Preemptive Intrusion Detection: Theoretical Framework and Real-World Measurements

Phuong Cao, Eric Badger,
Zbigniew Kalbarczyk, Ravishankar Iyer
Coordinated Science Laboratory
University of Illinois at Urbana Champaign
{pcao3,badger1,kalbarcz,iyer}@illinois.edu

Adam Slagell
National Center for Supercomputing Applications
University of Illinois at Urbana Champaign
{slagell}@illinois.edu

ABSTRACT

This paper presents a Factor Graph based framework called AttackTagger for highly accurate and preemptive detection of attacks, i.e., before the system misuse. We use security logs on real incidents that occurred over a six-year period at the National Center for Supercomputing Applications (NCSA) to evaluate AttackTagger. Our data consist of security incidents that led to compromise of the target system, i.e., the attacks in the incidents were only identified after the fact by security analysts. AttackTagger detected 74 percent of attacks, and the majority them were detected before the system misuse. Finally, AttackTagger uncovered six hidden attacks that were not detected by intrusion detection systems during the incidents or by security analysts in post-incident forensic analysis.

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 credentials can be bought on black markets at low cost [20]. Using stolen credentials, attackers impersonate as legitimate users, effectively bypassing traditional defenses, e.g., network firewalls. Such attacks are often discovered only in their final stages when attack payloads are delivered, e.g., authentication services are contaminated to harvest more credentials or computing infrastructure are utilized to build botnets [18].

Detecting such 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 is 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 physical location of the user login. The user activities remain to be seen on the target system. Examining an individual user activity is not a sufficient basis for drawing an accurate conclusion about the user's intention. Logging in from a remote location can indicate either a *legitimate* user is logging in from outside of the regular infrastructure, or an *illegitimate* user is logging in using stolen credentials. A framework is considered to reason about the user's activities collectively.

We propose the AttackTagger framework, which is built upon *Factor Graph*, a type of probabilistic graphical model consisting of random variables and factor functions [8]. 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 $imply(A, B)$ means B is often followed by A. For example, in the context of masquerade attacks, an attacker impersonates a legitimate user, e.g., by logging into 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 necessarily 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 identify a specific signature of an attack, our AttackTagger framework uses factor functions to reason about stages of an attack collectively. The factor function *imply*(B,C,D) can use the existing result of the previous factor *imply*(A,B) to determine that a user is malicious. 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 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). 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's 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 malicious users relatively early (from minutes to hours before the system is misused). 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.

HotSoS '15 Urbana, IL USA

http://dx.doi.org/10.1145/2746194.2746199 ...$15.00.

The main contributions are:

- A novel application of Factor Graphs that integrates user behaviors from a heterogeneous data sources for preemptive intrusion detection, i.e., before the system misuse.
- Design, implementation, and experimental evaluation 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.

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.

A credential-stealing incident (2008). In May 2008, a sophisticated credential-stealing incident occurred at NCSA. Using a compromised user account credential (e.g., a pair 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. NCSA computing infrastructures were shared among hundreds of users, and many of them logged in to NCSA using the compromised gateway node. Thus, the attackers were able to collect new credentials of subsequent user logins.

An excerpt from the raw logs of the incident is listed in Table 1. First, the attackers used the compromised credential to log into the gateway node from a remote host, i.e., a host located outside of NCSA's 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). A sophisticated attacker can change the file extension to a harmless one (e.g., .jpg). But our netflow monitor can identify a mismatch between a file extension and its content by analyzing the file header (e.g., a Windows executable file always begins with the *MZP* string because of its Portable Executable file format specification). The attackers then compiled, and escalated privilege to the root by exploiting a kernel bug (CVE-2008-0600) on the compromised node . Those 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 logs. To harvest credentials of users logging into the compromised node, after the attackers escalated to *root*, the attackers injected credential-collecting code into the original SSHd, forcing it to restart (which resulted in the *SIGHUP* signal in the event e^2). Each raw log entry was automatically mapped to an event identifier using regular expression scripts.

In this incident, the attackers were identified after the fact by security analysts. The collateral effect of the incident was: leaking credentials of subsequent users who logged into the compromised node and potential usage of the leaked credentials for subsequent attacks.

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 of credentials, remote login, download and execution of privilege escalation exploits, installation of backdoors, and periods of dormancy. On the other hand, single-staged attacks (which typically are 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).

[1]a widely deployed authentication service of UNIX systems.

