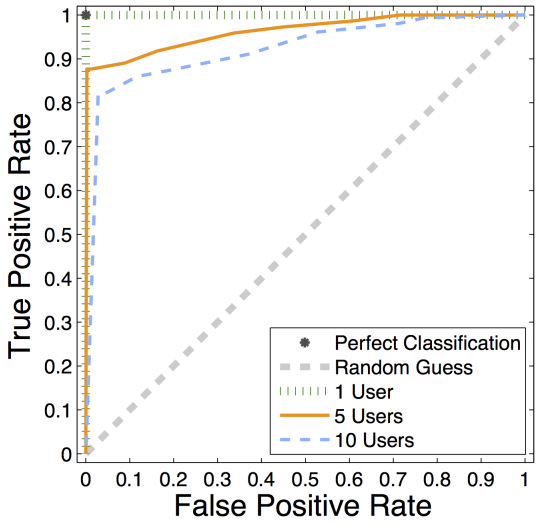

Figure 6: ROC Curves for Different Concurrency Settings in Anomaly Detection Problem

of thumb and experience.

An objective for autonomic systems is therefore to intelligently navigate the solution space and seek ways to optimally (re)deploy the system to continuously improve the overall system performance and cost [13].

To illustrate the self-optimization challenge, we turn our attention to the deployment topology of the EDS system. As a geographically distributed system, some of the components such as HQUI (recall Figure 1) needs to reside at the headquarters facility, while some such as the Resource Monitor are required to be at remote sites to be collocated with emergency response equipment. Other components are more flexible and can be deployed at either locations. Depending on the topology, inter-component messages can be either local (via inter-process communication on a single computer or over a LAN), or remote over a WAN, with the latter having a much larger network latency. Take the strategy analysis use case outlined in Figure 2 for example, if the *Strategy Analyzer* and *Strategy Analyzer KB* components reside at different sites, transaction t_6 may take a much longer time than what it would be if the two components were collocated, adversely affecting the response time experienced by the end user. Obviously, the system should employ a deployment topology that minimizes remote transactions to reduce overall network latency, subject to other constraints.

It is worth noting that this is by no means a new problem, and has been manifested in various settings such as system resource management [25], cloud performance optimization [4], wireless network configuration [21], etc. However, traditional approaches, including our own prior work [21, 19], assume the availability of a detailed component interaction model, that includes information about the component dependencies, frequency of interactions among the components, size of exchanged data, processing sequences, etc. As pointed out earlier in the paper, such a model is difficult to come by and costly to maintain.

Our proposed framework, on the other hand, leverages the same component interaction model for dynamic optimization of the deployment topology at runtime, a model that needs no prior development and can stay up to date even when the system behavior shifts.

7.2 Applying Association Rules

In essence, the transaction association rules produced from the mining process is a lookup table that provides the following information:

- For each single transaction type t, the probability of its occurrence, estimated by its frequency count during the past N transactions. (Note that a single transaction can be viewed as a TAR of the form $X \Rightarrow Y$ where X is the empty set ϕ and $Y = \{t\}$)
- For any set of transaction types $\{t_1, t_2, ..., t_3\}$, the probability of their *co-occurrence*, as indicated by the support level of the TARs.

It is easy to see that the problem of determining deployment topology, namely, assigning component c_i to location S_j, can be framed as a clustering problem. Intuitively, transactions that have a higher probability of occurring should be local (i.e., in the same cluster), and a set of transactions often happen in conjunction should be local as well. More formally, we can have the following heuristics for clustering components:

1. For any single transaction $t \equiv src \triangleright dst$, the higher probability of its occurrence, the more likely that placing components *src* and *dst* in the same cluster will reduce network latency;
2. For any Enclosing Transaction Sequences (ETS) $< t_1, t_2, ..., t_n >$ as defined in Section 6.2, the higher probability of their co-occurrence, the more likely that placing their underlying components in the same cluster will reduce network latency.

Now we see that our component interaction model captured from associations mining can be used to provide a *probabilistic distance measure* for any two components or two sets of components, which can be used by any clustering algorithm to compute the deployment topology. In particular, given a proximity matrix between the components derived from the rule base, agglomerative hierarchical clustering [28] can be used to quickly assign components to the desired number of sites. We are actively implementing the clustering algorithm and evaluating its effectiveness in improving EDS deployment topology.

Note that in real-world systems, optimal deployment of resources depends on many other factors besides network latency, such as hardware capacity at each location, cost of component migration or re-deployment, and so on. A more holistic approach needs to formulate a higher-level objective function that weighs benefits against various costs and constraints (e.g., as developed in [2] and [19]). In that case, the component-wise probabilistic proximity measure from our model can become an input to the larger optimization algorithm.

