Efficient Discovery of Abnormal Event Sequences in Enterprise Security Systems

Boxiang Dong[*,1,2], Zhengzhang Chen[*,1], Hui (Wendy) Wang[3], Lu-An Tang[1], Kai Zhang[4], Ying Lin[5], Wei Cheng[1], Zhichun Li[1], Haifeng Chen[1]

[1]NEC Laboratories America

[2]Montclair State University

[3]Stevens Institute of Technology

[4]Temple University

[5]University of Washington

[*]Corresponding authors: dongb@montclair.edu; zchen@nec-labs.com

ABSTRACT

Intrusion detection system (IDS) is an important part of enterprise security system architecture. In particular, anomaly-based IDS has been widely applied to detect single abnormal process events that deviate from the majority. However, intrusion activity usually consists of a series of low-level heterogeneous events. The gap between low-level process events and high-level intrusion activities makes it particularly challenging to identify process events that are truly involved in a real malicious activity, and especially considering the massive "noisy" events filling the event sequences. Hence, the existing work that focus on detecting single events can hardly achieve high detection accuracy. In this work, we formulate a novel problem in intrusion detection — suspicious event sequence discovery, and propose GID, an efficient graph-based intrusion detection technique that can identify abnormal event sequences from massive heterogeneous process traces with high accuracy. We fully implement GID and deploy it into a real-world enterprise security system, and it greatly helps detect the advanced threats and optimize the incident response. Executing GID on both static and streaming data shows that GID is efficient (processes about 2 million records per minute) and accurate for intrusion detection.

1 INTRODUCTION

With computers and networked systems playing indispensable roles in almost every aspect of modern society such as industry, government, and economy, cyber security undoubtedly bears the utmost importance in preserving right social orders. However, nowadays, serious cyber-attacks still keep being reported, which have caused significant financial loss and public tensions. One example is the leakage of sensitive, high-profile information from giant marketing establishments or financial institutions. According to a recent study by Ponemon Institute and IBM [13], data breaches cost companies an average of $201 per record in 2014, and the total cost paid by organizations reaches $5.9 millions.

To guarantee information security in the network of computers, an intrusion detection system (IDS) is needed to keep track of the running status of the entire network and identify scenarios that are associated with potential attacks or malicious behaviors. There are two types of intrusion detection approaches, namely the signature-based and anomaly-based intrusion detection approaches. Compared to signature-based methods [3, 18], which can only detect attacks for which a signature has previously been created, anomaly-based intrusion detection aims at identifying unusual entities, events, or observations from a running system that deviate from its normal pattern of behaviors. Detected anomaly patterns can be translated into critical actionable information that can significantly facilitate human decision-making and mitigate the damage of cyber-attacks. Towards this end, anomaly-based intrusion detection [12, 14] turns out to be a particularly useful tool.

Although the recent years have witnessed significant progress of intrusion detection techniques, the rise of big data has introduced new challenges for the design of efficient and accurate anomaly-based intrusion detection approaches. First, IDS typically deals with a large volume of system event data (normally more than 10, 000 events per host per second). The challenge is how to identify abnormal system behaviors from such large-scale (possibly fast streaming) data. Second, the variety of system entity types may necessitate high-dimensional features in subsequent processing. Such enormous feature space could easily lead to the problem, coined by Bellman as "the curse of dimensionality" [2].

More importantly, IDS often has to rely on a *coordinated or sequential, but not independent*, action of multiple system events to determine the security status. This is because system monitoring data are typically low-level process events or interactions between various system entities such as processes, files and sockets (*e.g.*, a program opens a file or connects to a server), while attempted intrusions are higher-level activities which usually involve multiple events together. For example, a network attack called Advanced Persistent Threat is composed of a set of stealthy and continuous computer hacking processes, by first attempting to gain a foothold in the environment, then using the compromised systems as the access into the target network, followed by deploying additional

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM'17 , November 6–10, 2017, Singapore, Singapore

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-4918-5/17/11…$15.00

https://doi.org/10.1145/3132847.3132854

tools that help fulfill the attack objective. The gap between low-level process events and high-level intrusion activities makes it particularly challenging to identify process events that are truly involved in a real malicious activity, and especially considering the massive "noisy" events filling the event sequences.