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

Table 1: Example raw logs and events of an incident

Challenges of detecting multi-staged attacks. Detecting a multi-staged attack requires identification of the states of the involved users throughout the attack. A user state can be *benign* (when a legitimate user logs in from the remote location as a part of his/her normal activity), *suspicious* (when an illegitimate user uses stolen credentials to log in from the remote location), or *malicious* (when a user violates a security policy). Each observed user event can be tagged with a *user state*.

In the above example, the single *remote login* event provides insufficient information to tag the corresponding user state as *malicious*. By itself, that event does not indicate a security violation, although other single events could do so, such as modification of a system service by someone who is not 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 how to tag the event, we need further information. For example, the existing context of the system, the user profile, and we need information from subsequent events. Therefore, 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 use rules or signatures to identify malicious users. In our example, given the event e^2 (*restart system service* in Table 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 is upgrading 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 knowledge of 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 collectively.

3. PROBABILISTIC GRAPHICAL MODELS

In this section, we provide an overview of *Probabilistic graphical models* (PGMs), graph-based representations of dependencies among random variables, in modeling security incidents. 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 [8]. While BNs and MRFs have been successfully employed in a variety of do-

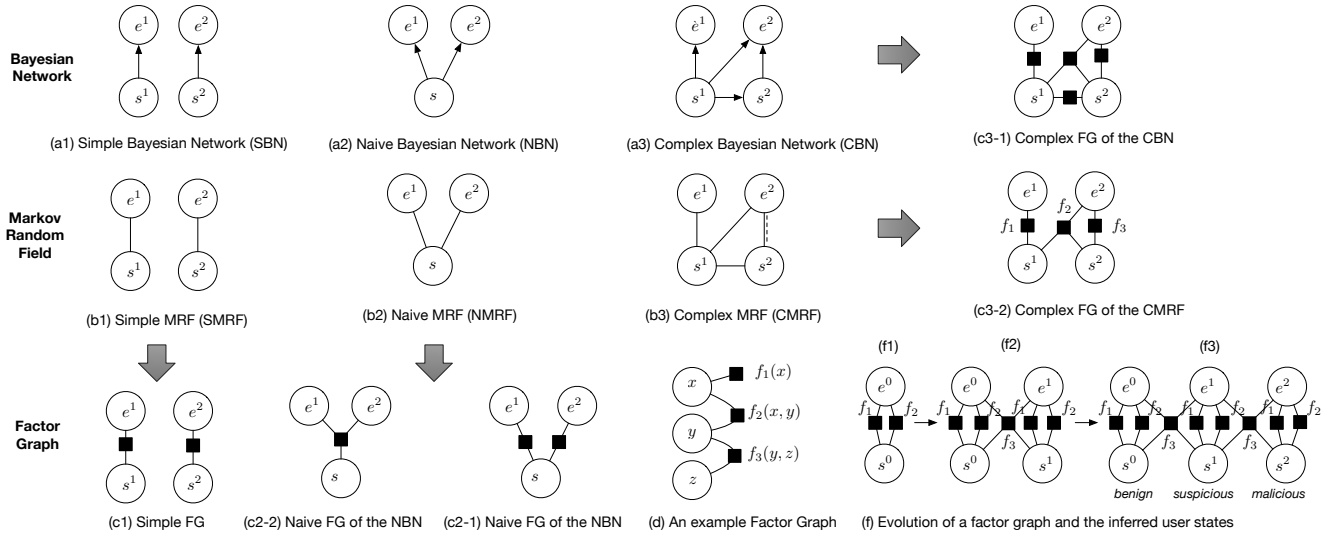


Figure 1: Illustrations of use of Bayesian Network, Markov Random Field, and Factor Graph to model security incidents.

mains, such as medical condition diagnosis or entity extraction from text [12, 10], the use of FGs in security domains has not been explored. We found that FG is more suitable than the others for modeling security incidents, since FG can subsume both BN and MRF [8].

When PGMs are used 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 1), we consider the sequence of the observed events $E = \{e^1 = \textit{download sensitive}, e^2 = \textit{restart system service}\}$, 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.

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

A simple Bayesian Network (SBN) models the dependencies of the observed events E and the user states S in Figure 1-a1. This model assumes that the observed events E are independent and that 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 user states. An example of such dependencies is, an event e^i that is not only caused by its corresponding user state s^i, but also caused by a previous user state s^{i-1}.

