Mining Software Component Interactions to Detect Security Threats at the Architectural Level

Eric Yuan
Department of Computer Science
George Mason University
Fairfax, Virginia, USA
eyuan@gmu.edu

Sam Malek
Department of Informatics
University of California, Irvine
Irvine, California, USA
malek@uci.edu

***Abstract*—Conventional security mechanisms at network, host, and source code levels are no longer sufficient in detecting and responding to increasingly dynamic and sophisticated cyber threats today. Detecting anomalous behavior at the architectural level can help better explain the intent of the threat and strengthen overall system security posture. To that end, we present a framework that mines software component interactions from system execution history and applies a detection algorithm to identify anomalous behavior. The framework uses unsupervised learning at runtime, can perform fast anomaly detection "on the fly", and can quickly adapt to system load fluctuations and user behavior shifts. Our evaluation of the approach against a real Emergency Deployment System has demonstrated very promising results, showing the framework can effectively detect covert attacks, including insider threats, that may be easily missed by traditional intrusion detection methods.**

***Index Terms*—Data Mining, Security, Software Architecture**

I. Introduction

Despite significant progress made in computer security over the past few decades, the challenges posed by cyberthreats are more prevalent than ever before. Many well-publicized incidents point to the agile, deliberate, and persistent nature of cyber attacks today. Conventional techniques for securing software systems, often manually developed and statically employed, are therefore no longer sufficient. This has motivated active research in dynamic, adaptive security approaches [1].

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, ranging from mining network traffic for intrusion detections to finding recurring vulnerabilities across multiple software codebases (see Section IX for details). However, the vast majority of security research to-date, including those using data mining, have focused on lower layers of a software system, that is, mining data at network, host machine, or source code levels. As a result, such approaches mainly address specific signatures or categories of threats that are *tactical* in nature, but the "big picture" understanding of attacker *strategy* and intent appears to be lacking, as argued in the authors' recent survey [2]. 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 [3].

In this paper, we present an innovative data mining approach for detecting anomalous behavior from interaction among software components at the *architectural* level. With the architecture-based approach, detection and mitigation of security threats are informed by an abstract representation of software that is kept in sync with the running system. The architectural model allows the system to monitor "macro-level" system properties (i.e., abstract components and their interfaces) and reason about the global impact of a potential security breach, as opposed to examining lower-level metrics such as system calls and network packets. Anomalous patterns at the application level can be much stronger clues for a human administrator or an autonomous agent to understand the *intent* of the malicious attack and deploy appropriate countermeasures. As such, our approach does not seek to replace existing security mechanisms such as network- and host-based Intrusion Detection Systems (IDS), but rather to complement them and achieve defense in depth.

Our proposed mining framework, dubbed **Ar**chitectural-level **M**ining **o**f **U**ndesired behavio**r** (ARMOUR), operates under the premise that by observing the execution traces of the system for component interaction events at runtime, it is possible to build a model consisting of *event association patterns* that approximates normal system behavior. The model is then used in real-time to determine the likelihood of software component(s) being potentially ill-used. To build the model, the ARMOUR framework employs Generalized Sequential Pattern (GSP) Mining [4] as a machine learning technique, but tailors it in novel ways such as incorporating knowledge of system use cases and components. The framework then uses an efficient and adaptive algorithm that applies the model for anomaly detection.

The remainder of paper is organized as follows. Section II introduces our reference system called EDS to illustrate how software components may be compromised for malicious use, causing anomalous system behavior. The problem definition and research objectives are followed in Section III. The overview of the ARMOUR framework is provided in section IV, followed by its details in Sections V and VI. Section VII reports on our evaluation results. Section IX surveys related work. The paper concludes with a discussion of the potential threats to the validity of our approach along with the future work.

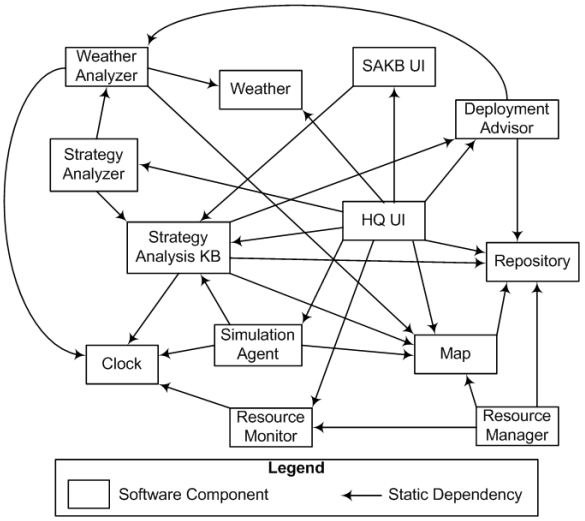


Fig. 1: Subset of EDS Software Architecture

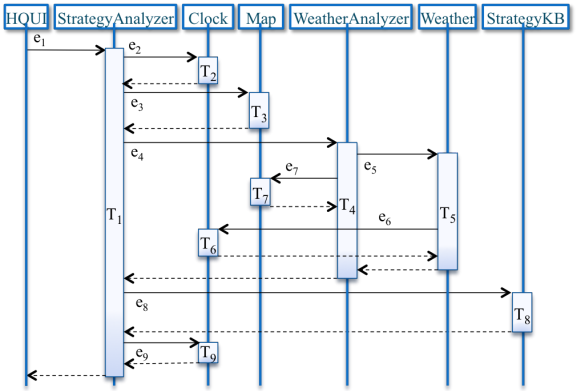
II. Motivating Example

We illustrate the concepts and evaluate the research using a real-world software system, called Emergency Deployment System (EDS), which was developed in another project in partnership with a government agency. EDS 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 support emergency responders on tasks such as tracking and distributing resources to rescue teams, analyzing different deployment strategies, etc. In the largest deployment of EDS to-date, it was deployed on 105 nodes and used by more than 100 users [5].

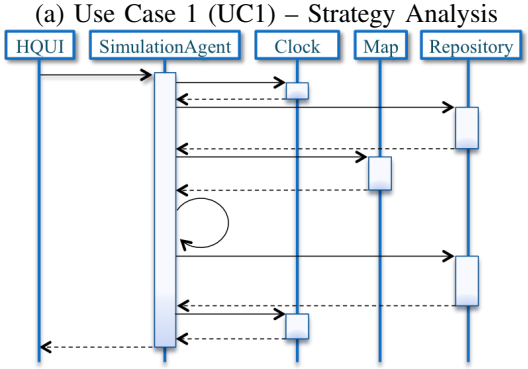
EDS is representative of a large *component-based* software system, with each component being a coarsely grained unit of software that can be independently built and deployed [6], possibly at different host nodes and locations. For EDS, in particular, each component is realized as a collection of Java objects deployed in its own JVM. Each component provides one or more Remote Method Invocation (RMI) interfaces (arrows in the diagram), discoverable from a registry and invoked over the network by any RMI client application.