Hence, the approaches that identify individual process events that confer a given system state are inappropriate to detect sequences of such interactions between process events. Therefore, there exists a vital need for the methods that can detect the sequences of process events that are related to the malicious intrusion activities in an efficient and accurate way.

To address the challenging problem of identifying truly relevant process events to higher-level system attacks from the massive amount of event sequences signifying extremely complex and dynamic behaviour of networked computer systems, in this paper, we introduce GID, a graph-based intrusion detection technique to capture the interaction behaviour among system entities, which sheds important light on event sequences or sequence patterns that are related to intrusion attacks. In particular, we design a compact graph structure that captures the information flow between different system entities. This structure effectively removes the redundancy in system monitoring data and provides the foundation for in-memory intrusion detection solutions. Then, we discover the routine behavior for each system entity in the graph based on a transition probability model. An *anomaly score* is calculated for each candidate event sequence that quantifies its "rareness" in compared with normal profiles. In specific, if the behavior of any involved entity largely deviates from its routine role, the event sequence is marked as suspicious. In this way, we are capable of capturing abnormal event sequences with a high probability.

However, the anomaly score is sensitive to the length of the event sequence. To eliminate the potential score bias from the sequence length, we use the power transformation based approach to normalize the anomaly scores so that the scores of paths with different lengths have the same distribution. To improve the detection accuracy, we perform the validation of the suspicious event sequences to ensure that only those event sequences that sufficiently deviate from normal ones can trigger alerts. We launch an extensive set of experiments on a real-world testbed to evaluate the time performance and detection accuracy of our approach. The results demonstrate that, on a real-world system monitoring dataset that contains 440 million system events, GID is efficient (as much as 2 million records per minute) and accurate. We fully implement GID and deploy it into an enterprise security system, which greatly helps detect advanced threats, and optimize the incident response.

The main contribution of the paper has been summarized as follows:

- We identify the important problem of suspicious event sequence discovery in intrusion detection;
- We design a compact graph model to preserve all the useful information from massive system monitoring data;
- We develop a suspicious path discovery algorithm and prove its convergence on directed acyclic graphs;
- We fully develop the detection engine and deploy it into a real enterprise security system.

2 PRELIMINARIES AND PROBLEM STATEMENT

In this section, we present the preliminaries and problem definition of our work. For the following sections, we assume that the computer system is UNIX system, for simplicity of discussion.

System Entities. System monitoring data, collected by our agent, indicates the interactions between a set of system entities. We consider four types of system entities: (1) *files*, (2) *processes*, (3) *Unix domain sockets* (UDSockets), and (4) *Internet sockets* (INETSockets). Each type of entity is associated with a set of attributes.

System Events. We model the interactions between entities as system events. Formally, a *system event* $e(n_b, n_d, t)$ is a record containing source entity n_b, destination entity n_d, a time stamp t when e happens. The n_b and n_d are entities of possibly different types. In computer systems, a heterogeneous event is a record involving entities of different types such as the files, processes, Unix domain sockets (UDSockets), and Internet sockets (INETSockets). According to the design of modern operating systems, sockets function as the proxy for different processes to communicate. Typically, two processes that execute on the same host communicate with each other via UDSockets, while processes on different hosts communicate with each other by INETSockets. Therefore, on a single host, the interactions exist between the following types of entities: (1) processes and files, (2) processes and sockets (both UDSockets and INETSockets), and (3) UDSockets and UDSockets.

System events can be generated at a high frequency (*e.g.*, tens of thousands events per second). In a modern computer system, plentiful system events can take place "silently" in the backstage without users ever being aware of them. For instance, the *Exim* process, which is the mail transfer agent, frequently accesses the */etc/hosts* file to check the mapping between host names and IP's.

Event Sequence. System events often happen in a chain. For instance, process A first opens file F, then reads F, and sends the content of F to process B. We formulate such chains as the *event sequence*. Formally, a sequence of events $e_1, \ldots, e_{\ell-1}$ that happen in a chain is denoted as $S = \{e_1, e_2, \ldots, e_{\ell-1}\}$, where $e_i.n_d = e_{i+1}.n_b$, i.e., e_i's destination entity is the e_{i+1}'s source entity, and $e_i.t < e_{i+1}.t$ for $i \in \{1, \ldots, \ell-1\}$. The length of S is ℓ. The timespan of a event sequence is $ts = e_{\ell-1}.t - e_1.t$. A simplified representation is to denote the event sequence as $S = \{n_1, n_2, \ldots, n_{\ell-1}, n_\ell\}$, informing that the data is transmitted from n_1 to n_ℓ, via $n_2, \ldots, n_{\ell-1}$ following the time order.

