AN EXPERIMENT USING FACTOR GRAPH FOR EARLY ATTACK DETECTION

BY

PHUONG MINH CAO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisers:

Professor Ravishankar K. Iyer
Professor Zbigniew T. Kalbarczyk

ABSTRACT

This paper presents a factor graph based framework (namely AttackTagger) for high accuracy and preemptive detection of attacks. We use security logs on real-incidents that occurred over a six-year period at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign to evaluate AttackTagger. Our data consist of attacks that led directly to the target system being compromised, i.e., not detected in advance, either by the security analysts or by intrusion detection systems. AttackTagger detected 74 percent of attacks, a vast majority of them were detected before the system misuse. AttackTagger uncovered six hidden attacks that were not detected by security analysts.

To my parents, friends, and colleagues for their love and support.

ACKNOWLEDGMENTS

I am grateful to be advised by Prof. Ravishankar K. Iyer and Prof. Zbigniew T. Kalbarczyk. Their insights, encouragements, and supports are indispensable to completion of this thesis.

This work was supported in part by the National Science Foundation under Grant No. CNS 10-185303 CISE, by the Army Research Office under Award No. W911NF-12-1-0086, by the National Security Agency under Award No. H98230-14-C-0141, by the Air Force Research Laboratory, and by the Air Force Office of Scientific Research under agreement No. FA8750-11-20084. The opinions, findings, and conclusions stated herein are those of the authors and do not necessarily reflect those of the sponsors.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Cyber-systems are enticing attack targets since they host mission-critical services and valuable data. Cyber-attacks are often tied to leaked credentials (millions of the credentials can be bought on the black markets at a low cost [1]). Using stolen credentials, attackers impersonate as legitimate users, effectively bypass traditional defenses (e.g., network firewalls). Such attacks are often discovered in their final stages of delivering the attack payloads, e.g., replacing authentication services (OpenSSH) to harvest more credentials or utilizing computing nodes to build spamnet/botnet [2].

Detecting cyber-attacks in their early stages presents several challenges. Attackers leave no discernible trace, as they infiltrate a target system as legitimate users using stolen credentials. Only a partial knowledge of the attacks are available at the early stages. As a user has just logged in at the beginning of a user session, only a few attributes of the user profile are available for examination, e.g., user role or user physical location of the login. The user activities remain to be seen on the target system. Examining an individual user activity is not sufficient to draw an accurate conclusion of the user intention. Logging in from a remote location can indicate either a *legitimate* user logs in from outside of the regular infrastructure, or an *illegitimate* user logs in using stolen credentials. A framework is needed to reason on the user activities altogether.

We propose AttackTagger framework which is built upon *factor graph*, a probabilistic graphical model consisting of random variables and factor functions [3]. A random variable quantifies an observed user behavior or a hidden state (e.g., the user intention - benign, suspicious, or malicious). Relationships among variables are defined by discrete factor functions. A factor function $imply_1(A, B)$ means B is often followed by A. For example, in the context of masquerade attacks, an attacker impersonates as a legitimate user, e.g., by logging in to the target system from the attacker's computer

using stolen credentials. In that case, the factor function means *When a user logs in from an unregistered computer (A), the user is likely to be suspicious (B)*. Each factor does not necessary capture entire user behaviors leading to an attack, rather, a factor only captures a part of the attack and can influence other factors. For example, *When a user is suspicious (B) and the user is downloading an executable file from an unknown remote server (C), then the user is likely to be malicious (D).*

While traditional signature-based detection methods often identify a specific stage of an attack, our AttackTagger framework uses factor functions to reason about stages of an attack thoroughly. A factor function $imply_2$(B,C,D) can use the existing result of the previous factor $imply_1$(A,B) to determine that the user is malicious. That means an entire sequence of hidden user states is jointly inferred as a whole, based on observed user behaviors and defined factors. This design allows AttackTagger to detect a variety of attacks relatively early and uncover the attacks that were undetected by security analysts.

As a case study, our experiment uses incident data of 116 security incidents over a six-year period (2008-2013) at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. Each incident includes data from a number of sources: an incident report in free text format, raw logs (e.g., network flows, syslogs, and security alerts), and user profiles (e.g., a user role or user registered physical location). Using factor graph as a framework allows AttackTagger to integrate user behaviors from a variety of data sources. As a result, AttackTagger can identify most of the malicious users relatively early (from minutes to hours before the system is misused). Note that all the NCSA incidents used in this study were in reality detected after the fact, i.e., after the attacker misused the system. In addition, AttackTagger identified hidden malicious users that were missed by NCSA security analysts.

The main contributions are:

- A novel application of Factor Graphs that integrates user behaviors from a variety of data sources for early attacks detection, i.e., before the system is misused.
- Design, implementation, and experimental evaluation of AttackTagger using data on a variety of security incidents collected over a six-year

period (2008-2013).

- Detection of six hidden malicious users that were missed by security analysts (in 65 tested incidents).
- Comparison of AttackTagger with other classification techniques including: rule-based classifiers, decision tree, and Support Vector Machine.

CHAPTER 2

A CREDENTIAL STEALING INCIDENT

In this section, we describe a credential stealing incident that occurred at NCSA and analyze the challenges of detecting such an incident promptly.

2.1 A credential-stealing incident (2008)

In May 2008, a sophisticated credential-stealing incident occurred at NCSA. Using a compromised user account credential (e.g., pairs of a username and a password), attackers logged into a gateway node at NCSA and injected credential collecting code into the secure shell daemon (SSHd) [1] of the node. Since NCSA computing infrastructures were shared among hundreds of users, many of them logged in to NCSA using the compromised gateway node. Thus, the attackers were able to collect a new set of credentials.

An excerpt of the raw logs of the incident is listed in Table 2.1. First, the attackers used the compromised credential to log into the gateway node from a remote host, i.e., a host which is located outside of NCSA computing infrastructure in the event e^0. Second, the attackers downloaded a source code file (vm.c) with a sensitive extension (.c) in the event e^1. A sensitive extension indicates either a source code file (e.g., .c, .sh) or an executable file (e.g., .exe). The attackers then compiled, and executed a root privilege escalation exploit (which was identified as CVE-2008-0600) on the compromised node. These actions were not captured by the monitoring systems at runtime, they were only revealed in the forensic analysis process after the incident, thus they were not shown in the raw log. In order to harvest credentials of users logging into the compromised node, the attackers escalated to the *root* user, and injected the credential collecting code into the original SSHd, forcing the SSHd to restart (which results in the *SIGHUP* signal in the event e^2). Each

[1] a widely deployed authentication service of UNIX systems

Raw log	*Event*
`sshd: Accepted <user> from <remote>`	e^0: `login remotely`
`HTTP GET vm.c (server6.bad-domain.com)`	e^1: `download sensitive`
`sshd: Received SIGHUP; restarting.`	e^2: `restart sys service`

Table 2.1: Example raw logs and events of an incident

raw log entry was automatically mapped to an event identifier using regular expression scripts.

In this incident, the attackers were not identified during the attack, neither by the security analysts nor by the deployed Intrusion Detection System (IDS). The collateral effect of the incident is: leaking credentials of subsequent users who logged into the compromised node. The leaked credentials can be used for subsequent attacks.

2.2 Characteristics of multi-staged attacks

The discussed incident is an example of a multi-staged attack, in which an attack i) spans an extended amount of time, and ii) involves several steps such as: stealing or brute-force guessing credentials, logging in remotely, downloading and executing privilege escalation exploits, installing backdoors, and staying dormant. On the other hand, single-staged attacks (typically remote exploits such as SQL injection or exploitation of VNC servers) are usually accomplished in a single execution step in a short amount of time (in terms of minutes) to launch the attack payload (e.g., reading hashed passwords from a database).

2.3 Challenges of detecting multi-staged attacks

Detecting a multi-staged attack is to identify the states of the involved users throughout the attack. A user can be in one of the three states: *benign, suspicious, or malicious*. For example, the *login remotely* event can be tagged with a *user state*, i.e., a *tag*. A user state can be *benign* (when a legitimate user logs in from the remote location as a part of his/her normal activity), or can be *suspicious* (when an illegitimate user uses stolen credentials to log in from the remote location).