8. RELATED WORK

Researchers have used log of event data collected from a system to construct a model of it for various purposes. Cook et al. [6] use the event data generated by a software process to discover the formal sequential model of that process. In a subsequent work [7], they have extended their work to use the event traces for a concurrent system to build a concurrency model of it. Gaaloul et al. [8] discover

the implicit orchestration protocol behind a set of web services through structural web service mining of the event logs and express them explicitly in terms of BPEL. Motahari-Nezhad et al. [23] present an algorithmic approach for correlating individual events, which are scattered across several systems and data sources, semi-automatically. They use these correlations to find the events that belong to the same business process execution instance. Wen et al. [32] use the start and end of transactions from the event log to build petri-nets corresponding to the processes of the system. To our knowledge, except our recent work [3], no previous work has used mining of execution log to understand the dynamic behavior of the system for the purpose of self-adaptation.

As mentioned earlier in this paper, even though data mining techniques have been extensively used in the security arena for decades, most of the research has centered around (a) intrusion detection, esp. at network and host levels (e.g., [15]) and (b) malware/virus detection at source code and executable level (e.g., [27]). Ongoing research also has also focused on other security problems such as detecting remote exploits of web applications [22] and peer-to-peer botnet attacks [30]. Few research, has focused on detecting malicious behavior at the architecture/component level, and none that employs data mining techniques. We believe detecting malicious behavior at the architectural-level is a prerequisite for developing self-protection mechanisms that modify the system's architecture to mitigate the security threats.

Data mining techniques are increasingly applied in the software engineering domain to improve software productivity and quality [34]. The datasets of interest includes execution sequences, call graphs, and text (such as bug reports and software documentation). One body of research, for instance, focuses on mining software specifications — frequent patterns that occur in execution traces [17], which is similar to our problem but the focus is on mining API call usages for purposes such as bug detection, not for self-adaptation; their techniques (such as libSVM) are also different.

9. DISCUSSION AND FUTURE WORK

The underlying assumption in the current version of our framework is that a single data mining algorithm can process all the events/transactions in the system and build the stochastic component interaction models. This may not be possible, especially when we consider distributed software systems that permeate boundaries of several enterprises. An enterprise may be unwilling to share its internal structure and event logs with an entity that is out of its control for various reasons (e.g., protecting competitive edge, security concerns, etc.). Therefore, we are working on a distributed version of our approach, which achieves the same goal by running multiple local data mining algorithms. In fact, initial results show that confining the algorithms to the boundaries of enterprises improves the precision, not to mention scalability. The natural structure imposed by the boundary of an enterprise only allows certain components to talk to the outside world (i.e., other enterprises). This knowledge helps to reduce the concurrency error significantly.

Regardless of the adaptation needs, the proposed mining framework needs to be highly efficient and as scalable as the base subsystem itself in order to process the system execution traces as they occur and provide dynamic, near-real time predictions. For this reason we plan to conduct in-depth analysis of the computational characteristics of association mining algorithms and ideally leverage elastic, on-demand computing platforms (e.g. MapReduce) to speed up the mining framework performance.

The data mining algorithm that we used to build the stochastic component interaction models is based on set theory. Therefore, it is not able to leverage the frequency of event occurrences nor the temporal ordering among events, which are already available in the execution log of the system. We believe using these extra information can increase the accuracy of the inferred models, and in turn, make our approach more precise. Hence, we are studying the application of other types of data mining algorithms (e.g., sequential pattern mining [28]) that can use the extra information.

The accuracy of mined rules depends on the availability of a sufficiently large usage history of the software, exercising the interactions among the system's component. Such data could either be collected through benchmark of the system or its previous deployments. However, determining how much data is needed to allow for generation of accurate rules is challenging. The notion of component interaction coverage metric [33] provides a good starting point in addressing this issue. In addition, we plan to investigate how this approach would work in a "cold-start" mode, i.e., when a system is initially launched. One solution would be to start off with pessimistic (conservative) predictions, until actual usage patterns are learned. In addition, we plan to explore the use of data stream mining [9] in this context, which allows for the mining to be performed incrementally using the real-time stream of observations from the system.

Our ongoing research for anomalous behavior detection focuses on more extensive evaluation of the detection algorithm to enhance its accuracy and robustness. In particular, we plan to develop a more useful likelihood-based measure, so that the anomalous events can be tagged not just by a simple yes/no flag, but by a quantitative confidence level. Equally important is the enhancement of the algorithm to cope with heightened levels of system concurrency, as indicated in Figure 6. Further, we will also evaluate the algorithm's sensitivity against different input parameters (e.g. minimum support and confidence levels) to better understand its "sweet spots" and limitations. Last but not the least, we seek to prove that our anomaly-based approach is effective against unknown threats, which we hypothesized in Section 6.1.

10. ACKNOWLEDGMENTS

The authors would like to acknowledge Kyle Canavera for his contributions to the development of the technique for reasoning about safety of adaptation. This work was supported in part by awards CCF-1252644 from the US National Science Foundation, W911NF-09-1-0273 from the US Army Research Office, and D11AP00282 from the US Defense Advanced Research Projects Agency.

11. REFERENCES

[1] Agrawal, R., and Srikant, R. Fast algorithms for mining association rules in large databases. In *20th International Conference on Very Large Data Bases* (1994), Morgan Kaufmann, Los Altos, CA, pp. 478–499.