Like any software system, the EDS is designed to fulfill a number of user requirements or *use cases*. The sequence diagrams for two of them are shown in Figure 2 as examples. We see that each diagram involves a sequence of interactions among different components. The components interact with one another by exchanging messages (events) that may utilize various communication mechanisms (Java RMI in this case).

In today's environment of ubiquitous connectivity, online systems such as EDS are often subject to various exploits and attacks, both external and internal. Intrusion detection sensors have been developed and perfected over the years to effectively detect network-level (such as port scanning and denial of service) and host-level (such as buffer overflow or illegal root access) attacks. Attacks at the *software application* level, however, are often more sophisticated and much harder to detect, especially when they exploit vulnerabilities in seemingly innocuous user interfaces such as a web browser or a mobile



(a) Use Case 1 (UC1) – Strategy Analysis

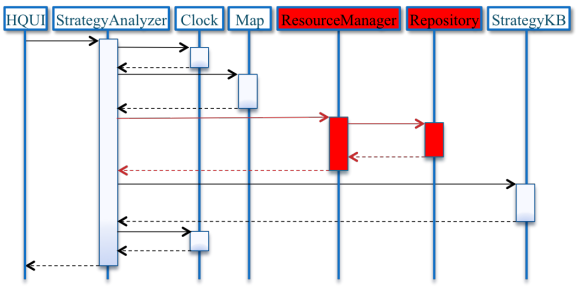


(b) Use Case 2 (UC2) – Scenario Simulation

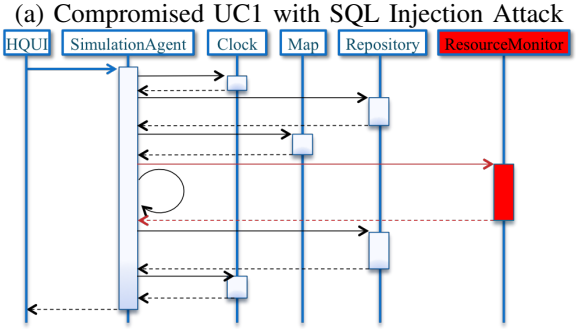
Fig. 2: Examples of EDS Component Interactions

app. Consider a scenario in which an attacker employs SQL Injection [7] through the browser-based Headquarters User Interface (HQUI) component and successfully compromises the $StrategyAnalyzer$, a Java Servlet application residing in a web server. Using a malicious database script to modify the component's configurations, the attacker is able to make $StrategyAnalyzer$ send requests to a $ResourceManager$ component to retrieve sensitive information about all deployed resources, as shown in Figure 3 (a) (Compare with Figure 2 (a)). SQL Injection is only one of many threats to web applications today (see [8]). As another example, the HQUI client may be used by a rogue employee to log into $SimulationAgent$ with administrator credentials. The employee manually rewrites the URL parameters to access $ResourceMonitor$ for real-time personnel locations (Figure 3 (b), compare with Figure 2 (b)).

Can these threats be thwarted by conventional security mechanisms? In the former scenario, a network-based IDS (e.g., Snort) may be used to examine the HTTP traffic and look for specific patterns (such as SQL quote marks). However, the security administrator must manually maintain an up-to-date database of known signatures to keep up with evolving attacks and also keep a difficult balance between the signature patterns being too generic (high false positives) and too specific (high false negatives). The limitations of IDS tools against such application-level attacks have been highlighted in recent empirical studies [9]. In the second scenario, since the user is a legitimate user with full access to the system, she will not trigger any network or host based IDS alarms. In this



(a) Compromised UC1 with SQL Injection Attack



(b) Compromised UC2 resulting from Insider Attack

Fig. 3: Examples of EDS Attack Cases

case, the conventional mechanisms are generally ineffective against insider attacks.

Given these limitations, we believe a more robust approach should (a) complement existing security approaches with additional focus on architectural-level behavior and (b) use the system's "normal" usage model as the basis for threat detection, which eliminates the need for maintaining attack specifications. The approach will have the obvious advantage of being effective against insider attacks as well as outside, and being able to detect threats both known and unknown.

Building such a model for a component-based software system, however, is not without challenges. First, the target system's behavior model is not always completely and accurately documented; nor is it always kept up to date with actual implementations. Reverse-engineering the system design from source/binary code can only help to a limited extent. Second, component interactions (such as sequences shown in Figure 2) can be user-driven, stochastic and non-deterministic. In service-oriented or peer-to-peer architectures, in particular, components may be dynamically discovered and invoked without a prescribed specification. The combinations of all possible execution sequences can be very large. Our approach to addressing these challenges involves learning a usage model of dynamic component interactions *at runtime*, without any pre-defined behavior specifications. We first start with some definitions and assumptions to frame the problem.

III. Problem Description

First, let $C = \{c_i\}$ be a set of **software components** that run independently and are capable of sending and receiving messages among one another. An **event** is defined as a message from a source component to a destination component, captured as a tuple $e =< c_{src}, c_{dst}, t_s, t_e >$, where $c_{src}, c_{dst} \in C$ and t_s, t_e are the start and end timestamps of the event's occurrence, respectively[1]. In Figure 2 (a), for example, event e_3 is a message from $StrategyAnalyzer$ to Map, also denoted as $StrategyAnalyzer \rightharpoonup Map$. Here we assume the network clock is synchronized (e.g., using the Network Time Protocol (NTP) [10]) such that timestamps recorded in all components are within a margin of error, ε_{NTP}; we account for this margin in the mining process (see Section V-A).

A **transaction** T_i is performed in a component when receiving a message e_i. Note that even though Figure 2 depicts transactions that happen to be *synchronous*, i.e., a response message goes back to the source component at the end of T_i (as in the case with many request-response protocols like HTTP or Java RMI), we do NOT rely on such an assumption; an event may also be asynchronous, without waiting for the completion of the transaction. In the latter case, the duration of the event is simply the network latency.

An event's **Perceived Execution Closure** (PEC) e^+ is an itemset that includes the event itself plus all child events that *may* have been triggered by the event. In Figure 2 (a), for example, events e_5, e_6, and e_7 are all triggered by e_4, therefore $e_4^+ = \{e_4, e_5, e_6, e_7\}$. Here the closure is considered *perceived* because we do not assume we know the true causality among events; they can only be inferred by the source and destination of components and the timestamps of events.

A **Use-case Initiation Event** (UIE) represents a "top level" event that is not part of the PEC of any other events. For EDS, a UIE naturally corresponds to one of the system's user interface sections such as simulation analysis or strategy analysis. In Figure 2 (a), for instance, e_1 is the UIE that initiates events e_2 through e_9.

These definitions reflect the following key assumptions:

- Even though the components of the system are known (as depicted in Figure 1), the *dynamic* behavior model of the target system, like what is shown in Figure 2, is *not* available or even non-deterministic due to reasons cited in the previous section.
- *Component interactions are observable at runtime*, i.e., some runtime supporting infrastructure exists to monitor and log component-level interactions. Web-based systems, for instance, typically have web server logs that can be filtered and processed for this purpose. In EDS, we developed a common logging service that stores all RMI calls in a database.
- *UIEs can be identified*. Here we simply assume that, when treating the system as a black box, its main use cases can be externally identified. An online banking system, for example, may have menu items such as "Withdrawal", "Deposit", or "Check Balance" that trigger different internal processes. The EDS system, likewise, has user-triggered events such as $HQUI \rightharpoonup StrategyAnalyzer$ and $HQUI \rightharpoonup SimulationAgent$ that are identifiable, which provide context for the ensuing interactions among software components.