In the above example, the single *login remotely* event itself provides insuf-

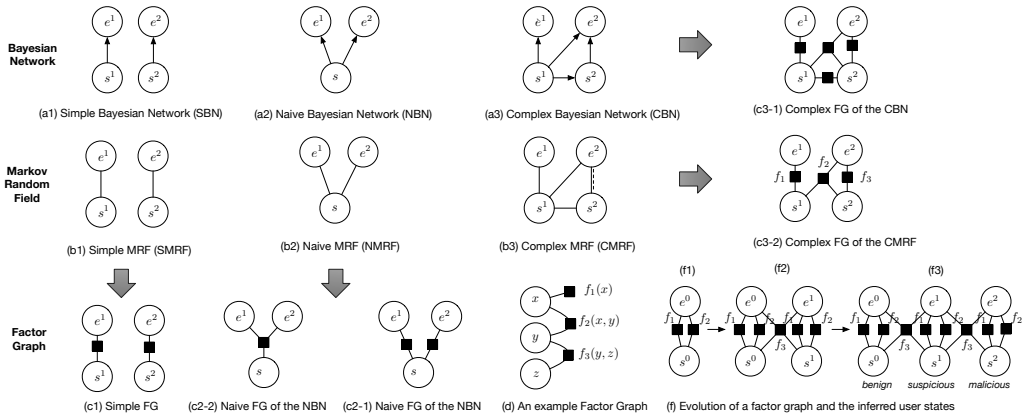

Figure 2.1: Illustrations of Bayesian Network, Markov Random Field, and Factor Graph to model security incidents.

ficient information to tag its state as *malicious*. Because this event does not indicate a security violation, such as modifying a system service without being a system administrator. Based solely on this event, it is more reasonable to tag its state as either *benign* or *suspicious*. In order to be more conclusive about which state to tag the event, intuitively we need further information. For example, the existing context of the system, the user profile, and the information from the subsequent events. Due to this reason, the usual approach of using *per-event classifiers* is not effective in detecting multi-staged attacks.

To detect single-staged attacks, existing IDSes often employ *per-event classifiers*, which assign a tag to an event using rules or signatures. In this example, given the event e^2 (*restart system service* in Table 2.1), a possible tag $s^2 = benign$ could mean that the event corresponds to a maintenance activity of a benign user, e.g., the user upgrades the SSHd to a newer version. The tag $s^2 = benign$ is plausible because an upgrade of the SSHd often requires restarting of the current SSHd in order to load the updated binaries.

However, *per-event classifiers* consider each event individually and do not take advantage of the knowledge on an event sequence. For example, when it is known that the previous observed event was tagged as *suspicious*, the current event e^2 can be tagged differently in light of this knowledge. In such a case, a more likely tag $s^2 = malicious$ could indicate that the event e^2 corresponds to an unauthorized activity of an already suspicious user, who is attempting to inject malicious code into the SSHd, thus forcing it to restart. A framework is needed to reason on the user events altogether.

CHAPTER 3

PROBABILISTIC GRAPHICAL MODELS

In this section, we provide an overview of probabilistic graphical models in modeling security incidents. Probabilistic graphical models (PGM) are graph-based representations of dependencies among random variables. PGMs such as Bayesian Networks (BNs), Markov Random Fields (MRFs), and Factor Graphs (FGs) can compactly represent complex joint distributions of random variables over a high-dimensional space [3]. While BNs and MRFs have been successfully employed in a variety of domains such as medical condition diagnosis or entity extraction from text [4, 5], the use of FGs in security domains has not been explored. We found that FG is more suitable to model security incidents since FG can subsume both BN and MRF [3].

When using PGMs to model security incidents, the random variables consist of *observed user events* (derived from incident reports and raw logs) and *hidden user states* associated with the events. Specifically, in the credential stealing incident example (Table 2.1), we consider the sequence of the observed events $E = \{e^1 = download\ sensitive, e^2 = restart\ system\ service\}$ and the sequence of the corresponding user states is $S = \{s^1, s^2\}$. Based on the observed user events, PGMs are defined to capture the dependencies among the random variables. We compare the use of BN, MRF, and FG to model the example incident as follows.

3.1 Bayesian Networks.

A Bayesian Network (BN) is a type of probabilistic graphical model representing conditional dependencies among random variables using a directed acyclic graphs $G = (V, E)$. Each vertex $v \in V$ corresponds to a random variable; each edge $e \in E$ represents a causal relation between two variables (e.g., X causes Y).

A simple Bayesian Network (SBN) models the dependencies of the observed events E and the user states S in Figure 2.1 a1. This model assumes that the observed events E are independent and the event-state dependencies are causal relations: an event e^i depends only on its user state variable s^i ($s^1 \rightarrow e^1$, and $s^2 \rightarrow e^2$). Because of the independent assumption, the SBN cannot capture the dependencies of a sequence of events and the corresponding sequence of the user states. An example of such dependencies is: an event e^i is not only caused by its corresponding user state s^i, but also caused by a previous user state s^{i-1}.

In Figure 2.1 a2, a Naive Bayesian Network (NBN) models the dependencies of all the observed events E and a single user state s. NBN assumes an event is independent of others. Thus, the conditional dependencies are simplified: each event e^i depends only on the single user state variable s (e.g., $s \rightarrow e^1, s \rightarrow e^2$). NBN is not suitable for an early detection of attacks, since it operates on a complete sequence of the observed events E to infer the user state. To detect attacks in real-time, a detection system should determine the user state after arrival of each new observed event (i.e., based on an incomplete set of the observed events).

In Figure 2.1 a3, a more complex BN (CBN) models the sequential dependencies among a group of random variables. Consider the user states s^1, s^2 and the observed event e^2, to model dependencies among the three variables, the CBN must make an assumption of the pairwise conditional dependencies among the random variables ($s^1 \rightarrow e^2, s^1 \rightarrow s^2, s^2 \rightarrow e^2$). The disadvantages of this CBN are as follows. Despite that CBN is relatively simple in this example incident, the number of pairwise dependencies among a group of variables in a CBN can grow quickly as the number of variables in the group increases. When a group involves n, variables, a CBN may have to define up to $n(n-1)/2$ pairwise dependencies in the group, making the CBN much more complex. Moreover, in some domains (e.g., natural language processing), a causal relation between a pair of variables cannot be claimed (only a non-causal relation can be assumed). This non-causal relation is discussed in a more detail in the part of speech tagging example in the following sub-section.

The discussed BN models allow explicit representation of conditional dependencies among, however, they become more complex as the number of random variables grow.

3.2 Markov Random Fields

A Markov Random Field (MRF) is a type of probabilistic graphical model representing relations among random variables using an undirected graph $G = (V, E)$. Each vertex $v \in V$ corresponds to a random variable; each an edge $e \in E$ represents a relation between two variables.

A simple Markov Random Field (SMRF) depicted in the Figure 2.1 b1 is an equivalent model to the simple Bayesian Network (SBN) in the Figure 2.1 a1. In the SMRF, the dependency among e^1 and s^1 is represented by a function $\phi(e^1, s^1)$. In this case $\phi(e^1, s^1)$ is defined as a conditional probability mass function $p(e^1|s^1)$.

We briefly discuss the characteristics of a MRF. Let $n(v)$ is the set of v's neighbors (i.e., the vertices that are directly connected to v by a single edge). G is an MRF if its random variables satisfy the local Markov property: a variable is conditionally independent on other variables given its neighbors. Variables in a MRF are grouped into cliques, in which all variables in the cliques must be pairwise connected. A clique is a maximal clique if it cannot be extended by including an adjacent variable to the clique.

A complex joint probability mass function of variables in an MRF can be factorized into a product of simpler local functions, defined on the set of maximal cliques in the MRF. Each local function corresponds to a clique and describes relations of variables in the clique. The factorization simplifies both representation of MRFs and computation of the joint probability mass function.

MRFs are used in domains where variable relations are non-causal, e.g., it is natural to indicate that X correlates with Y, rather than X causes Y [5]. For example, in part of speech (POS) tagging, a word (an observed variable) are often tagged with a part of speech (a hidden variable), e.g., a noun or a verb, based on the word itself and its context. Depending on the context (*my research* or *I research*), the word *research* can be correlated with a different part of speech. In this example, the relation between the observed word (research) and its part of speech is non-causal.

When the variable dependencies are simple (e.g., dependencies among a group of two or three variables), a MRF can be used as an alternative representation to a BN. Figure 2.1 b1 and Figure 2.1 b2 depicts an equivalent MRF model to its BN models (Figure 2.1 a1 and Figure 2.1 a1), where the

directed edges in the BNs has been replaced by the undirected edges. Note that a MRF does not make any assumption of the causal relation among the variables. An arbitrary function can be used to define the relation among the variables.