In Figure 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 causal 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 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 the arrival of each new observed event (i.e., based on an incomplete set of the observed events).

In Figure 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 causal 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. Although 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. That non-causal relation is discussed in more detail in the part-of-speech tagging example in the MRFs sub-section.

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

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

The simple Markov Random Field (SMRF) depicted in Figure 1-b1 is equivalent to the simple Bayesian Network (SBN) in Figure 1-a1. In the SMRF, the dependency between e^1 and s^1 is represented by a function $\phi(e^1, s^1)$. The function $\phi(e^1, s^1)$ is defined as a conditional probability function $p(e^1|s^1)$.

Characteristics of an MRF are as follows. Let $n(v)$ be the set of v's neighbors, i.e., the vertices that are directly connected to v by a single edge. Variables in a MRF are grouped into cliques, in which all variables within each clique must be pairwise connected. A clique is a maximal clique if it cannot be extended by addition of an adjacent variable.

A complex joint probability function (pdf) 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 its variables. The factorization simplifies representation and computation of MRFs.

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 say X causes Y [10]. For example, in part of speech (POS) tagging, a word (an observed variable) is often tagged with a part of speech (a hidden variable), e.g., noun or 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 different parts 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, an MRF can be used as an alternative to a BN. Figure 1-b1 and Figure 1-b2 depict an MRF model's equivalent to the BN models in Figures 1-a1,a2, where the directed edges in the BNs have been replaced with the undirected edges. An MRF does not make any assumptions on the causal relation among the variables. An arbitrary function can be used to define the relation among the variables.

In our example, an event (an observed variable) and a user state (a hidden variable) can have a non-causal relation. For example, when a user logs in remotely, it is usually that the user is traveling (i.e., the user state is benign), not because an attacker is impersonating the user with a stolen user credential (i.e., the user state is malicious).

An MRF model (Figure 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 the 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 either the local functions of an event and the corresponding user state (e.g., $\phi(e_2, s_2)$), or the local functions of 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 1-b3) simply uses one local function to describe the relation among the three variables in the clique, instead of using three local functions (i.e., conditional probability distribution function) to describe the three pair-wise causal dependencies in the equivalent BN model (Figure 1-a3). Despite the 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.

The advantages and disadvantages of using MRFs are as follows. In MRFs, the use of one local function per clique avoids the need to make explicit assumptions about causal dependencies among variables, as necessary 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 that cannot be naturally expressed using MRFs, because a MRF is built upon maximal cliques.

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

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 Figure 1-d.

Factor Graphs are simpler and more expressive than BNs and MRFs. In a Factor Graph, 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 [8]. 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 have to be defined. Equivalent FG representations of BNs and MRFs are shown in Figures 1-{c1, c2-1, c2-2, c3-1, and c3-2}. A detailed discussion of conversions among FGs, BNs, and MRFs can be found in [8]. FGs has led to development of effective inference algorithms (e.g., Gibbs sampling or belief propagation) [16, 8]. Since FGs offer the same representation for both BNs and MRFs, those algorithms can be used for existing BNs and MRFs when they are converted to FGs.

In our security domain, Factor Graphs are more flexible to define different types of relations among the events to the user state compared to BNs and MRFs. Specifically, FGs can integrate sequential relation among events and external knowledge (e.g., expert knowledge or knowledge of a user profile) to their models.

4. FRAMEWORK OVERVIEW

In this section, we provide an overview of using Factor Graphs in our framework to model the example incident described Section 2. We briefly overview steps of our framework in Figure 2.

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 $E = \{e^0 = \textit{login remotely}, e^1 = \textit{download sensitive}, e^2 = \textit{restart system service}\}$. 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 takes random variables, e.g., observed user events or hidden user states as the input, and outputs a discrete value indicating relations among the inputs.

For example, a *Type-1* factor function $f(e, s)$ can be de-

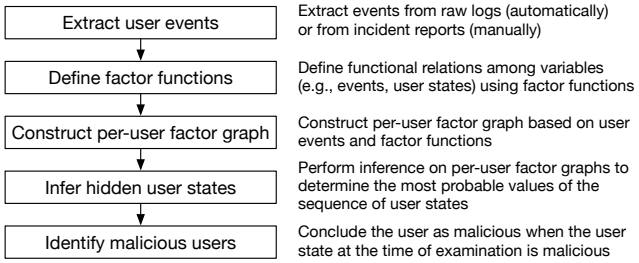


Figure 2: Process of modeling using Factor Graphs.

fined 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*; it returns 0 otherwise. For example, the function f_1 in Figure 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 an SSHd, which is likely a maintenance activity. The function f_2 in the Figure 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 Figure 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 and user profiles, and to integrate expert knowledge into Factor Graphs. A more formal definition and discussion of Type-1 and Type-2 factors are provided in Section 5.