[1]For convenience we assume each component has a single interface or "port". In reality different types of messages may be going through different ports, in which case we can simply add ports to the tuple

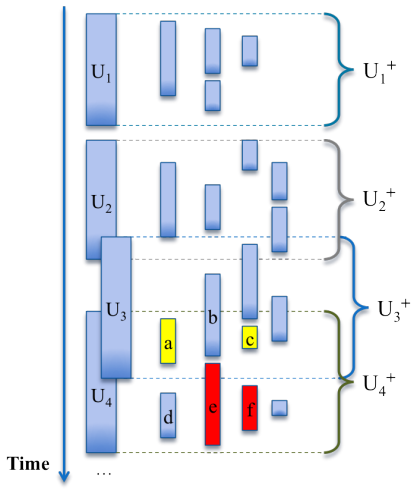


Fig. 4: Creating PECs from the Event Stream

Given the observability of the UIEs, we can use them to divide the system's execution event traces into "baskets", with each basket being a UIE U_i's perceived execution closure $U_i^+ = \{e_1, e_2, ..., e_n\}$, as illustrated in Figure 4. The figure is a visual depiction of events from the system execution history, each aligned along the time axis based on its start time and duration. Four UIEs, U_1 through U_4, divide the events into four PECs U_1^+ through U_4^+, respectively. Note that, in a multi-user system such as EDS, multiple concurrent user sessions will cause UIEs to overlap. The events a and c (highlighted in yellow), for example, are captured in both U_3^+ and U_4^+. Conceivably, the larger the number of concurrent users, the more overlap UIEs will have with one another. As will be seen later, concurrency of user activities turns out to be a major source of noise in the data mining process and thus a major challenge to be addressed.

Under our problem setting, anomalous and potentially malicious events may be present in the target system's event stream, like events e and f (highlighted in red) in Figure 4. Our threat detection problem can now be summarized as the following question: *Given an event stream* $Q = \{..., e_{j-1}, e_j\}$ *in which* e_j *is the most recent event, the observed UIEs* $\{U_i\}$ *along with their PECs* $\{U_i^+\}$*, what is the likelihood of* e_j *being an anomaly?*

IV. Approach Overview

The architecture of our mining framework, ARMOUR, is depicted in Figure 5. It consists of two major phases, *Mining the Behavior Model* and *Applying the Model*, that form a continuous MAPE-K cycle to protect the target system at runtime.

In the mining phase, system execution traces are captured by sensors in the target system, from which inter-component events are extracted and stored in the Event Log. The events are then preprocessed to produce event sequences for each UIE based on pre-identified UIE types. Afterwards, a tailored Generalized Sequential Pattern (GSP) mining algorithm is applied to the sequences. Sequential patterns mining [4], also known as frequent episode mining, is a technique for

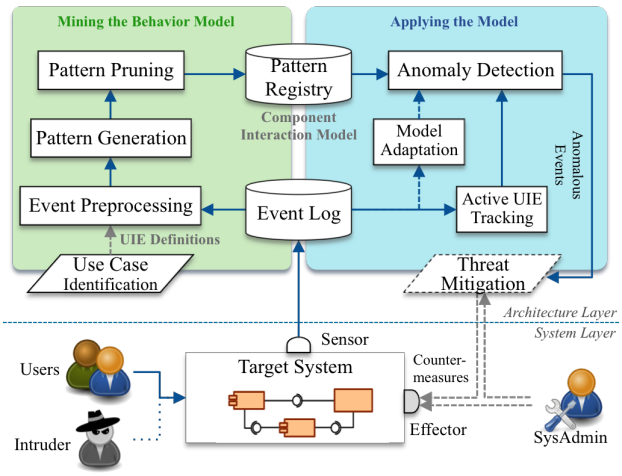


Fig. 5: ARMOUR Self-Protection Framework Overview

discovering frequently occuring patterns in data sequences, used in a wide range of application domains from retail industry to biological research. Here we employ it to build a *pattern registry* that represents a component interaction model for the normal system behavior at runtime. Our framework is based on the intuitive insight that if an event does not have *any* strong associations with past events in the model, the event is likely to be anomalous. Since regular GSP mining may generate a vast number of patterns, we customize it using the architecture knowledge in order to drastically prune the search space and make the algorithm efficient enough for runtime use, as described in detail in Section V.

The second phase is *applying the model* on current system events as they occur to detect anomalous behavior. We developed an efficient detection algorithm for this purpose, which produces a quantitative anomaly measure for each event. When the anomaly likelihood exceeds a configurable threshold, the event is marked as suspicious and its details are recorded. As mentioned earlier, the number of simultaneously running user sessions may fluctuate at runtime and adversely impact the detection accuracy. We therefore take an extra step to monitor the degree of execution concurrency and adapt the model accordingly. Section VI elaborates on the design details of this phase. Once detected, the anomalous events are flagged and sent to *Threat Mitigation*, which may result in autonomic and/or manual countermeasures being deployed to the target system (e.g., isolating a compromised component). This final step is beyond the scope of this paper, but has been demonstrated by the authors using a pattern-based self-protection approach [11].

V. Mining the Behavior Model

In the context of sequential pattern mining, a *sequence* is defined as an ordered list of itemsets, where each itemset is a set of literals called items. We denote a sequence by $\prec s_1, s_2, ..., s_n \succ$, where s_i is an (unordered) itemset, denoted by $(x_1\ x_2\ ...\ x_k)$. Using a retail example, a customer's purchase history at a store may be viewed as a data sequence consisting of a list of transactions ordered by transaction time, with each transaction s_i containing one or more purchased items.

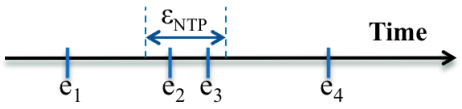


Fig. 6: Event Sequence Example

A sequence $\prec a_1, a_2, ..., a_n \succ$ is a *subsequence* of another sequence $\prec b_1, b_2, ..., b_m \succ$ if there exists integers $i_1 < i_2 < ... < i_n$ such that $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, ..., a_n \subseteq b_{i_n}$. For example, $\prec c, (d\ e) \succ$ is a subsequence of $\prec g, (c\ h), k, (d\ e\ f) \succ$, but $\prec d, e \succ$ is not a subsequence $\prec (d\ e) \succ$ (and vice versa). Given a set of data sequences, the objective of sequencial pattern mining is therefore to find all subsequences that meet a user-defined minimum frequency (called *support*). Continuing with the retail example, mining customer transactions may reveal a pattern like "15% customers bought Star Wars movies, followed by Lego Star Wars toys in a later transaction". Now we provide details of applying this technique to mine component interactions in the EDS system.