In security domain and particularly in our example, an event (an observed variable) and a user state (a hidden variable) can have a non-causal relation. For example, for the event that a user logs in remotely, it is usually that the user is traveling (i.e., the user state is benign), rather than an attacker is impersonating the user using the stolen user account (i.e., the user state is malicious).

An MRF model (Figure 2.1 b3) illustrates non-causal dependencies among the events and the user states. Consider a group of variables s^1, s^2, e^2, they can have following cliques: the two-variable cliques e^2, s^2 (represented by a dotted line), and the three-variable clique e^2, s^1, s^2. In the cliques, one can define local functions of an event and the corresponding user state (e.g., $\phi(e_2, s_2)$); or an event, the corresponding user state, and the previous user state (e.g., $\phi(e_2, s_1, s_2)$).

In the example MRF, the function of a clique simplifies the representation of the MRF compared to the equivalent representation in a BN. For example, the clique e_2, s_1, s_2 in the MRF (Figure 2.1 b3) simply uses one local function, instead of using the three pair-wise causal dependencies, which requires three conditional probability mass functions to build the equivalent BN model (Figure 2.1 a3). Despite a simpler representation in MRFs, a practitioner can still model complex dependencies by factorizing a local function into a product of smaller functions, e.g., the $\phi(e_2, s_1, s_2)$ can be factorized into the three functions representing the pairwise causal dependencies between the variables in the clique.

Advantages and disadvantages of using MRFs are as follows. In MRFs, the use of one local function per clique avoids the making of explicit assumptions about causal dependencies among variables as in BNs. However, there is an overlap between the three-variable clique s^1, s^2, e^2 and the two-variable clique s^2, e^2 which cannot be naturally expressed using MRFs, in which a MRF is built upon maximal cliques.

The above analysis suggests a common representation of both BNs and MRFs, which is Factor Graph.

3.3 Factor Graphs

A Factor Graph is a type of probabilistic graphical model that can describe complex dependencies among random variables using an undirected graph representation, specifically a bipartite graph. The bipartite graph representation consists of variable nodes representing random variables, factor nodes representing local functions (or factor functions), and edges connecting the two types of nodes. Variable dependencies in a factor graph are expressed using a global function, which is factored into a product of local functions.

Suppose a global function $g(x, y, z)$ of the three variables x, y, z can be factored as a product of the local functions f_1, f_2, f_3 as follows: $g(x, y, z) = f_1(x)f_2(x, y)f_3(y, z)$. In this example, the variable nodes are x, y, z, the factor nodes are f_1, f_2, f_3, and the edges are shown in the Figure 2.1 d.

Factor graphs are simpler and more expressive than BNs and MRFs. In factor graphs, factor functions explicitly identify functional relations among variables, including causal relations (BNs) and non-causal relations (MRFs). Moreover, complex dependencies in BNs and MRFs can be subsumed using factor graphs [3]. A factor function can be used to represent multiple causal relations or non-causal relations. The use of factor functions can simplify a complex BN or a complex MRF by reducing the number of functional relations that has to be defined. Equivalent FG representations of BNs and MRFs are shown in the Figure 2.1 c1, c2-1, c2-2, c3-1, and c3-2. A detailed discussion on conversions among FGs, BNs, and MRFs can be found in [3].

The universal graph representation of FGs has led to development of effective inference algorithms (e.g., Gibbs sampling or message passing) [6, 3]. Since FGs offer the same representation for both BNs and MRFs, these algorithms can be used for existing BNs and MRFs models when they are converted to FGs.

In the practice of the security domain, using factor graphs is more flexible to define different types of relations among the events to the user state com-

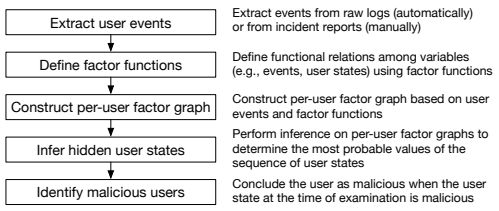

Figure 3.1: Process of modeling using factor graph

pared to Bayesian Network and Markov Random Field. Specifically, using FG allows capturing sequential relation among events and enables integration of external knowledge (e.g., expert knowledge or knowledge of a user profile) to its framework. Our framework is built upon factor graph.

CHAPTER 4

FRAMEWORK OVERVIEW

In this section, we provide an overview of using factor graphs in our framework to model the example incident (Section 2). The main modeling steps, from extraction of events to making a decision about a user, are illustrated in Figure 3.1.

Step 1: Extract user events. User events can be extracted automatically from raw logs (using regular expression scripts) or manually from incident reports. In the example incident, the sequence of observed events was $E = \{e^0 = login\ remotely,\ e^1 = download\ sensitive,$
$e^2 = restart\ system\ service\}$. The event sequence is associated with a sequence of hidden user states $S = \{s^0, s^1, s^2\}$.

Step 2: Define factor functions. A factor function defines the relations among variables. Each factor function is a discrete function that inputs random variables, e.g., observed user events or hidden user states, and outputs a discrete value indicating relations among the inputs.

For example, a *Type-1* factor function $f(e, s)$ can be defined to imply the relation if e happens then s. Suppose we have the relation "if a user downloads a file with a sensitive extension then the user is suspicious". Here we have two variables: one event e=*download sensitive* and a state s = *suspicious*. The function $f(e, s)$ returns 1 if e=*download sensitive* and s = *suspicious*; and returns 0 otherwise. For example, the function f_1 in the Figure 2.1 f is defined as follows.

$$f_1(e^t, s^t) = \begin{cases} 1 & \text{if } s^t = suspicious \\ & \ \& \ e^t = download\ sensitive \\ 0 & otherwise \end{cases}$$

Similarly, a factor function can capture the case when a system administrator restarts a SSHd, which is likely a maintenance activity. The function f_2 in the Figure 2.1 f is defined as follow.

$$f_2(e^t, s^t) = \begin{cases} 1 & \text{if } s^t = benign \\ & \& \ e^t = restart\ sys\ service \\ 0 & otherwise \end{cases}$$

The function f_2 returns 1 when the user event is restarting a system service (i.e., SSHd in our example) and the user state is *benign*. It returns 0 otherwise.

To identify a user state based on the context of an event, a more complex function can involve more variables, e.g., the previous user state or the previous event. A *Type-2* factor function $f(e^t, e^{t-1}, s^t, s^{t-1})$ defines the relation among a user state s^t, its previous user state s^{t-1}, and observed events e^{t-1}, e^t. For example, the function f_3 in the Figure 2.1 f is defined as follows.

$$f_3(e^t, e^{t-1}, s^t, s^{t-1}) = \begin{cases} 1 & \text{if } s^{t-1} = suspicious \\ & \& \ s^t = malicious \\ & \& \ e^t = restart\ sys\ service \\ 0 & otherwise \end{cases}$$

The function f_3 returns 1 when an already *suspicious* user restarts a system service and the current user state is *malicious*. Given the event *restart system service*, it identifies the current user state in the context that the previous user state is *suspicious*. It returns 0 otherwise.

In this illustration, we consider only two types of factors: Type-1 factors and Type-2 factors. More factor functions can be manually defined to capture user state in the context of events, user profiles, and to embed expert knowledge into factor graphs. A more formal definition and discussion on Type-1 and Type-2 factors is provided in Section 5.

Step 3: Construct per-user factor graph. Given a sequence of user events E and a defined set of factor functions F, a factor graph is automatically constructed for that user, namely *per-user factor graph*. Each factor connect its corresponding user events and user states.

Figure 2.1 f demonstrates the *evolution* of a per-user factor graph, as new events are observed. When only one event is observed, the factor graph only contains two Type-1 factors (f_1, f_2) for the event e^0 and its corresponding state s^0. When two events are observed, the two Type-1 factors are used to connect the new event e^1 and its corresponding state s^1. In addition, the factor graph has a Type-2 factor (f_3) connecting both the events and their states: e^0, s^0, e^1, s^1. As more events are observed, the same set of defined factors (f_1, f_2, f_3) is used to connect the new events.

Step 4: Infer hidden user states. Given a per-user factor graph (Figure 2.1 f), a possible sequence of user states S is automatically evaluated by summing the weighted factor functions F.

$$score(S|E) = \sum_{f \in F} w_f f(c_f)$$

where w_f is the weight of the factor function f, and c_f is the set of inputs to the factor function f. The sequence of user states that has the highest score represents the most probable sequence corresponding to the event sequence observed until that point.