[2] Bobroff, N., Kochut, A., and Beaty, K. Dynamic placement of virtual machines for managing SLA

violations. In *10th IFIP/IEEE International Symposium on Integrated Network Management, 2007. IM '07* (2007), pp. 119–128.

[3] Canavera, K. R., Esfahani, N., and Malek, S. Mining the execution history of a software system to infer the best time for its adaptation. In *20th International Symposium on the Foundations of Software Engineering* (Nov. 2012).

[4] Casalicchio, E., Menascé, D. A., and Aldhalaan, A. Autonomic resource provisioning in cloud systems with availability goals. In *ACM Cloud and Autonomic Computing Conference (CAC)* (2013), p. 1.

[5] Chess, D. M., Palmer, C. C., and White, S. R. Security in an autonomic computing environment. *IBM Systems Journal 42*, 1 (2003), 107–118.

[6] Cook, J. E., and Wolf, A. L. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol. 7*, 3 (July 1998), 215–249.

[7] Cook, J. E., and Wolf, A. L. Event-based detection of concurrency. In *Int'l Symp. on the Foundations of Software Engineering* (Lake Buena Vista, Florida, Nov. 1998), pp. 35–45.

[8] Gaaloul, W., Baina, K., and Godart, C. Log-based mining techniques applied to web service composition reengineering. *Service Oriented Computing and Applications 2*, 2-3 (May 2008), 93–110.

[9] Gaber, M. M., Zaslavsky, A., and Krishnaswamy, S. Mining data streams: A review. *SIGMOD Rec. 34*, 2 (June 2005), 18–26.

[10] Garlan, D. et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer 37*, 10 (Oct. 2004), 46–54.

[11] Hall, M. et al. The weka data mining software: an update. *SIGKDD Explor. Newsl. 11*, 1 (Nov. 2009), 10–18.

[12] Han, J., Pei, J., and Yin, Y. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record* (2000), vol. 29, ACM, pp. 1–12.

[13] Kephart, J., and Chess, D. The vision of autonomic computing. *Computer 36*, 1 (Jan. 2003), 41–50.

[14] Kramer, J., and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng. 16*, 11 (Nov. 1990), 1293–1306.

[15] Lee, W., Stolfo, S., and Mok, K. A data mining framework for building intrusion detection models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999* (1999), pp. 120–132.

[16] Li, H. et al. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems* (2008), ACM, pp. 107–114.

[17] Lo, D. et al. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2009), KDD '09, ACM, pp. 557–566.

[18] Ma, X. et al. Version-consistent dynamic reconfiguration of component-based distributed systems. In *Int'l Symp. on the Foundations of Software Engineering* (Szeged, Hungary, Sept. 2011), ACM, pp. 245–255.

[19] Malek, S., Medvidovic, N., and Mikic-Rakic, M. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering 38*, 1 (Feb. 2012), 73–100.

[20] Malek, S., Mikic-Rakic, M., and Medvidovic, N. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering 31*, 3 (Mar. 2005), 256–272.

[21] Malek, S. et al. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In *Int'l Conf. on Software Engineering* (Minneapolis, Minnesota, May 2007), pp. 591–601.

[22] Masud, M. et al. Detecting remote exploits using data mining. *Advances in Digital Forensics IV* (2008), 177–189.

[23] Motahari-Nezhad, H. R. et al. Event correlation for process discovery from web service interaction logs. *The VLDB Journal 20*, 3 (June 2011), 417–444.

[24] OWASP.org. Owasp top ten project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[25] Poladian, V. et al. Dynamic configuration of resource-aware services. In *Int'l Conf. on Software Engineering* (Scotland, UK, May 2004), pp. 604–613.

[26] Salem, M. B., Hershkop, S., and Stolfo, S. J. A survey of insider attack detection research. In *Insider Attack and Cyber Security*. Springer, 2008, pp. 69–90.

[27] Schultz, M. et al. Data mining methods for detection of new malicious executables. In *2001 IEEE Symposium on Security and Privacy, 2001. S P 2001. Proceedings* (2001), pp. 38–49.

[28] Tan, P.-N., Steinbach, M., and Kumar, V. *Introduction to data mining*. Addison Wesley, 2005.

[29] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[30] Thuraisingham, B. et al. Data mining for security applications. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2008. EUC '08* (Dec. 2008), vol. 2, pp. 585–589.

[31] Vandewoude, Y. et al. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng. 33*, 12 (Dec. 2007), 856–868.

[32] Wen, L. et al. A novel approach for process mining based on event types. *J. Intell. Inf. Syst. 32*, 2 (Apr. 2009), 163–190.

[33] Williams, A., and Probert, R. A measure for component interaction test coverage. In *Computer Systems and Applications, ACS/IEEE International Conference on. 2001* (2001), pp. 304–311.

[34] Xie, T. et al. Data mining for software engineering. *Computer 42*, 8 (Aug. 2009), 55–62.

[35] Yuan, E., Esfahani, N., and Malek, S. A systematic survey of self-protecting software systems. *ACM Trans. Auton. Adapt. Syst. 8*, 4 (Jan. 2014), 17:1–17:41.