A. Event Preprocessing

The first step of our approach is pre-processing the event log into data sequences as input to the mining algorithm. Under our problem setting, the list of events supporting a use case naturally constitutes a "customer" sequence, with each sequence being initiated by a UIE, bounded by its PEC, and ordered by their start time t_s. For the EDS system, because all user actions are initiated at the user interface component $HQUI$, each UIE is of the form $HQUI \rightharpoonup c \in C$, such as $HQUI \rightharpoonup SimulationAgent$, $HQUI \rightharpoonup StrategyAnalyzer$, etc. At runtime, we can easily keep track of active user sessions by monitoring web server logs, and allocate each incoming event into one or more UIE clousures according to their start and end timestamps. When a UIE is completed, a data sequence is produced and stored.

As mentioned in Section III, concurrent user activities may produce component interaction events that occur at the same time. Further, due to the margin of error in network time keeping, it is not possible to determine the exact order of events occurring with close proximity to one another. Therefore, in the preprocessing step we use a sliding ε_{NTP} window: consecutive events whose start timestamps fall within the window are treated as co-occurring and added to the sequence as a single itemset. For instance, the events shown in Figure 6 will be sequenced into $\prec e_1, (e_2\ e_3), e_4 \succ$.

B. Customized GSP Mining

A variety of sequential pattern mining algorithms exist; we chose to use the well-known GSP algorithm [4], due in part to its available open-source implementation [12]. We use GSP to discover ***Event Association Patterns*** (EAP) of the form:

$$P = \prec s_1, s_2, ..., s_n \succ :\ supp \tag{1}$$

where each element s_i is an itemset of co-occurring events and support $supp$ is the count of all sequences to which P is a subsequence, divided by the total number of data sequences: $supp = \sigma(P)/N$. Naturally $supp \in [0, 1]$, and the more frequently P occurs, the higher the $supp$. As a concrete example, here is an EAP generated from GSP test runs:

$$\prec HQUI \rightharpoonup StrategyAnalyzer,\ (StrategyAnalyzer \rightharpoonup Clock\ StrategyAnalyzer \rightharpoonup StrategyKB) \succ :\ 0.45$$

The original GSP algorithm follows an iterative process that generates candidate sequences with $(k+1)$ elements based on existing k-sequences, then prunes the candidates that do not meet the minimum support level.

We need to tailor the GSP algorithm in two unique ways. First, given the fact that anomalies, esp. malicious attacks seeking to covertly exploit the target system, are rare events occurring with very low frequency, the vast majority of the EAPs represent normal system use. Therefore, we need to set the minimum support level $minsupp$ to very low (e.g., 0.1) in order to comprehensively capture the system behavior model. This is very different from the typical use of GSP, which is to discover only the highly frequent patterns. A direct consequence of this is that an exponentially large number of pattern sequences will be produced for any slightly more complex system, with unacceptable time and space requirements. Fortunately, our architecture knowledge of the system comes to assistance in this regard. In particular, we notice that component interactions have causal relationships: an event is triggered by a user action (UIE), and the event's destination component in turn triggers events to other components. Therefore we inserted the following heuristic to the candidate generation phase of GSP: keep a candidate sequence $\prec s_1, s_2, ..., s_n \succ$ *iff* each non-UIE event in the sequence has a preceding triggering event. Formally, for any element s_i,

$$\forall e_v \in s_i\ (\ \exists e_u \in s_{j,1 \le j \le i-1}\ (\ e_u.c_{dst} = e_v.c_{src}\ ))\tag{2}$$

With this heuristic, the number of added candidates in each GSP iteration becomes linear to the number of possible event types (i.e., source to target component combinations), or $O(|C|^2)$, effectively reducing the growth of candidate sequences from permutational to quadratic in relation to the number of system components.

Second, we recognize that the prior probabilities of use cases that drive system behavior may vary greatly. Some UIEs, such as viewing map and weather data for situation awareness, occur quite frequently, whereas others such as performing system maintenance occur far less often. Furthermore, the system behavior shifts over time. Deploying disaster response resources, for instance, tends to occur at times of emergency. As a result, it is likely that some component interaction events, especially those associated with infrequent user actions, may not meet the minimum support level. This led us to tailor the GSP logic to mine the component interactions *under a specific usage context*. We therefore revise the definition of an EAP:

$$P = \prec s_1, s_2, ..., s_n \succ :\ supp_i \mid UIE_i \tag{3}$$

where $supp_i$ is the *conditional* support with respect to a specific use case. The implementation of this tailored logic is relatively simple: the training sequences from the event log are put into different "bins" by their UIE types; the GSP algorithm is applied to each bin to generate the conditional EAPs for the

specific UIE. The pattern registry can thus be viewed as having multiple partitions, one for each UIE type.

It is not difficult to envision that, the set of EAPs produced from GSP mining can serve as a simple form of a system behavior model at runtime. The EAPs collectively provide insight into how the components interact with one another at the architecture level.

VI. Detecting Anomalous Behavior

Since the generated EAPs collectively represent the system's *normal* behavior, we have reasons to suspect any component interaction pattern that falls outside of this model to be an anomaly. In security nomenclature, our method takes an *anomaly-based* approach for threat detection as opposed to a *signature-based* approach: rather than attempting to formulate and recognize all possible attack signatures, we consider any interaction pattern that falls outside of the normal system usage a potential threat. As will be discussed later, the former approach has a significant benefit of being effective against future unknown threats.

Algorithm 1 *measureAnomaly*: Compute Anomaly Likelihood

```
   Input: P = {p_i}                          ▷ Pattern registry, use case-partitioned, indexed
   Input: minsupp                            ▷ Minimum support level used by GSP
1: procedure measureAnomaly(e)
2:     L_anomaly(e) ← 1
3:     for U_i ∈ findEnclosingUIEs(e) do
4:         {PTS_j} ← findPTSs(U_i, e)        ▷ Find all PTS sequences for e
5:         supp_i ← Max_j(patternLookup(PTS_j, U_i))   ▷ Find highest support
6:         L_anomaly(e) ← L_anomaly(e) × (1 − supp_i)
7:     end for
8:     L_anomaly(e) ← L_anomaly(e)^{1/m}
9:     return L_anomaly(e)
10: end procedure
11: procedure patternLookup(PTS, UIE)
12:     if ∃(p = PTS : supp_p|UIE) ∈ P then
13:         return supp_p                     ▷ Look up pattern registry, return support value
14:     else
15:         return minsupp/2                  ▷ PTS not found in registry, return estimate
16:     end if
17: end procedure
```

We have developed an effective method for quantitatively determining the anomaly likelihood of an event. The skeleton of the method, *measureAnomaly*, is shown in Algorithm 1. The idea behind the algorithm is as follows. Any new event e captured in the system event log may fall under multiple UIEs due to system concurrency (recall Figure 4), and it is possible to observe the UIE closures of which e is a perceived member, that is, $e \in U_i^+,\ i = 1, ..., m$. Method $findEnclosingUIEs(e)$ (line 3) performs a time stamp-based search in the event log to find all enclosing UIEs for event e. Within each U_i^+, it is also possible to use event source and destination information to discover zero or more *Perceived Triggering Sequences* (PTS) for e, in the form of $\{PTS_e =\prec U_i, e_1, ..., e_k, e \succ\}$, such that $U_i.c_{dst} = e_1.c_{src}, e_1.c_{dst} = e_2.c_{src},\ ...\ , e_k.c_{dst} = e.c_{src}$. Method $findPTSs(U_i, e)$ on line 4 achieves this by performing a recursive, depth-first search within U_i^+.