A naive approach is to iterate over possible sequences of user states in the constructed factor graph, the most probable sequence is S=$\{$*benign, suspicious, malicious*$\}$ as shown in the Figure 2.1 f. The obtained sequence has the highest score and corresponds to the most probable sequence of user states, determined based on the events observed up to that point in time.

In our model, we compute the probabilities of the user state sequences using more efficient methods. Refer to Section 5 for details of our inference process.

Step 5: Conclude malicious users. The compromised user is automatically identified when the user state at a time of observation is *malicious*.

Note that most steps in our framework are automated, except the Step-2 (defining factor functions), which requires expert knowledge. Using our framework, security analysts can quickly examine the user states to identify the transition of a user from benign to suspicious and malicious, without having to manually examine a large amount of raw logs. As a result, security analysts have more time to response to security incidents or to place additional monitoring of suspicious users to uncover potentially unauthorized activities.

CHAPTER 5

ATTACKTAGGER MODEL

In this section, we provide a generic formulation of the factor graph model for incident modeling and detection.

5.1 Preliminaries.

Consider a *user* u of a target system. The user is characterized by a *user profile* U, which is a vector of *user attributes*. Examples of the user attributes are shown in Table 5.2. U does not change during usage of the target system. In order to capture the user activities in the target system, monitors are deployed at various system and network levels to collect raw logs. At runtime, each log entry is automatically converted to a discrete *event* e.

An *event* e^t indicates an important activity in the target system (e.g., restart of a system service), or an alert on a suspicious activity (e.g., download of a file with a sensitive extension .c, .sh, or .exe). The set of events $\mathcal{E}$ is system-specific and is pre-defined based on: the capabilities of the system/application-level monitoring tools (e.g., events generated by the IDS) and expert knowledge of the target system. Examples of the events are shown in Table 5.2 As the target system evolves, $\mathcal{E}$ can be extended using expert knowledge.

A *user session* is a sequence of *user events* $E^t = \{e^1 \dots e^t\}$ from the time when user started using the target system until the observation time t.

A *user state* $s^t \in \mathcal{S} = \{\text{benign, suspicious, malicious}\}$ is a hidden variable whose value determines the suspiciousness of the user. The initial user state is determined based on the user profile. A user is *benign* when no security event (i.e., a policy-violation event or an alert) has been observed for the user and the user profile is clean of suspicions. For example, the initial user state is *benign* if the user has just logged in and the user account has

not been compromised in the past. As the user proceeds, each user state s^i is associated with the arriving event e^i of the user. A user is *suspicious* when there is more than one security events that has been observed for the user, however, further information is needed to make a conclusion. A user is *malicious* when the user is determined to violate a security policy or there is enough information to conclude the malicious intention of the user.

While in this study we limit the user state to the three defined states, more fine-grained user states can be defined.

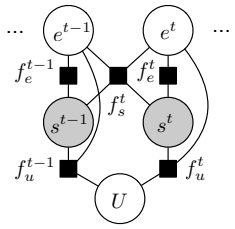

Figure 5.1: A snapshot of the factor graph model of an attack at a time t

5.2 Characterization of factor functions

A factor function can capture: i) the relation between a user state and an event, ii) the relation among a user state and the earlier events/states observed during the progression of the incident, and iii) the relation among a user state and a user profile. Defining such factor functions can assert a user state with a higher degree of confidence. Factor functions can be characterized into the three main types of relations: Type-1 (event-state), Type-2 (state-state), and Type-3 (user-state).

Type-1. A factor node $f_e^t(e, s)$ captures the relation between the event e and the hidden state variables s.

Type-2. A factor node $f_s^t(e^{t-1}, e^t, s^{t-1}, s^t)$ captures the relation among the hidden states s^{t-1}, s^t, events e^{t-1} and e^t.

Type-3. A factor node $f_u^{t-1}(U, e^{t-1}, s^{t-1})$ captures the relation among a user profile U, an event e^{t-1}, and a hidden state s^{t-1}.

A factor function has a discrete value output 0 or 1. Each factor $f(x)$ is defined by an *indicator function* $I_A(x) : X \rightarrow \{0, 1\}$ that returns 1 if an input $x \in X$ is a match with A and 0 otherwise, where A is a tuple of values and x is a tuple of variables. A match between x and A (i.e., $x = A$) means that the values of variables in x is the same as that of A, element-wise.

$$I_A(x) = \begin{cases} 1 & \text{if } x = A \\ 0 & \textit{otherwise} \end{cases}$$

For example, in Section 3.3, we defined a factor function f_3 for capturing the user state associated with the event *restart system service*, given that the previous observed event was labeled as *suspicious*. This factor function belongs to the Type-2 category and can be defined using indicator function as follows. Let A be a tuple of ($e^{t-1} = e^*$, e^t= restart system service, $s^{t-1} =$

Symbol	*Description*
$e, \mathcal{E}, E$	Event, event set, sequence of events
u, U	User, user profile
f, F	Factor function, set of factor functions
$s_u, \mathbb{S}$	User state, user state set

Table 5.1: Notation of variables used in our model

suspicious, $s^t = malicious$). The notation e^* for the event e^{t-1} means that the event e^{t-1} can be any of the events in the event set $\mathcal{E}$. Using our definition, the factor function is defined as $f_s^t(e^{t-1}, e^t, s^{t-1}, s^t) = I_A(e^{t-1}, e^t, s^{t-1}, s^t)$. We illustrate real factor functions, derived from our real-world incidents dataset, in Section 6.

Note that in theory, more complex factor functions (e.g., to relate multiple events) can be defined, however, they produce more complex factor graphs.

5.3 An example factor graph

Figure 5.1 a shows a generic factor graph model of an attack. The notation and the meaning of the variables of an attack in the model are given in Table 5.1. Variable nodes consist of observed variables (U, E^t) and hidden variables (S^t). Factor nodes represent discrete functions describing functional relations among the observed variables and hidden variables. For the purpose of illustration, Figure 5.1 a shows four factors for the events e^{t-1} and e^t. In our model, the factors are defined for the sequence of events from e^0 (when a user begins using the system) to e^t (time t).

Inference of hidden user states AttackTagger evaluates a sequence of the user states to identify the malicious user. At a time t, given observed variables U, E^t, our approach finds the *most probable* ***sequence*** *of the hidden user states* S^t that correspond to the observed variables. The compromised user is identified by the values of user state s^t. Specifically, a user state $s_u^t = suspicious$ should result in additional monitoring efforts. Otherwise, a user is concluded to be compromised if the user state is $s_u^t = malicious$.

The most probable hidden state variables S^t can be estimated by maximizing the *joint probability distribution function* $P(U, E^t, S^t)$ of the observed variables and hidden state variables. Note that in this generic formulation, we define a proper joint distribution instead of computing the *score* of S^t as seen in the Section 4. Let $F = \{F_{em}, F_s, F_u\}$ be the set of factor function

User attributes	Registered physical location (categorical) Number of days since the last login (integer) Has been compromised previously (boolean)
Event	Login remotely (using secure shell) Download sensitive file (.exe, .c) Restart system service (secure shell server) Large number of incorrect login attempts Large number of open network connections

Table 5.2: Examples of user attributes and events

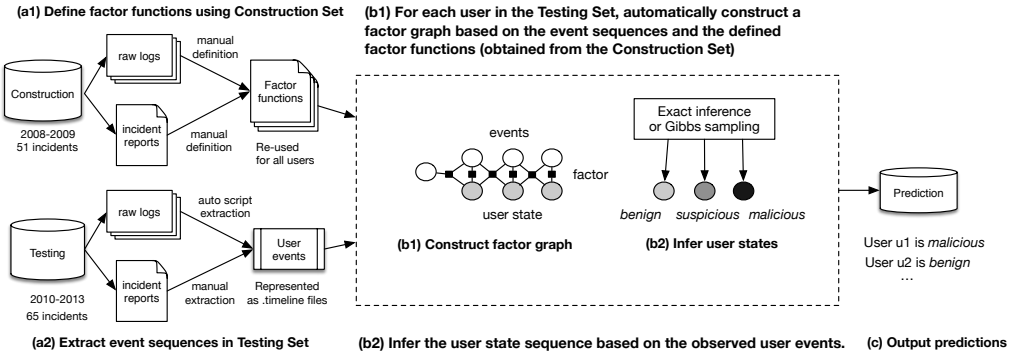

Figure 5.2: Experiment flow with input is incident report or raw logs, and output is prediction of malicious users.

sets for Type-1, Type-2, and Type-3 factor functions, respectively. The joint distribution can be factorized as follows:

$$P(U, E^t, S^t) = \frac{1}{Z} \prod_{f \in F} f(c_f)$$

where c_f is the set of inputs to a factor function f. The inputs are from observed and hidden state variables. We use Z as the normalization factor to ensure that $P(U, E^t, S^t)$ is a proper probability distribution function. The normalization factor Z can be computed by summing values of f over all possible combinations of the variables $\{U, E^t, S^t\}$. Since the observed variables (U, E^t) are known, the most probable hidden state variables S^t can be found by enumerating (brute-forcing) all possible sequences of the hidden state variables and returning the sequence that maximizes $P(U, E^t, S^t)$:

$$S^t = \arg\max_{S^t} \frac{1}{Z} \prod_{f \in F} f(c_f)$$

Although the brute-force approach can give an exact result for the most probable hidden state variables S^t, naive enumeration of all possible sequence combinations is costly. Since each state variable s has a discrete value, approximation methods such as Gibbs sampling, which have been successfully utilized in computer vision and natural language processing, can be used to estimate the most probable sequence of hidden state variables S^t [6, 7].