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

Figure 1-f illustrates the *evolution* of a per-user Factor Graph as new events are observed. When only one event is observed, the Factor Graph contains only 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 1-f), a possible sequence of user states S is automatically evaluated through a summation of 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 values of the user states in the constructed Factor Graph and select the sequence of values that results in a highest score. The most probable sequence of values is S={*benign, suspicious, malicious*}, as shown in Figure 1-f. In our model, we compute the probabilities of the user state sequences using more efficient methods (Section 5).

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

Most steps in our framework are automated, except Step-2 (defining of factor functions), which requires expert knowledge. Using our framework, security analysts can quickly examine 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 respond to security incidents or to increase additional monitoring of suspicious users to uncover potentially unauthorized activities.

5. ATTACKTAGGER MODEL

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

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 3. 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). The set of events $\mathcal{E}$ (Table 3) is system-specific and is predefined based on: the capabilities of the monitoring tools (e.g., IDS alerts) and expert knowledge of the target system.

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* $s^t \in \mathcal{S}$ = {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 (e.g., 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 more than one security events 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 that the user has malicious intentions. More fine-grained user states can be defined.

The notation and the meaning of the variables of an attack in the model are given in Table 2.

Characterization of factor functions. A factor func-

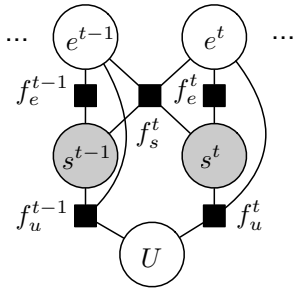


Figure 3: A snapshot of the Factor Graph model of an attack at a time t.

tion 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 between 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 categorized 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 of 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 are the same as those of A, element-wise.

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

For example, in Section 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*. The 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, $s^{t-1} = suspicious$, $s^t = malicious$). 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.

Higher-order and complex factor functions relating multiple events can be defined, however, they construct more complex Factor Graphs.

A generic Factor Graph. Figure 3 shows a generic Factor Graph model of an attack. Variable nodes correspond to either observed variables U, E^t or hidden variables (S^t). Factor nodes represent factor functions describing functional relations among the observed variables and hidden variables.

For the purpose of illustration, Figure 3 shows a part of the complete Factor Graph at the time t. Five factor functions are illustrated: f_e^{t-1}, f_e^t (Type-1), f_s^t (Type-2), and f_u^{t-1}, f_u^t (Type-3). In our model, the factor functions are defined for the sequence of events E^t from e^1 (when a user begins using the system) to e^t (at an observation time t).

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 2: Notations of variables used in our model.

Inference of hidden user states. To identify malicious users, AttackTagger infers the most probable values of the user state in the sequence S^t using the constructed factor graph. Specifically, if the user state $s^t = suspicious$, then the user is allowed to operate in the target system under close monitoring (e.g., logging network traffic of the user or logging user commands); if the user state $s^t = malicious$, the user is identified as an attacker and actions are taken to disconnect the user from the target system (e.g., terminating the user's active network connections or disabling the user account). Our inference is based on the joint probability distribution on the Factor Graph.

Joint probability distribution function (pdf). Let $F = \{F_e, F_s, F_u\}$ be the set of factor functions of Type-1, Type-2, and Type-3, respectively. Let $f(c_f)$ be a factor function $f \in F$ where c_f is the set of its inputs that can be observed and hidden state variables. The joint probability distribution $P(U, E^t, S^t)$ of the observed variables and hidden state variables can be factorized using factor functions F: $P(U, E^t, S^t) = \frac{1}{Z} \prod_{f \in F} f(c_f)$. In the joint pdf, we use Z as the normalization factor to make sure that the joint pdf is a proper distribution, instead of computing the *score* of S^t as seen in Section 4. The normalization factor Z can be computed by summing values of f over all possible combinations of the variables $\{U, E^t, S^t\}$.

Inference. The most probable values of the user states in the sequence S^t can be inferred by enumerating all possible values of the user states in the sequence and returning the values that maximize $P(U, E^t, S^t)$.