Again, the word "perceived" signifies that the sequence is purely by observation, not by any knowledge of the events' true causality. For example, in Figure 4, potential PTS's for event c include $\prec U_3, a, c \succ$, $\prec U_3, b, c \succ$, $\prec U_3, c \succ$, $\prec U_4, a, c \succ$, and $\prec U_4, c \succ$. Intuitively, if we can find at least one PTS_e matching an EAP in the pattern registry, it means the presence of event e is "explained" by UIE U_i. Conversely, if none of the U_i's can explain e, we have reason to believe it is an anomaly. In other words, the likelihood of e being an anomaly is the conjunctive probability of e ***not*** explained by U_i for all $i = 1, ..., m$. Assuming mutual independence of system use case occurrences, the anomaly likelihood is $\prod_{i=1}^{m}(1 - supp_i)$, where $supp_i$ is the highest support we can find in the pattern registry for e's triggering sequences within U_i^+ (see line 5). When no matching EAP is found, it is necessary that the support of a PTS falls between 0 and $minsupp$. In this case *patternLookup*() returns the mean, $(0 + minsupp)/2$, as an estimate for $supp_i$ (line 15).

Algorithm 2 *mainArmour*: Main Routine

```
   Input: P = {p_i}                          ▷ Pattern registry, use case-partitioned, indexed
   Input: Q = {..., e_i, ...}                ▷ Event log / queue
   Input: L_threshold                        ▷ Detection confidence threshold
1: procedure mainArmour(Q)
2:     e ← dequeue(Q)                        ▷ Retrieve event from queue
3:     while e ≠ null do
4:         if refreshPeriodReached() then    ▷ Refresh pattern registry
5:             minsupp ← 1.0 − L_threshold
6:             P_new ← runCustomGSP(minsupp)   ▷ Run in new thread
7:             P ← P_new
8:         end if
9:         L_anomaly(e) ← measureAnomaly(e)
10:        if L_anomaly(e) ≥ L_threshold then
11:            flagEvent(e)                  ▷ Flag event for subsequent mitigation
12:        end if
13:    end while
14: end procedure
```

This likelihood value, however, is not yet a useful detection metric due to the compounding effect introduced by system concurrency. For example, suppose we let $minsupp = 0.10$ and use a detection threshold of 0.9. In a single user scenario, if an anomalous event is injected into the system, its perceived triggering sequences will not match any EAPs in the registry, therefore procedure *patternLookup*() will return the default minimum confidence $minsupp/2 = 0.05$ (line 15 of Algorithm 1). The resulting likelihood of anomaly will then be $(1 - 0.05) = 0.95 > 0.9$, correctly marking the event as an anomaly. Now let's imagine the malicious event falls into 3 concurrent UIE closures. Here the likelihood will be $\prod_{i=1}^{3}(1 - conf_i) = (1 - 0.05)^3 = 0.86 < 0.9$, falling below the threshold and thus rendering this event as a false negative. To address this challenge, we further normalize the likelihood by taking its geometric mean:

$$L_{anomaly}(e) = [\prod_{i=1}^{m}(1 - supp_i)]^{\frac{1}{m}} \tag{4}$$

where m is the number of enclosing UIEs for e (line 8 of Algorithm 1). Obviously $L_{anomaly}(e) \in [0, 1]$, and the higher its value, the higher likelihood that e is an anomaly.

As mentioned in Section IV, ARMOUR runs in a continuous loop in parallel to the target system. Events are fed into the detection algorithm as they are captured in the event log. At the same time, the pattern registry is regenerated periodically based on most recent event history, in a separate thread that doesn't block the detection process. Putting

everything together, Algorithm 2 lists the pseudo code for the main routine of the ARMOUR framework, omitting details of self-explanatory subroutines. Please note the complementary relationship between the detection threshold and the $minsupp$ parameter (line 5): the higher confidence we would like on the detections, the lower support level we need for the GSP mining algorithm in order to have a more thorough capture of the system behavior model.

VII. Evaluation

Our experimentation environment involves an instantiation of the original EDS system that uses Java RMI for sending and receiving event messages among components. As mentioned earlier, logging functionality ensures all component-level events are captured in a MySQL database. A multi-threaded RMI client is written to play back a configurable number of concurrent user sessions. All ARMOUR framework components are developed in Java, including the customized GSP implementation adapted from Weka [12]. Both test runs and data analysis are run on quad-core Mac OS X machines.

As mentioned in Section V-A, we take into account the impact of network timing errors as we pre-process the event log into data sequences. Even though we conducted all test runs within a LAN, we set the network timing error margin ε_{NTP} to be 10ms, a value safe for the public Internet [10] to mimic the real-world EDS runtime environment.

In order to evaluate the performance of the framework under different concurrency settings, we set up different test cases with a varying number of concurrent users, from 10 to 100. In reality, however, users may have different proficiency levels and browsing habits, therefore the number of concurrent users is not a consistent measure of system concurrency. The number of enclosing UIEs for an event, described in Section VI, on the other hand, is a more objective measure. Here we define a concurrency measure γ at the time of event e_k as the moving average of the number of UIE closures to which e_k belongs, computed for the most recent N events:

$$\gamma(e_k) = \frac{1}{N}\Big(\sum_{j=k-N+1}^{N} |findEnclosingUIEs(e_j)|\Big) \quad (5)$$

The correlation between γ with the number of simulated users in our experiments is shown in the header rows of Table I and other tables. Note that the users in our simulations are active, always-busy users intended to generate a heavy load on the system, and therefore may represent a much larger number of human users in a real-world setting.

A. Determining Training Window

How much training data from the system execution history should we use to generate (and re-generate) the pattern registry? In our evaluation we observed that as the size of the training set increases, the size of the pattern registry $|P|$ grows asymptotically towards a stable limit, indicating nearly all component interactions have been captured. Accordingly, we run GSP repeatedly over an increasing training window size, until the growth in $|P|$ over the previous iteration falls below a threshold (say, 5%). Figure 7 shows the pattern registry

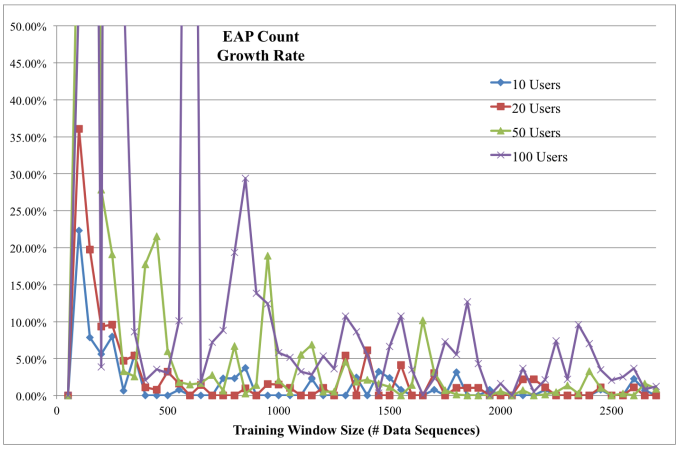


Fig. 7: Pattern Registry Growth Over Training Window Size

size (i.e., EAP count) growth rate under different concurrency settings. It shows that the higher the concurrency level, the more noise is added in system behavior (recall Section III), therefore more training data is needed for the pattern registry to converge (with a training window of 2,500 for 100 users).