Gibbs sampling on factor graphs. Given a constructed factor graph of a user session, the user state sequence can be approximated using Gibbs sampling, a popular inference algorithm that can be used on factor graph. In a real-world detection system that requires inference in near real-time, Gibbs sampling can produce an approximate result within a predefined bounded time (e.g.,

the algorithm stops after 100 iterations). Performance and ease of use are the main reasons of selecting Gibbs sampling rather than using exact inference (for which the complexity is exponential to the length of the sequence). We briefly describe how a Gibbs sampler works.

A Gibbs sampler runs over N iterations. It starts with a random user state sequence at iteration 0. At iteration 1, it samples a new user state, starting at a user state s^0. That sampling process is conditioned on the value of the previous user state sequence and the factor graph. In the next step, this sampling process is repeated for the next user state s^i until it reach the last user state s^n. This concludes the sampling process for a user state sequence at the iteration 1. The Gibbs sampler repeats the iteration process and stops when it reaches one of the two termination conditions: i) N iterations or ii) the user state sequence converged (i.e., the user state sequence does not change from iteration k to iteration $k + 1$).

CHAPTER 6

EVALUATION OF ATTACKTAGGER

This section describes the incident dataset, generation of the factor functions, construction of factor graphs, and evaluation of AttackTagger.

6.1 Dataset

We use data on 116 real-world security incidents observed at NCSA during a six-year (2008-2013) period. The incidents contain sophisticated attacks such as tampering system services (e.g., SSHd) to steal credentials, misusing of computing infrastructure to build botnets/send spam emails/launch denial of service attacks, or remotely exploiting Virtual Network Computing servers to get a system shell.

Incident data. Incident data includes incident reports and raw logs. For each incident in our data set, we obtained its *incident report* manually created by NCSA security analysts in free format text. Each incident report contains a detailed post-mortem analysis of the incident, including alerts generated by NCSA security monitoring tools. An incident report often includes snippets of *raw logs* (e.g., syslogs, network flows, and Bro IDS logs) associated with malicious activities. Incident reports may also contain extra information about the incident, such as records of emails exchanged among security analysts during the incident.

Most incidents considered in our dataset are related to multi-staged attacks, in which an attack spans a duration of 24 hours to 72 hours. Thus, for a subset of security incidents we also gathered their *raw logs* for a period of 24 hours to 72 hours before and after the NCSA security team detected a given incident. This duration of time is sufficiently long to cover most of the traces of attacks in our dataset. Since the data retention policy changed during the time (6 years) when incident data were collected, the raw logs were only available for a subset of the incidents (Table 6.1). The raw logs

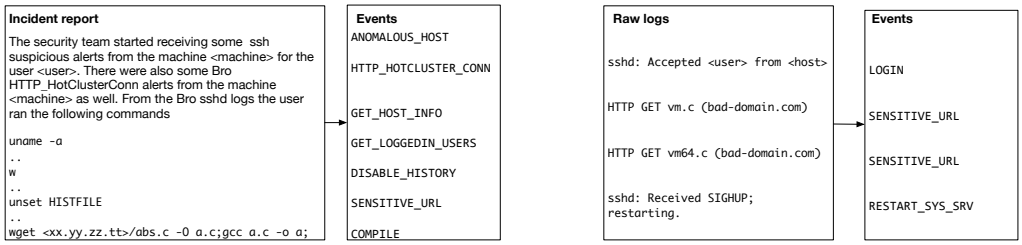

Figure 6.1: Manual and automatic conversion of incident reports and raw logs to events

are valuable since they captured activities of both benign and malicious users in the incidents.

Construction Set and Testing Set. The data on 116 incidents have been partitioned into two disjoint sets: (i) one *Construction Set* of 51 incidents collected during the 2008-2009 period, (these incident data are used to extract the set of events observed during the incidents and to define the factor functions), and (ii) *Testing Set* of 65 incidents collected during the 2010-2013 period (these incident data are used to construct the factor graphs for each user and to evaluate the detection capabilities of the constructed factor graph). The partition is based on the following. In the 2008-2009 period, a subset of the incidents were credential stealing incidents. Our conjecture is: in many incidents observed during the 2010-2013 period, the attackers used the stolen credentials to infiltrate the NCSA infrastructure. As a result, our model has been constructed using the Construction Set and evaluated using the Testing Set. Figure 5.2 summarizes the two disjoint sets.

Ground truth. The benign and malicious users provided by incident data are considered the *ground truth* in our evaluation. The 51 incident reports and 18 incident raw logs in the Construction Set identified 46 malicious users, 5 benign users misclassified as malicious by NCSA security analysts, and 2,612 benign users who were involved in the incidents. The 65 incident reports and 5 incident raw logs for the Testing Set identified 62 malicious users, 3 benign users misclassified as malicious users by the NCSA security analysts, and 1,253 benign users who were involved in the incidents. In total, nearly four thousand users logged into the target system during the six year (2008-2013) period. When counting the number of users, the same

Data Set	Available Data	
	Incident reports	Raw logs
Construction Set (51)	51	18
Testing Set (65)	65	5

Table 6.1: Summary of the incident dataset

user u observed in separated incidents is considered as separated users. Note that there could be hidden malicious users who were not indicated in the incident reports by the NCSA security analysts. Our model detected six hidden malicious users (Section 6.2).

6.1.1 Extraction of events and definition of factor functions

Given the data for an incident, we extracted *user sessions* from the incident report and the raw logs. Usually there were several hundred user sessions during a time window of 24 to 72 hours (the typical duration of an incident in our dataset).

Extraction of Events. A sequence of events was extracted from each user session. In the case of a raw log snippet listed in the written incident report, we used the regular expression scripts to automatically extract the corresponding events. In the case of a textual description of a user activity, we manually extracted a list of events in an order that matched the textual description (to the best of our knowledge). Note that the textual descriptions often do not include an accurate timestamp associated with each event, but rather were arranged in an order that we inferred from the incident reports. To illustrate our manual extraction process, an excerpt of a written report and the extracted events are given in Figure 6.1 a1.

When the raw logs corresponding to a user session were available, they were automatically converted to a sequence of events using regular expression scripts. Each log entry in the raw logs were mapped to an event, which consists of a timestamp, the user identifier, and a unique event identifier. Through that process, the orders of the events are guaranteed, since we used the exact timestamp. Examples of a log entry and the corresponding event are illustrated in Figure 6.1 a2. Note that for incidents for which we have both the incident report and the raw logs, we combined the events extracted from the written report and the raw logs.

Preprocessing of incident data resulted in a list of *.timeline* files for each incident in the Testing Set. Each file contains a sequence of events for a user and the ground truth information, indicating whether the user is malicious or not. There were 1,315 users and 65,389 events for the incidents in the Testing Set.

Definition of factor functions. The factor functions were defined manu-

ally using incident data from the Construction Set and experts' knowledge of the system. In the following, we illustrate the three types of factor functions derived from real incidents in the Construction Set.

A Type-1 factor function can directly associate an *event* with a *malicious user state* when the event is an obvious violation of a security policy, e.g., a simple factor function could capture the following relation: *the user downloads a known exploit/malware file* (the observed event) implies *the user is malicious* (the assigned user state). A Type-1 factor can also capture a less obvious policy violation, e.g., *the user logs in from a remote location* (the observed event) implies the user is *suspicious* (the assigned user state). Note that the accuracy of the established association between the *event* and the *user state* depends on the representativeness of the data on the past incidents and the confidence of the expert.