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

Although the brute-force approach can give an exact result for the most probable hidden state variables S^t, its naive enumeration of all possible values of the user states in the sequence is costly. Since each state variable s^t 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 for inference [16].

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 on Factor Graphs [5]. 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 for using 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

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 3: Examples of user attributes and events.

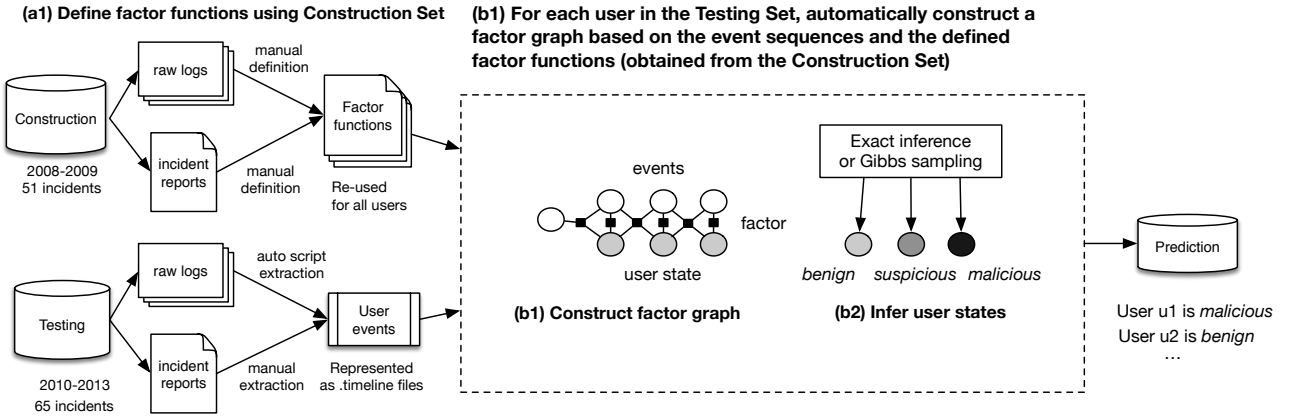


Figure 4: Experiment flow with input is incident report or raw logs, and output is prediction of malicious users.

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 and so forth, until it reaches the last user state s^n. That concludes the sampling process for a user state sequence at 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 converges (i.e., the user state sequence does not change from iteration k to iteration $k+1$).

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 Threat model

We consider networked computers in an enterprise environment (NCSA) where adversaries come from outside the enterprise perimeter. We assume that the monitoring infrastructure at NCSA was implemented to capture events leading to attacks [18].

6.2 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 with system services (e.g., SSHd) to steal credentials, misuse of computing infrastructure (to build botnets, send spam emails, or launch denial of service attacks), or remote exploitation of Virtual Network Computing servers to get a system shell.

Incident data. The incident data include incident reports and raw logs. For each incident in our dataset, 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 spanned a duration of 24 to 72 hours. Thus, for a subset of security incidents we also gathered the *raw logs* for a period of 24 to 72 hours before and after the NCSA security team detected a given incident. That duration of time is sufficiently long to cover most of the traces of attacks in our dataset. Since the data retention policy changed during 6 years when incident data were being collected, the raw logs were only available for a subset of incidents (Table 4). The raw logs are valuable because they captured activities of both benign and malicious users during the incidents.

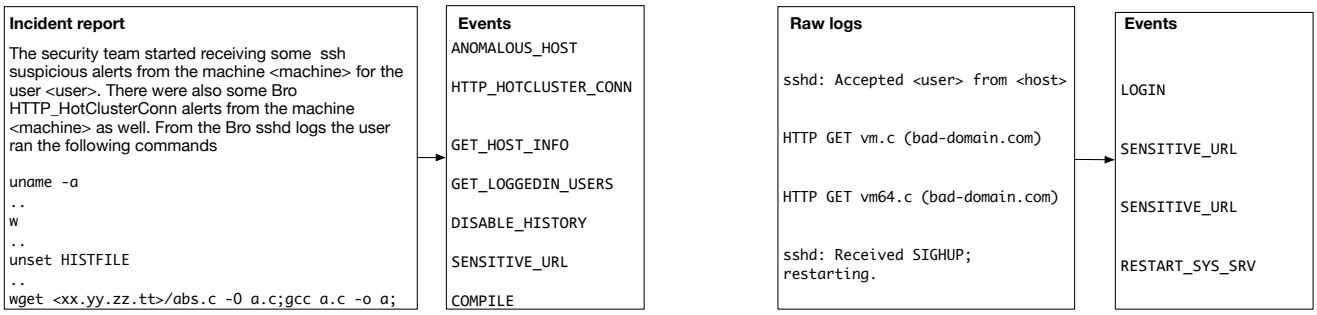
Construction Set and Testing Set. The data on 116 incidents have been partitioned into two disjoint sets: (i) a Construction Set of 51 incidents collected during the 2008-2009 period, and (ii) a Testing Set of 65 incidents collected during the 2010-2013 period. We used the incident data from the Construction Set to extract the set of events observed during the incidents and to define the factor functions. We use the Testing Set incident data to construct the Factor Graph for each user and to evaluate the detection capabilities of the constructed Factor Graphs.