B. Threat Detection Accuracy

Our evaluation uses two threat scenarios described in Figure 3. Based on the assumption that anomalies are outlier events, each of the attack cases was inserted with a probability of 0.27% (i.e., following the three-sigma rule of thumb [13]). We will revisit this assumption later in this section. We used a training window of 2,500 sequences (that is, UIEs) from the event log for GSP mining, while testing was done on 1,000 sequences. $minsupp$ for the mining algorithm is set to 0.1 in order to capture as many normal component interactions as possible. Accordingly we set the confidence threshold to $(1 - minsupp)$ (recall line 5 of Algorithm 2), or 0.9.

The test results, listed in Table I, demonstrate our framework is quite effective in detecting both threats hidden among a large number of normal system events, especially considering these attacks may have been missed by IDS sensors. Precision is close or equal to 100% for most tests and recall is consistently over 70%, without any degradation over the 10x increase in system concurrency, an indicator of not only the effectiveness of the concurrency-normalized anomaly measure defined in Section VI, but also the practical potential of using the ARMOUR framework with large-scale, multi-user systems. The table also includes metrics for ARMOUR's computational efficiency such as mining time and detection time, which will be analyzed later in Section VII-F.

C. Training Environment

Under ideal circumstances, the pattern registry would need to be mined using clean, attack-free training sets before use at runtime on real data. In fact many prior data mining techniques for security require such supervised learning (more details in Section IX). The nature of the pattern mining process, however, hinted that we can eliminate the need for a clean training environment altogether – due to the outlier nature of anomalies, they do not occur frequently enough to make their

TABLE I: Detection Results for 2 Threat Scenarios

#active users	10	20	50	100
Concurrency Measure(γ)	5	11	28	56
Attack A (Figure 3 (a))				
TP Count (#Events)	14	22	16	26
FP Count	2	0	0	0
FN Count	5	9	6	9
TN Count	14,375	13,606	13,377	13,954
TP Rate (TPR)	0.737	0.710	0.727	0.743
FP Rate (FPR)	1.39E-4	0.0	0.0	0.0
Precision	0.875	1.0	1.0	1.0
Recall	0.737	0.710	0.727	0.743
F-Measure	0.800	0.830	0.842	0.852
EAP Count	125	147	147	152
Mining Time (ms)	315	347	356	436
Detection Time / Event (ms)	31	29	35	41
Attack B (Figure 3 (b))				
TP Count	17	19	14	15
FP Count	1	0	0	0
FN Count	5	8	4	4
TN Count	13,475	14,005	13,697	14,810
TP Rate (TPR)	0.773	0.704	0.778	0.789
FP Rate (FPR)	7.42E-5	0.0	0.0	0.0
Precision	0.944	1.0	1.0	1.0
Recall	0.773	0.704	0.778	0.789
F-Measure	0.850	0.826	0.875	0.882
EAP Count	127	143	147	147
Mining Time (ms)	359	322	344	451
Detection Time / Event (ms)	35	32	34	61
$TPR = TP/(TP+FN)$; $FPR = FP/(FP+TN)$; $Precision = TP/(TP+FP)$; $Recall = TPR$; $F\text{-}Measure = 2TP/(2TP+FP+FN)$				

way into the EAPs, as long as the $minsupp$ level of the GSP algorithm is set well-above the anticipated anomaly rate.

It is worth noting that the detection results shown in Table I were indeed produced while running ARMOUR in parallel to the EDS system, with the model trained over actual, tainted event logs. This confirms that ARMOUR can run *unsupervised* and does not need a clean training environment. This gives ARMOUR a big practical advantage as clean training data for a real-world system is usually costly to obtain.

D. Detecting Unknown Threats

As alluded to in Section VI, because an anomaly-based detection model focuses on capturing normal system behavior rather than capturing all possible attack signatures, it has the natural advantage of being able to detect new threats that are not previously known. To validate that the ARMOUR framework has this capability, we set up a cross-validation procedure that performs GSP pattern mining in the presence of Attack A and then runs threat detection against an event log that includes injected Attack B, and vice versa.

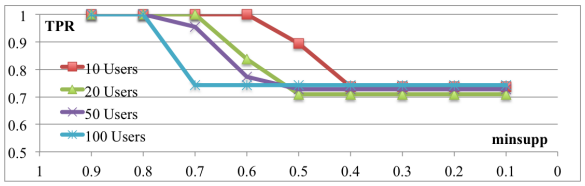
Table II shows the test results at different concurrency levels. We can see the algorithm is equally effective for detecting threats that were not present during the mining phase! Given the ever-changing attack vectors and so-called "zero-day" threats for applications today, ARMOUR can be an extra layer of protection against future exploits involving undiscovered software vulnerabilities.

E. Sensitivity Analysis

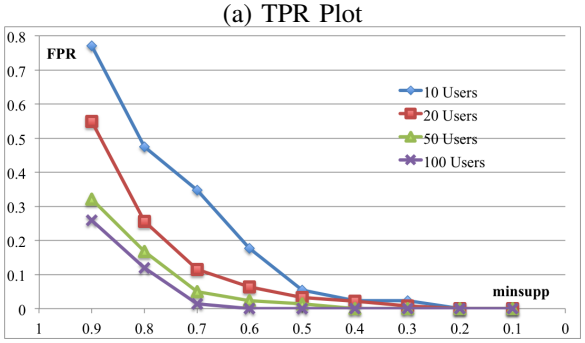
ARMOUR makes very few assumptions on the target system's runtime behavior and its runtime environment. In fact, the only two key parameters our evaluations depend on are the $minsupp$ level of the mining algorithm and the prior probability of anomalies. We would like to examine how ARMOUR's detection performance is impacted by these parameters in order to discover any limitations of our approach.