More advanced factor functions, i.e., Type-2 and Type-3, take into account the knowledge of the user state, as determined based on the earlier events observed during the progression of the incident. As a result, Type-2 and Type-3 factor functions can assert the user state with a higher degree of confidence. For example, a Type-2 factor function could assert the following relation: *the user download a file with a sensitive extension (e.g., .c, .sh, or .exe)* (the most recent event) and *the user state is suspicious* (determined based on an earlier event) imply *the user state is malicious*. Type-3 factor functions are ***extensions*** of the Type-2 factor functions, in which the user profile is taken into account, thanks to the flexibility of factor graph. For example, a Type-3 factor function could assert the following relation: *a user has been previously compromised* (established based on the user profile) and *the user state is suspicious* (determined based on an earlier event) and *the user restarts a system service* (the most recent event) imply *the user state is malicious*.

Following our illustrated definitions, a practitioner can construct their own factor functions based on their events and expert knowledge of their target system. We defined a total of 65 factors, in which there are 29 Type-1 factors, 34 Type-2 factors, and 2 Type-3 factors. Due to the space limitation, a complete list of factor functions will be included in an online appendix later.

6.1.2 Construction and inference on factor graph

Given the defined factor functions, we construct a factor graph for each user session (*per-user factor graph*) and perform inference on the constructed factor graph.

Construction of factor graphs. Each per-user factor graph was used to re-evaluate the user state (benign, suspicious, or malicious) on arrival of a new event. The resulting factor graphs were dense with many edges, since the entire defined factor functions have to link the entire events in the user event sequence. For a sequence of n user events, a Type-1 factor function links each event e^i with the user state s^i ($i = 1..n$). The process is repeated for the Type-2 and Type-3 factor functions with their corresponding events and user states. Figure 5.2 shows the experimental flow including the process of constructing a factor graph for each user.

In our experiments, the weights for the factor functions were assumed to be equal (i.e., the weight was 1). No training was performed to obtain the weights. The main difficulty in determining weight was the required human supervision for labeling each event with a user state. A value of a user state must be assigned for each observed event (i.e., whether the corresponding user state of the event is benign, suspicious, or malicious), and that is an arduous manual process taking into account about 300,000 observed events (2008-2013). Despite the use of equal factor weights, our model still achieves a good detection performance compared with detection by security analysts (Section 6.2).

Inference of user states on factor graphs. In Figure 5.2 b2, given a constructed factor graph of a user session, the user state sequence was approximated using Gibbs sampling [8], as described in Section 5.

We implemented our inference process in Python, using the Gibbs sampler in the OpenGM library to estimate the hidden state variables [9]. We observed that our inference converges after 100 iterations. Runtime performance of our model was evaluated on a desktop running Ubuntu 12.04 on Intel i5-2320 CPU at 3.00 GHz with 6 GB of RAM.

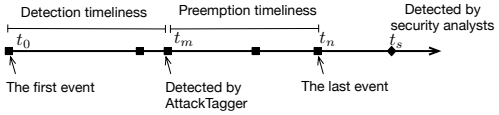

Figure 6.2: An attack timeline: the first event is observed at t_0; AttackTagger detects the attack at t_m; the attack finishes at t_n; security analysts detect the attack at t_s. Each square dot is an event related to the attack.

6.2 Empirical results

Our model was able to detect most of the malicious users (74.2%) relatively early (i.e., before system misuse). More importantly, our model uncovered a set of six hidden malicious users, which have not been discovered by NCSA security analysts. In this section, we describe how we analyzed detection timeliness and detection accuracy of our model using the Testing Set.

6.2.1 Timestamp and ordering of events.

We use *Lamport timestamp* (or logical clock), to establish the relative order of events [10]. The Lamport timestamp was used because *absolute timestamps* of events were not available for most of the incidents in our dataset.

Each event in a user session was assigned with a Lamport timestamp (specifying the order of events) or an absolute timestamp. For example, when a user session had a single event a, its Lamport timestamp was $\mathrm{C}(a) = 1$. As more events are observed, the events were assigned increasing values of the Lamport timestamp, such that if an event a happened before b, then $\mathrm{C}(a) < \mathrm{C}(b)$. Note that for incidents for which raw logs were available, each event was assigned an *absolute timestamp* in addition to its Lamport timestamp.

Figure 6.2 illustrates an event timeline of a malicious user. In the following, we refer to a timestamp as either Lamport timestamp or absolute timestamp, depending on the context. Consider a sequence of events, t_0 is the timestamp of the first observed event, t_m is the timestamp when AttackTagger concludes the user is malicious, t_n is the timestamp of the last observed event, and t_s is the timestamp when the malicious user is detected by a security analyst. We define the *attack duration* t_a of the malicious user to be given by $t_a = t_n - t_0$. A *Lamport attack duration* or an *absolute attack duration* can be derived from that formula. In practice, a larger Lamport attack duration (expressed in the number of events) corresponds to a larger number of events during the

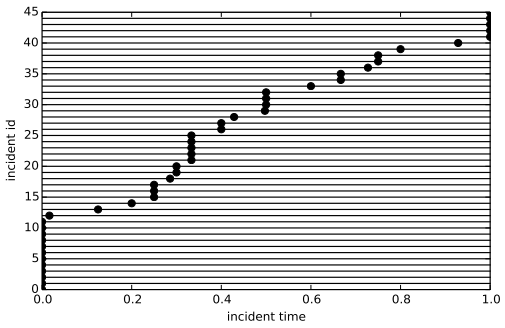

Figure 6.3: The x axis is the Lamport attack duration (incident time) of the malicious users normalized to the range [0-1]. Each row (incident id) in the y axis is a detected malicious user in an incident. The dot in a row represents the time when the malicious user was detected by AttackTagger.

attack and indirectly corresponds to an absolute attack duration (expressed in seconds, minutes, or hours). Note that to measure the absolute attack duration, we need an absolute timestamp to be associated with each event.

Note that all reported incidents were discovered by NCSA security analysts after system misuse, when attack payloads had already been executed; that means $t_s \geq t_n$. Our objective is to improve the detection time of the incidents, i.e., detect a progressing attack as early as possible.

6.2.2 Detection timeliness and preemption timeliness

We use two metrics to characterize the detection capabilities of our approach.

Detection timeliness characterizes the responsiveness of an intrusion detection system to an attack. The detection timeliness is measured by $t_d = t_m - t_0$. In our experiments, a *Lamport detection timeliness* (LDT) was computed using Lamport timestamps associated with each event. An LDT corresponds to the number of events observed from the start of a user session until the determination that the user was malicious.

In addition, for incidents which raw logs were available, we computed its *absolute detection timeliness* (ADT) using the absolute timestamp associated with each event. ADT provides the absolute time duration from the start of the user session until the determination that the user was malicious. Note that the shorter detection timeliness is better.

Preemption timeliness characterizes the amount of time that a human or an automated system had to respond to an attack, from the time when a user was identified as malicious until the time of the last observed user

event. The preemption timeliness is measured by $t_p = t_a - t_m$. Note that preemption timeliness was measured only for incidents for which a ground truth on when the attack was stopped was available.

In our experiment, a *Lamport preemption timeliness* (LPT) was computed using the Lamport timestamp associated with each event. In addition, for incidents which raw logs were available, we computed its *absolute preemption timeliness* (APT) using the absolute timestamp associated with each event. Note that longer preemption timeliness is better.

Visualizing detection and preemption timeliness. The Lamport detection timeliness and the Lamport preemption timeliness are presented by detection points in Figure 6.3. For example, the malicious user 15 was detected by AttackTagger when the malicious user progressed 24% of the total attack duration represented by the number of observed events (e.g., the malicious user was detected at the event number 24 of 100 observed events for that malicious user).

Certain insights can be drawn from timeliness measurements. In total, our approach detected 46 of 62 (74.2%) of the malicious users. Of the detected malicious users, 41 of 62 (66.1%) were detected before the attackers deliver their attack payloads. Note that we considered only 62 of 65 incidents when computing detection performance, since we excluded the three incident reports misclassified three benign users as malicious. 5 of 62 (8.1%) were detected at the last stage of the attacks. 12 of 46 identified malicious users were identified at the first observed event, at which they violated an obvious security policy (e.g., downloaded a known malware or logged in using an expired account).