The partition was based on the following. In the 2008-2009 period, a subset of the incidents were credential stealing incidents. Our conjecture is that, in many incidents observed during the 2010-2013 period, the attackers used the stolen credentials, exploited weak user passwords, or used similar attack patterns (e.g., remote login, download sensitive file, and escalate privilege) to infiltrate the NCSA infrastructure. As a result, our model has been constructed using the Construction Set and evaluated using the Testing Set. Table 4 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. Based on post-incident analysis, 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

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

Table 4: Summary of the incident dataset



a1) Manual conversion from incident report to events

a2) Automatic conversion from raw logs to events

Figure 5: Manual and automatic conversion of incident reports and raw logs to events.

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 user u observed in separated incidents is considered as separated users. 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 6.5).

6.3 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 in an incident.

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 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). 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 5-a1.

When the raw logs corresponding to a user session were available, regular expressions scripts were used to convert them to a sequence of events. Each log entry in the raw logs was mapped to a unique event identifier and event metadata, including epoch timestamp and the user identifier who triggered the event. Order of events happenned in the raw logs are inferred from the timestamps. Examples of a log entry and the corresponding event are illustrated in Figure 5-a2. 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 manually 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). 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 downloads a file with a sensitive extension* (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 because of the flexibility of Factor Graphs. 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, practitioners can construct their own factor functions based on their events and expert knowledge of their target systems. 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 space limitation, a complete list of factor functions is available online [2].

6.4 Construction and inference on Factor Graph

Given the defined factor functions, we constructed a Factor Graph for each user session (*per-user Factor Graph*) and performed inference on the constructed Factor Graphs.

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 all of the 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 shows the experimental flow, including the process of constructing a Factor Graph for each user.

[2]http://bit.ly/preemptive-intrusion-detection

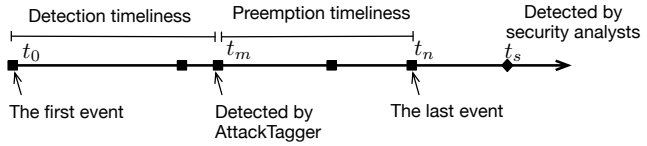


Figure 6: An attack timeline: the first event is observed at t_0; AttackTagger detects the attack at t_m; the attack finishes at t_n; security analysts detect the attack at t_s. Each square dot is an event related to the attack.

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 6.5).

Inference of user states on Factor Graphs. In Figure 5-b2, given a constructed Factor Graph of a user session, the user state sequence was approximated using Gibbs sampling [5] in the OpenGM library [2]. 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.

6.5 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 had not been discovered by NCSA security analysts. In this section, we describe how we analyzed the detection timeliness and detection accuracy of our model using the Testing Set.

6.5.1 Timestamps and ordering of events

We used *Lamport timestamp* (or logical clock) to establish the relative order of events [11]. 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 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 were observed, the events were assigned increasing values of the Lamport timestamp, such that if an event a happened before b, then $\mathrm{C}(a) < \mathrm{C}(b)$. For incidents for which raw logs were available, each event was assigned an *absolute timestamp* in addition to its Lamport timestamp.

Figure 6 illustrates an event timeline of a malicious user. In the following, we refer to a timestamp as either a Lamport timestamp or an 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 attack and indirectly corresponds to an absolute attack duration (expressed in seconds, minutes, or hours). To measure the absolute attack duration, we need an absolute timestamp to be associated with each event.