TABLE II: Unknown Threat Detection Results

#active users	10	20	50	100
Concurrency(γ)	6	11	28	56
Mining with Attack A, detection against Attack B				
TP Count (#Events)	13	15	19	10
FP Count	1	0	0	0
FN Count	5	5	8	2
TN Count	14,444	14,239	14,090	14,103
Precision	0.929	1.0	1.0	1.0
Recall	0.722	0.750	0.704	0.833
Mining with Attack B, detection against Attack A				
TP Count	16	24	14	16
FP Count	1	0	0	0
FN Count	8	12	7	6
TN Count	35,086	34,539	34,430	34,339
Precision	0.941	1.0	1.0	1.0
Recall	0.667	0.667	0.667	0.727



(a) TPR Plot



(b) FPR Plot

Fig. 8: TPR and FPR Sensitivity to $minsupp$

First, we repeated the test runs for Attack A under different $minsupp$ levels, from high to low, in order to see the impact on the True Positive Rate (TPR) and False Positive Rate (FPR) results, which are shown in Figure 8. We see that as $minsupp$ is lowered, TPR remains at a high level and is not very sensitive to minsupp changes, while FPR rapidly decreases towards 0. These plots confirm our earlier understanding that setting support at low levels can effectively capture the component interaction model. The "hockey stick" shape of the FPR plot indicates our framework is far less sensitive to $minsupp$ at lower levels, implying that there is no need to search for an optimal $minsupp$; ARMOUR can operate effectively over a range of $minsupp$ towards the lower end.

Also of interest to our work is the prior probabilities of anomalous events, which we assumed is very low compared with normal system use. Our approach, when run in the unsupervised mode with tainted data, is based on the premise that rare events do not occur frequent enough to interfere with building the normal system usage model. Using the Attack A scenario, we performed test runs that gradually increased the anomaly rate above the default 3-sigma level, with other parameters kept the same. The results for 20 concurrent users are listed in Table III. The results of default anomaly rate (0.27%) are repeated from Table I for comparison. We can see that, while precision remains high, recall deteriorates quickly

TABLE III: Detection Results Under Increased Anomaly Rates

Anomaly Rate	0.27%	1.0%	2.0%	3.5%	5.0%
TP Count	22	5	11	11	6
FP Count	0	2	0	1	1
FN Count	**9**	**28**	**59**	**117**	**169**
TN Count	13,606	13,892	15,066	14,712	14,284
Precision	1.0	0.714	1.0	0.917	0.857
Recall	0.710	0.152	0.157	0.086	0.034

when the anomaly rate gets higher, driven by more false negatives, indicating more and more anomalies start to be considered normal events.

The results are hardly surprising: when the prior probability of anomalies approaches $minsupp$, anomalous patterns start to be captured by the mining algorithm as EAPs and thus become mixed with the normal system behavior. At this point the model starts to lose its ability to distinguish anomalous events from legitimate ones[2]. Fortunately, higher-frequency attacks that seek to do immediate harm to the system will likely be recognized as Denial of Service (DoS) attacks and dealt with accordingly. This serves as a good reminder that ARMOUR should be used in conjunction with existing intrusion detection mechanisms rather than replacing them.

F. Computational Efficiency

1) Mining Performance: Since our customized GSP algorithm complexity is closely tied to the number of valid candidates it generates, we expected to see a strong correlation between the EAP Count and the mining time, as confirmed in Table I. The table shows that at the highest concurrency level for attack scenario A, it takes about 0.4 second to complete a mining run. Considering the mining process runs in a separate thread and the pattern registry only needs to be refreshed periodically, it is quite practical to apply this algorithm to support on-the-fly anomaly detection at runtime. To put this in perspective, we were not even able to complete a mining run of the original, unaltered GSP algorithm for just 10 concurrent users once $minsupp$ drops below 0.4 due to either time or memory limitations; the number of candidates it generated was simply too large. By comparison, our architecture-based heuristics prove to be highly effective in reducing the candidate search space even at low support levels.

2) Anomaly Detection Performance: To be used at runtime, the detection portion of ARMOUR must be efficient and fast, in order to keep up with real-time system execution. A quick complexity analysis of Algorithm 1 shows that:

- Running time of $findEnclosingUIEs(e)$ on line 3 depends on concurrency measure γ, i.e., in $O(\gamma)$ time;
- Similarly, the for loop on lines 3-7 is repeated γ times;
- $findPTSs()$ on line 4 carries out a DFS search in U_i^+. Its runtime depends on the size of U_i^+ which is UIE specific, but can be amortized to $O(\gamma)$ time;
- *patternLookup*() is hash table-based and runs in $O(1)$.

In theory, we can therefore conclude that Algorithm 1 runs in $O(\gamma)$ time, that is, proportional to the concurrency measure. After realizing that once a UIE and its associated data structures (such as DAGs for the PTSs) are constructed in memory, they can be cached and reused, we further optimized Algorithm 1 so that the amortized running time per event is nearly $O(1)$, regardless of the system concurrency level. The average detection time per event, as shown in Table I, is about 30ms, indicating the detection algorithm is highly capable of keeping up with high-volume user activities.

[2]Clearly, if clean training data is available from a controlled environment, this will not be an issue

VIII. Threats to Validity

Two possible threats to the validity of our approach deserve additional discussion. First, the ability to identify external interfaces or UIEs might be a challenge for certain types of systems. An ad-hoc system with free-form interactions across components (such as a wireless network of sensors), for example, may not be a good candidate for our approach. The ideal candidates for applying the ARMOUR framework would be online enterprise systems that have clear business use cases and process patterns, regardless of application domain.

Second, all self-protection frameworks, including ARMOUR, are subject to exploitations once their approaches are known to attackers. In ARMOUR's case, for instance, an attacker could slowly increase the frequency of the anomaly events, to a point that they start to "blend in" with normal usage, as mentioned in earlier sensitivity analysis. To mitigate this threat in practice, the ARMOUR framework should be used in conjunction with conventional security mechanisms, such as DoS prevention, application firewalls, and IDS, that are quite effective at detecting frequently seen attacks at the network or host level. ARMOUR complements these approaches by detecting covert attacks at the application level.

IX. Related Work

Software engineering research has actively focused on mining behavioral models automatically from system execution traces ([14], [15], and our prior work [16], [17], to name a few). Recent efforts have also expanded to modeling user behavior [18] and concurrent systems [19]. However very few of these efforts focused on security. Only recently did we start to see adaptive security approaches tackling application-level attacks including insider threats, as seen in [20] for instance, which employs dynamically generated access control models.

The idea of detecting security anomalies based on a model of the system's "normal" behavior is by no means a new one. Early research in the nineties ([21] and many others) exploited the correlation of short sequences of system calls; a sequence that falls outside of normal call patterns may indicate abnormal behavior. Later approaches used other constructs such as method call profiles based on dynamic sandboxing [22], finite state automata based on static program analysis [23], program execution paths extracted from call stack information during normal program runs [24], execution graphs [25], process creation trees [26], or call graphs and calling context trees of cloud-based applications [27]. ARMOUR differs from these approaches in that: (a) these approaches are host-based, i.e. mining metadata of individual processes or programs, while ARMOUR looks at abstract software component interactions that may span across multiple process spaces or even multiple hosts; (b) most require supervised training whereas ARMOUR

can run unsupervised; (c) most assume normal program behavior is deterministic and stable, where ARMOUR assumes the behavior for an interactive system is inherently fluid and user-driven, and hence continually updates the model based on recent system execution traces.