Detection timeliness of an example incident.

In the incident 2010-05-13, the following sequence of events was observed (Table 6.2), as determined from the incident report. After infiltrating the target system, the attackers started delivering the payloads by connecting to a high-risk domain (milw0rm.com, which provides ready-to-use exploits), downloading a sensitive file (xploit.tgz), and then placing a backdoor that connected to an external IRC server (irc2.bad-domain.fi). Our approach identified the user as suspicious after repeated incorrect login attempts (event INCORRECT_PASSWORD, LOGIN and
HIGHRISK_DOMAIN). Most importantly, our factor graph based approach identified the user as malicious immediately when attack payloads began

Event	Description	UserState
INCORRECT PASSWORD (5 times)	A user supplies an incorrect credential at login. A repeated alerts indicates password guessing or bruteforcing.	benign
LOGIN	A user logs into the target system	*suspicious*
HIGHRISK DOMAIN	A user connects to a high-risk domain, such as one hosted using dynamic DNS (e.g., .dyndns, .noip) or a site providing ready-to-use exploits (e.g., milw0rm.com). The dynamic DNS domains can be registered free and are easy to setup. Attackers often use such domains to host malicious webpages.	*suspicious*
SENSITIVE URL	A user downloads a file with a sensitive extension (e.g., .c, .sh, or .exe). Such files may contain shell code or malicious executables.	***malicious***
CONNECT IRC	A user connects to an Internet Relay Chat server, which is often used to host botnet Control servers.	***malicious***
SUSPICIOUS URL	A user requests an URL containing known suspicious strings, e.g., leet-style strings such as expl0it or r00t, or popular PHP-based backdoor such as c99 or r57.	***malicious***

Table 6.2: Observed events during incident 2010-05-13

to be delivered (events SENSITIVE_URL, CONNECT_IRC, and SUSPICIOUS_URL).

For 5 incidents for which we did not detect the malicious user until the end of the attack. The main reason was a limited number of events generated by the monitoring system during these incidents. For example, in incident 2010-10-29, there were only two events observed: ANOMALOUS_LOGIN and DISABLE_BASH_LOGGING. A better monitoring infrastructure would improve the detection timeliness. For discussion of the 16 incidents for which we did not detect malicious users, refer to the False Negatives paragraph in the next section.

Measuring both LDT and LPT. To get a summary of detection timeliness for a set of incidents, we computed a new metric to measure both LDT and LPT called the *area under the Lamport timeliness curve* (AULTC). An AULTC value of 1 means that all malicious users were identified from the first observed event (in theory), which is ideal. Similarly, an AULTC value of 0 means that all malicious users were identified after the fact (in reality by the NCSA security team). Using a Lamport timeliness curve formed by connecting the detection points in Figure 6.3, we obtained the AULTC of 62.5%.

Compared to human detections which often happens after the system misuse (AULTC = 0), our model is relatively good at early detection.

Absolute Detection Timeliness. For the subset of 5 incidents in the Testing Set, we had the raw logs. For those incidents we computed the ADT values over the attack duration (in seconds): 1.97/1.97, 59.00/3,601.00, 1,787.00/1,787.00, 3,600.00/3,600.00, and 10,897.00/21,913.00. The best result is detection of malicious users at the very first minute (59th second) of an hour-long attack (3,601 seconds). In that case, the aggressive attacker caused a burst of security events and/or alerts. The attacker logged in using a stolen credential from a remote location, and then immediately collected system information (using the command *uname -a*), and downloaded privilege escalation exploits stored in .c files; that ultimately gave our model enough evidence to conclude that the user was malicious. Our detection timeliness is better than that of human detection, which only detects the attacks after the system misuse.

6.2.3 Detection performance

Detection performance was evaluated using standard performance metrics for machine learning based classifiers. The true positive rate (TP), i.e., the *detection rate* is the percentage of malicious users who are correctly identified as malicious. The false positive rate (FP) is the percentage of benign users who are incorrectly identified as malicious. The true negative rate (TN) is the percentage of benign users who are correctly identified as benign. The false negative rate (FN) is the percentage of malicious users who are incorrectly identified as benign.

True positives. AttackTagger detected 46 of 62 (74.2%) of the malicious users relatively early. Most of the attacks were detected before the attack payloads were launched. Our model detected attacks as early as within the first minute of observing events related to the attack.

False negatives. AttackTagger did not detect 16 out of 62 (25.8%) malicious users. The major reasons for mis-detection were: a lack of events (very few events generated), new event types (i.e., events that were not observed in the incidents included in the Construction Set), and generation of only one type of events.

Specifically, for seven of the false negatives, input to our model included

Incident	Activity
20100416	Illegal activities
20100513	Incorrect credentials (multiple times); Sending spam emails
20100513	Logging in from multiple IP addresses; Illegal activities
20101029	Logging in using expired passwords; Illegal activities
20101029	Illegal activities
20101029	Illegal activities

Table 6.3: Six hidden malicious users uncovered

only 1 to 2 events, which made it difficult even for security analysts to reach a conclusion. That suggests a need for comprehensive monitoring infrastructure across a system and network stacks (e.g., at the kernel or the hypervisor level) to capture the attacker behavior. For three of the false negatives, the malicious users performed one repeated activity (e.g., using an incorrect credential), which were seen as merely suspicious by AttackTagger. That phenomenon can be addressed by refining the factor functions. Similarly, for the remaining six false negatives, new event types were observed (e.g., misconfiguration of a web proxy, logging in using an incorrect version of SSH, or downloading of adult content) that had not been captured in our factor functions derived based on the Construction Set. The fix would be to update the factor funtions continously (this step needs human intervention) when system infrastructure changes or when a new event of interest is observed.

False positives. AttackTagger identified 19 of 1,253 benign users as malicious (1.52%) although these users were not recorded as malicious in the incident reports. We analyzed the false positives for incidents when raw logs of the incident were available and discussed our analysis with NCSA. Six of the 19 users were confirmed to have behaved maliciously and should be investigated further. Table 6.3 summarizes those users [1]. Although we misidentified the remaining 13 users, the remaining discovery of the six malicious users suggests that our method can uncover hidden attacks that have been missed by NCSA security analysts.

6.2.4 Performance comparison.

Using the Test Set, we compared our approach with other types of binary classifiers. A primitive type of classifier (baseline) is based on rules to detect attackers. More sophisticated classifiers are learning-based such as Decision Trees or Support Vector Machine.

[1]Examples of illegal activities are: download of a file with sensitive extensions (.c, .sh, .exe) or execution of anomalous commands (w, uname -a).

The main difference between our approach and the others is that our approach works with progressing attacks (i.e., using an incomplete sequence of events). The other binary classifiers often rely on a complete sequence of events to classify a user, so usually can be used only after attacks have reached their final stage.

In the following, we compare the detection performance of the selected techniques.

AttackTagger (AT), our approach, tags each observed event with a user state using Type-1, Type-2, and Type-3 factors.

Rule Classifier (RC) is a baseline rule-based classification model. We implemented it to identify attacks based on the most frequently observed alert in the Construction Set, namely a log in from an anomalous host.

Decision Tree (DT) is a rule-based classification model that groups decisions into a tree. It learns the rules from previous attacks. We used the $C45$ decision tree implementation in the scikit-learn machine learning library [11].

Support Vector Machine (SVM) is a frequently used classifier that uses a hyperplane and margins to classify classes. We used $classifier$=Support Vector Classification, with $kernel = linear$ using the scikit-learn implementation [11].

Implementation parameters. Parameters of the aformentioned techniques except AT were optimized based on the Construction Set. In our AT model, we constructed only the factor functions from the Construction Set and considered all the weights of factor functions to be equal. The training-free approach makes our approach less dependable on a training set, i.e., there is less overfit.

Performance analysis. We compared our detection performance and that of other techniques (Table 6.4).

The rule-based techniques (RC) performed poorly compared to AttackTagger. The Rule Classifier (RC) has a true positive rate of 9.8% since it identifies malicious users solely based on the most frequent alert in the Construction Set: a log in from an anomalous host. In the Testing Set, this alert was not observed in many of the incidents.