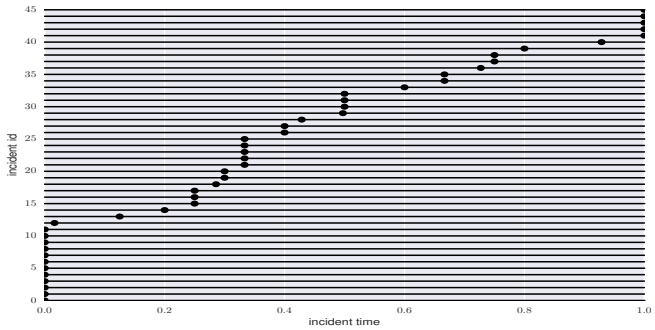


Figure 7: The x axis is the Lamport attack duration (incident time) of the malicious users normalized to the range [0-1]. Each row (incident id) in the y axis is a malicious user detected by AttackTagger in an incident. The dot in a row represents the time when the malicious user was detected by AttackTagger.

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., to detect a progressing attack as early as possible.

6.5.2 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 $t_d = t_m - t_0$. 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 for which raw logs were available, we computed their *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. 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 $t_p = t_a - t_m$. 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 for which raw logs were available, we computed their *absolute preemption timeliness* (APT) using the absolute timestamp associated with each event. Longer preemption timeliness is better.

Detection and preemption timeliness. The Lamport detection timeliness and the Lamport preemption timeliness are presented by detection points in Figure 7. For example, malicious user 15 was detected by AttackTagger when the malicious user had progressed 24% of the total attack duration represented by the number of observed events.

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

Event	Description	UserState
INCORRECT PASSWORD (5 times)	A user supplies an incorrect credential at login. Repeated alerts indicate 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 set up. Attackers often use such domains to host malicious webpages.	*suspicious*
SENSITIVE URL	A user downloads a file with a sensitive extension . Such files may contain shell code or malicious executables.	***malicious***
CONNECT IRC	A user connects to an Internet Relay Chat server. IRC are often used to host botnet Control servers.	***malicious***
SUSPICIOUS URL	A user requests a URL containing known suspicious strings, e.g., leet-style strings such as expl0it or r00t, or popular PHP-based backdoors such as c99 or r57.	***malicious***

Table 5: Observed events during incident 2010-05-13.

Detection timeliness of an example incident.
In incident 2010-05-13, the following sequence of events was observed (Table 5), 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 to be delivered (events SENSITIVE_URL, CONNECT_IRC, and SUSPICIOUS_URL).

For the 5 incidents for which we did not detect the malicious user until the end of the attacks, the main reason was a limited number of events generated by the monitoring system during these incidents. For example, in incident 2010-10-29, only two events were observed: ANOMALOUS_LOGIN and DISABLE_BASH_LOGGING. A better monitoring infrastructure would improve the detection timeliness. For a discussion of the 16 incidents for which we did not detect the 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 used 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. 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 7, we obtained an AULTC of 62.5% normalized for 46 detection points. Compared to human detections, which often happen after system misuse (AULTC = 0), our model is relatively good at early detection.

Absolute Detection Timeliness. For a 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 was detection of a malicious user 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 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 attacks after system misuse.

6.5.3 Detection performance

Detection performance was evaluated using standard performance metrics for machine learning 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%) 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 misdetection were: a lack of events (very few events were observed), 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 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 attacker behavior. For three of the false negatives, the malicious users performed one activity repeatedly (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

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: Six hidden malicious users uncovered.

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 is to update the factor functions continuously (which requires 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 those 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 summarizes those users [3]. Although we misidentified the remaining 13 users, the discovery of the six malicious users suggests that our method can uncover hidden attacks that have been missed by NCSA security analysts.

6.5.4 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 Machines.

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 Trees (DT) are rule-based classification model that groups decisions into a tree. They learn the rules from previous attacks. We used the $C45$ decision tree implementation in the scikit-learn machine learning library [14].

Support Vector Machines (SVM) are 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 [14].

Implementation parameters. Parameters of the aforementioned 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.

[3]Examples of illegal activities include download of a file with sensitive extensions or execution of anomalous commands (w, uname -a).

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 7: Detection performance of the techniques

Performance analysis. We compared our detection performance and that of other techniques (Table 7).

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 based solely on the most frequent alert in the Construction Set: a log in from an anomalous host. In the Testing Set, that 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 the Testing Set; their true positive are 21.0% and 27.4% respectively.