From the techniques perspective, ARMOUR joins a large body of research in applying data mining methods to the security domain, especially those for anomaly detection [28]. Much of existing research, however, centered around intrusion detection, especially at network and host levels (e.g., [29]), and malware/virus detection for source code and executables (e.g., [30]). Among those, several efforts share our approach of unsupervised learning, i.e., using unlabeled or "noisy" training data [21], [31]. Still others used mining algorithms such as Support Vector Machines (SVM) [32], Hidden Markov Models (HMM) [33], ensemble based learning [34], etc. Few of these efforts took advantage of sequential pattern mining as does ARMOUR. Moreover, many previous efforts rely on domain-specific features (e.g., TCP/IP protocol attributes, UNIX shell commands), whereas ARMOUR is domain-independent.

X. Conclusions and Future Work

In this paper, we have argued for the importance of monitoring and assessing the overall security posture of a software system at the *architectural* level in order to detect more sophisticated threats that may otherwise go unnoticed using traditional network or host level intrusion detection techniques. Towards this objective, we have proposed a use case-driven mining framework with an adaptive detection algorithm to efficiently identify potential malicious events. Our evaluation of the approach has demonstrated very promising results with high detection accuracy, regardless of system concurrency levels. ARMOUR complements existing perimeter-based security mechanisms, provides an approach to achieve "defense in depth" for large-scale software systems with many practical advantages, including unsupervised learning with no need for clean training data, potential to detect unknown threats, and effective detection of application-level insider attacks.

Our future work will attempt to make the framework more robust, such as improving computational efficiency using cloud computing techniques and integrating ARMOUR with system self-protection methods (such as those proposed in [11]) to auto-respond to threats at runtime.

Acknowledgment

This work was supported in part by awards CCF-1252644 from the National Science Foundation, D11AP00282 from the Defense Advanced Research Projects Agency, W911NF-09-1-0273 from the Army Research Office, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

References

[1] D. M. Chess *et al.*, "Security in an autonomic computing environment," *IBM Systems Journal*, vol. 42, no. 1, pp. 107–118, 2003.

[2] E. Yuan, N. Esfahani, and S. Malek, "A systematic survey of self-protecting software systems," *ACM TAAS*, vol. 8, no. 4, Jan. 2014.

[3] M. B. Salem *et al.*, "A survey of insider attack detection research," in *Insider Attack and Cyber Security*. Springer, 2008, pp. 69–90.

[4] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Advances in Database Technology ł EDBT '96*. Springer Berlin Heidelberg, Mar. 1996, no. 1057, pp. 1–17.

[5] S. Malek *et al.*, "A style-aware architectural middleware for resource-constrained, distributed systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 256–272, Mar. 2005.

[6] B. Councill and G. T. Heineman, "Definition of a software component and its elements," in *Component-based Software Engineering*, 2001.

[7] The MITRE Corporation, "CWE-89: 'SQL injection'," http://cwe.mitre.org/data/definitions/89.html.

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

[9] I. Elia *et al.*, "Comparing SQL injection detection tools using attack injection: An experimental study," in *ISSRE 2010*, pp. 289–298.

[10] The NTP Public Services Project, "The NTP FAQ," http://www.ntp.org/ntpfaq/NTP-s-algo.htm.

[11] E. Yuan, S. Malek *et al.*, "Architecture-based self-protecting software systems," in *QoSA '13*. New York, NY, USA: ACM, 2013, pp. 33–42.

[12] M. Hall *et al.*, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[13] Wikipedia.org, "68-95-99.7 rule," https://en.wikipedia.org/wiki/68-95-99.7 rule.

[14] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *FSE*. ACM, 2009, pp. 345–354.

[15] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *FSE*, 2014.

[16] E. Yuan, N. Esfahani, and S. Malek, "Automated mining of software component interactions for self-adaptation," in *SEAMS 2014*. New York, NY, USA: ACM, 2014, pp. 27–36.

[17] N. Esfahani, E. Yuan, K. R. Canavera, and S. Malek, "Inferring software component interaction dependencies for adaptation support." ACM Trans. on Autonomous and Adaptive Systems, 2016, vol. 10, no. 4.

[18] C. Ghezzi *et al.*, "Mining behavior models from user-intensive web applications." in *ICSE*, 2014, pp. 277–287.

[19] I. Beschastnikh, Y. Brun *et al.*, "Inferring models of concurrent systems from logs of their behavior with csight," in *ICSE*, 2014, pp. 468–479.

[20] C. Bailey, L. Montrieux, R. de Lemos *et al.*, "Run-time Generation, Transformation, and Verification of Access Control Models for Self-protection," in *SEAMS 2014*. ACM, pp. 135–144.

[21] T. Lane and C. E. Brodley, "An application of machine learning to anomaly detection," in *Proc. of the 20th National Information Systems Security Conference*, vol. 377. Baltimore, USA, 1997, pp. 366–380.

[22] H. Inoue and S. Forrest, "Anomaly intrusion detection in dynamic execution environments," in *Proceedings of the 2002 Workshop on New Security Paradigms*. New York, NY, USA: ACM, 2002, pp. 52–60.

[23] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *2001 IEEE Symposium on Security and Privacy*, 2001, pp. 156–168.

[24] H. Feng *et al.*, "Anomaly detection using call stack information," in *2003 Symposium on Security and Privacy*, May 2003, pp. 62–75.

[25] D. Gao *et al.*, "Gray-box extraction of execution graphs for anomaly detection," in *CCS '04*. ACM, 2004, pp. 318–329.

[26] M. Kwon *et al.*, "PROBE: a process behavior-based host intrusion prevention system," in *Information Security Practice and Experience*. Springer Berlin Heidelberg, Jan. 2008, pp. 203–217.

[27] S. Alsouri *et al.*, "Dynamic anomaly detection for more trustworthy outsourced computation," in *Information Security*. Springer Berlin Heidelberg, 2012, vol. 7483, pp. 168–187.

[28] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–58, Jul. 2009.

[29] W. Lee *et al.*, "A data mining framework for building intrusion detection models," in *IEEE Security and Privacy*, 1999, pp. 120–132.

[30] M. Schultz *et al.*, "Data mining methods for detection of new malicious executables," in *IEEE Security and Privacy*, 2001, pp. 38–49.

[31] E. Eskin *et al.*, "A geometric framework for unsupervised anomaly detection," in *App. of Data Mining in Comp. Sec.*, 2002, pp. 77–101.

[32] L. Khan, M. Awad, and B. Thuraisingham, "A new intrusion detection system using support vector machines and hierarchical clustering," *The VLDB Journal*, vol. 16, no. 4, pp. 507–521, 2007.

[33] C. Warrender *et al.*, "Detecting intrusions using system calls: Alternative data models," in *IEEE Security and Privacy*. IEEE, 1999, pp. 133–145.

[34] P. Parveen *et al.*, "Supervised learning for insider threat detection using stream mining," in *23rd IEEE ICTAI, 2011*, pp. 1032–1039.