The other techniques (DT and SVM) seem to have an overfit problem, such that they only learn patterns of existing attacks in the Construction Set: the true negative is 100.0% for both, which means these models are conservative in classifying a user as malicious. As a result, they do not generalize well in

Name	*TP*	*TN*	*FP*	*FN*
AttackTagger	*74.2*	*98.5*	*1.5*	*25.8*
Rule Classifier	9.8	96.0	4.0	90.2
Decision Tree	21.0	100.00	0.00	79.0
Support Vector Machine	27.4	100.00	0.00	72.6

Table 6.4: Detection performance of the techniques

the Testing Set: their true positive are 21.0% and 27.4% respectively.

Comparing detection performance.

In this experiment, although AT had the best detection rate among the techniques (74.2% vs. 27.4% of the next-best techniques SVM). We performed a hypothesis test to show that the true positive rate for AttackTagger is significantly better than the true positive rate for the SVM approach. Our null hypothesis H_0 is that AT and SVM have an equal detection performance. The alternate hypothesis H_1 is that AT and SVM has a different detection performance. We test our hypothesis using the McNemar test, which is often used to compare effectiveness of drug treatments over test samples [12, 13].

We measured differences in detection of AT and SVM. For example, AT^+SVM^+ means that for a user, both AT and SVM determined that the user was malicious. Similarly, we measure the number of differences and agreements between the two techniques by four metrics: a = AT^+SVM^+, b = AT^+SVM^-, c = AT^-SVM^+, and d = AT^-SVM^-.

The McNemar test statistic is based on the number of discordant pairs (identified by b and c) between the two methods. The test statistic is computed by: $\chi^2 = (b+c)^2/(b-c)$. In our case, a=17, b=48, c=0, and d=1250, the test statistic is $\chi^2 = 48$. A p-value can be inferred according to the χ^2 value. The inferred p-value is < 0.00001 (i.e., the result is significant).

According to the test, we can safely reject the null hypothesis H_0. It means that the detection performance of AT is significantly different from the next-best (SVM) – in our case it is a better detection rate (74.2% vs. 27.4%).

6.2.5 Runtime performance.

A detection model must come up with a decision in a reasonable amount of time; otherwise, it misses the attack. AttackTagger was able to tag user states with events within seconds. Since we use Gibbs sampling for approximate inference instead of exact inference, the time it takes to infer the user states depends on sampling iterations and is linear to the length of the event

sequence. On average, it took AttackTagger 530 ms to tag an event with following statistics: minimum tagging time is 328ms, maximum tagging time is 644ms, with standard deviation is 0.1 for 65,389 events. The number of events can be limited by a fixed time-window or by importance sampling of interesting events.

CHAPTER 7

RELATED WORK

Intrusion detection systems have been investigated ever since Anderson report was published over thirty years ago [14]. Most of the works focused on signature-based or anomaly-based techniques. Signature-based techniques often identify only a stage of an attack using known patterns [15, 16]. Anomaly-based methods use profiles, statistical measurements, or distance measurements to capture abnormal behaviors of potential novel attacks at the cost of overwhelming false alarms [17].

As IDSes have been wildly deployed, new challenges arise because of dynamic infrastructure (e.g., variety of constantly changing hosts and network devices) [18]. IDS alerts are generated from monitoring data across system stacks and network interfaces, e.g., network packet captures, authentication logs (SSH or Kerberos), access logs (HTTP requests). Such diverse and overwhelming number of alerts challenge automated systems to correlate alerts (i.e., normalization, aggregation, correlation, and analysis of alerts) with an attack and to identify users involved in the attack [19]. Given the correlated alert, security analysts still have to spend a significant time investigating false or insignificant alerts [18].

Probabilistic graphical models have been employed to model uncertainty in multi-staged attacks. In attack scenario modeling, Bayesian Networks can model causal relations among high-level attack stages [20]. A BN and its parameters can be derived based on domain knowledge of the target system and known attacks. The network allows inference on a potential attack stage. The main challenge of BN is the assumption of the model structure and its parameters, which have a high uncertainty in constantly changing infrastructure. In attack sequence modeling, Markov models such as Markov Random Fields define an attack as a sequence of actions that causes a transition in the underlying system state [21, 22]. Previous sequence modeling techniques (such as variable length markov model or matrix-based recommendation sys-

tem) built models based on observed events [23]. Those techniques do not integrate external knowledge of users or the target system (e.g., the user profile in our model) to improve accuracy of inference.

To address the limitations of previous works, we use Factor Graph, a type of probabilistic graphical model that unifies both BN and MRF[3]. Unlike signature and anomaly techniques, Factor Graphs do not rely on a single rule or an anomaly measure. Instead, using factor functions, a factor graph collectively identifies attacks using both rules, anomaly measures, and sequential measures among observed events. Type-1 factors represent rules and Type-2 factors represent sequential dependency among events and user states. Moreover, Type-3 factors can incorporate historical data (such as attributes of a user) and expert knowledge into our model.

Note that our technique does not mean to replace existing IDSes, instead, our technique operates on top of monitoring data provided IDSes and system/network monitors. By combining strengths of individual techniques, AttackTagger can identify progressing attacks using only a partially observed events leading to the attacks.

CHAPTER 8

CONCLUSION

In this paper, we evaluated effectiveness of using factor graph to detect progressing attacks at early stages. Incident data of 116 real-world security incidents were used in our evaluation. Our approach i) detected 74% of the attacks as early as minutes to hours before the system misuse (compared to human detection which was after the fact of the attacks) and ii) uncovered six hidden malicious users from 65 incidents in our Testing Set. In the future, we plan to investigate effectiveness of individual or groups of factor functions in our detection performance.

REFERENCES

[1] A. Shulman, "The underground credentials market," *Computer Fraud & Security*, vol. 2010, no. 3, pp. 5–8, 2010.

[2] A. Sharma, Z. Kalbarczyk, J. Barlow, and R. Iyer, "Analysis of security data from a large computing organization," in *Dependable Systems & Networks (DSN)*. IEEE, 2011.

[3] B. J. Frey, F. R. Kschischang, H.-A. Loeliger, and N. Wiberg, "Factor graphs and algorithms," in *Proceedings of the Annual Allerton Conference on Communication Control and Computing*, 1997, pp. 666–680.

[4] D. Nikovski, "Constructing bayesian networks for medical diagnosis from incomplete and partially correct statistics," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 12, no. 4, pp. 509–516, 2000.

[5] J. Lafferty, A. McCallum, and F. C. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," 2001.

[6] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[7] C. Zhang and C. Ré, "Towards high-throughput gibbs sampling at scale: A study across storage managers," in *International Conference on Management of Data*. ACM, 2013.

[8] C. K. Carter and R. Kohn, "On gibbs sampling for state space models," *Biometrika*, vol. 81, no. 3, pp. 541–553, 1994.

[9] B. e. a. Andres, "An empirical comparison of inference algorithms for graphical models with higher order factors using opengm," in *Pattern Recognition*. Springer, 2010, pp. 353–362.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, 1978.

[11] F. e. a. Pedregosa, "Scikit-learn: Machine learning in python," *JMLR*, vol. 12, 2011.

[12] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.

[13] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[14] J. P. Anderson, "Computer security threat monitoring and surveillance," Tech. Rep., 1980.

[15] Bro, "Bro intrusion detection system," www.bro-ids.org.

[16] B. Caswell, J. Beale, and A. Baker, "Snort ids and ips toolkit," *America: Syngress*, 2007.

[17] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, no. 2, pp. 222–232, 1987.

[18] A. Pecchia, A. Sharma, Z. Kalbarczyk, D. Cotroneo, and R. K. Iyer, "Identifying compromised users in shared computing infrastructures: a data-driven bayesian network approach," in *Proc. of Reliable Distributed Systems (SRDS)*. IEEE, 2011.

[19] R. Sadoddin and A. Ghorbani, "Alert correlation survey: framework and techniques," in *Proc. of Intl. Conference on Privacy, Security and Trust*. ACM, 2006.

[20] X. Qin and W. Lee, "Attack plan recognition and prediction using causal networks," in *Computer Security Applications Conference, 2004. 20th Annual*. IEEE, 2004, pp. 370–379.

[21] J. Hu, X. Yu, D. Qiu, and H.-H. Chen, "A simple and efficient hidden markov model scheme for host-based anomaly intrusion detection," *Network, IEEE*, vol. 23, no. 1, pp. 42–47, January 2009.

[22] K. K. Gupta, B. Nath, and R. Kotagiri, "Layered approach using conditional random fields for intrusion detection," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 1, pp. 35–49, 2010.

[23] D. S. e. a. Fava, "Projecting cyberattacks through variable-length markov models," *Information Forensics and Security, IEEE Trans. on*, 2008.