Comparing detection performance. In this experiment, AT had the best detection rate among the techniques (74.2% vs. 27.4% for the next-best technique 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 the same detection performance. The alternative hypothesis H_1 is that AT and SVM have significant different detection performance. We tested our hypothesis using the McNemar test, a popular drug treatments statistical test [19].

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 measured 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=1,250; 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 that of the next-best (SVM); in our case it has a better detection rate (74.2% vs. 27.4%).

6.5.5 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. The minimum tagging time was 328ms, the maximum tagging time was 644ms, and the standard deviation was 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.

7. RELATED WORK

Intrusion detection systems have been investigated ever since the Anderson report was published over thirty years ago [1]. Most of the work has focused on signature-based or anomaly-based techniques. Signature-based techniques often identify only a stage of an attack that uses known patterns [3]. Anomaly-based methods use profiles, statistical measurements, or distance measurements to capture abnormal behaviors of potential novel attacks at the cost of overwhelming number of false alarms [6].

As IDSes have been widely deployed, dynamic infrastructure (e.g., a variety of constantly changing hosts and network devices) presents new challenges [13]. IDS alerts are generated from monitoring across system stacks and network interfaces, e.g., network packet captures, authentication logs (SSH or Kerberos), and access logs (HTTP requests). Such diverse and numerous 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 [17]. Given the correlated alerts, security analysts still have to spend a significant amount of time investigating false or insignificant alerts [13].

Probabilistic graphical models have been employed to model uncertainty in multi-staged attacks. In attack scenario modeling, BNs can model causal relations among high-level attack stages [15]. 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 high uncertainty in a constantly changing infrastructure. In attack sequence modeling, Markov models such as MRFs define an attack as a sequence of actions that causes a transition in the underlying system state [9]. Previous sequence modeling techniques (such as variable length markov models or matrix-based recommendation systems) built models based on observed events [7]. 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 efforts, we use Factor Graphs, a type of probabilistic graphical model that unifies BNs and MRFs [8, 4]. 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 rules, anomaly measures, and sequential measures among observed events. In our model, Type-1 factors represent rules and Type-2 factors represent sequential dependency among events and user states. Moreover, Type-3 factors can incorporate user profile and expert knowledge into our model.

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

8. CONCLUSION

In this paper, we evaluated the effectiveness of using Factor Graphs to detect progressing attacks at early stages. Incident data for 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 (whereas human detection always occurred after misuse) and ii) uncovered six hidden malicious users from 65 incidents in our Testing Set. In the future, we plan to investigate the effectiveness of individual or groups of factor functions in our detection performance.

Acknowledgements

We would like to acknowledge the NCSA security team for providing incident data and ground truth; DEPEND group members, Dr. Charles Kamhoua, Dr. Shuo Chen, and anonymous reviewers for providing valuable feedbacks; and Ms. Jenny Applequist for proofreading. 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.

9. REFERENCES

[1] Anderson, J. P. Computer security threat monitoring and surveillance. Tech. rep., 1980.

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

[3] Bro. Bro intrusion detection system. www.bro-ids.org.

[4] Cao, P., Chung, K.-w., Kalbarczyk, Z., Iyer, R., and Slagell, A. J. Preemptive intrusion detection. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security* (2014), ACM, p. 21.

[5] Carter, C. K., and Kohn, R. On gibbs sampling for state space models. *Biometrika 81*, 3 (1994), 541–553.

[6] Denning, D. E. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 2 (1987), 222–232.

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

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

[9] Hu, J., Yu, X., Qiu, D., and Chen, H.-H. A simple and efficient hidden markov model scheme for host-based anomaly intrusion detection. *Network, IEEE 23*, 1 (January 2009), 42–47.

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

[11] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21* (1978).

[12] Nikovski, D. Constructing bayesian networks for medical diagnosis from incomplete and partially correct statistics. *Knowledge and Data Engineering, IEEE Transactions on 12*, 4 (2000), 509–516.

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

[14] Pedregosa, F. e. a. Scikit-learn: Machine learning in python. *JMLR 12* (2011).

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

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

[17] Sadoddin, R., and Ghorbani, A. Alert correlation survey: framework and techniques. In *Proc. of Intl. Conference on Privacy, Security and Trust* (2006), ACM.

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

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

[20] Shulman, A. The underground credentials market. *Computer Fraud & Security 2010*, 3 (2010), 5–8.