Abnormal Event Sequence. Most of the abnormal system behaviors, such as cyber intrusion, spying, and information stealing, often involve a sequence of low-level events to achieve their goals. It is typically these event sequences, not individual events, that behave differently from regular (or normal) patterns of system behavior (see the formal definition of *abnormality* in Section 3.4). We call those event sequences that are associated with malicious attacks as *abnormal event sequences*. Abnormal system activities, such as cyber attacks, often finish in a relatively short time period, in order to avoid detection. Thus, the time span of abnormal event sequences is short.

To measure the degree of severity of suspicious event sequences, we assign an *anomaly score* to each candidate event sequence (the formulation of anomaly score is in Section 3.4).

Intrusion attacks usually complete in a short time period, in order to avoid being detected. In other words, only those events that are temporally close may be truly involved in committing a cybercrime. Therefore, instead of considering all possible event sequences, we focus on those events with a short time span. More formally, our problem can be defined as follows:

Problem Statement. Given the system monitoring data that contains a set of events $\mathcal{E}$, the user-specified positive integers ℓ, k, and time window size Δt, we aim to find the top k abnormal event sequences in $\mathcal{E}$ that include at most ℓ system events occurring within the time period of Δt.

There are two major challenges in this problem: (1) How to define and compute the anomaly score of event sequence containing multiple heterogeneous entities; and (2) How to rank the event sequences of different lengths at the same time. In the next section, we present our approach to resolve the challenges in detail.

3 ALGORITHM

3.1 Overview

In this paper, we propose GID, a graph-based intrusion detection system, that can find abnormal event sequences from a large number of heterogeneous event traces. Figure 1 shows the framework of GID. In particular, the *graph modeling* component (Section 3.2) generates a compact graph that captures the complex interactions among event entities, aiming to reduce the computational cost in subsequent analysis components. The *candidate path searching* component discovers all the candidate event sequences that may correspond to malicious event sequences that the adversary exploits to disclose sensitive information (Section 3.3). From those identified candidate paths, the *suspicious path discovery* component discriminates those abnormal event sequences from the normal ones (Section 3.4). The distinction between abnormal and normal paths is based on the *anomaly scores* that measure the “rareness” of each candidate event sequence compared with the historical ones. The *suspicious path discovery* component returns those paths of top-k anomaly scores as suspicious paths. To further reduce false alarms, the *suspicious path validation* component measures the deviation between the suspicious sequences from the normal ones, and mark those sequences as abnormal only if their deviation is sufficiently large (Section 3.5). In the following sections, we discuss the details of each component.

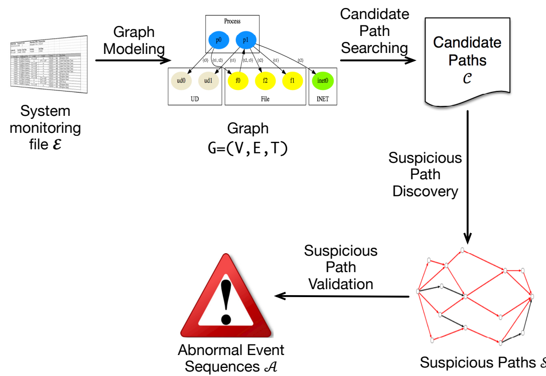


Figure 1: Framework of GID

3.2 Graph Modeling

System monitoring data can be massive. For example, the data collected from a single computer system by monitoring the process interactions in one hour can easily reach 1 GB. Searching over such massive data is prohibitively expensive in terms of both time and space. Therefore, we devise a compact, graph-based representation of the system event data.

The idea of compact graph representation comes from our observation that system events data are often redundant in several ways. The first source of redundancy comes from the attributes, as each event record contains not only the corresponding entities but also attributes from these entities. Repeatedly storing the attributes of those entities in a large number of events introduce significant redundancy. The second source of redundancy comes from the events that involve the same entities; the information of these entities always repeatedly saved (with different time stamps). Furthermore, normally intrusion attacks complete in a short time window. Therefore, it is not necessary to search the data outside of the user-defined time window.

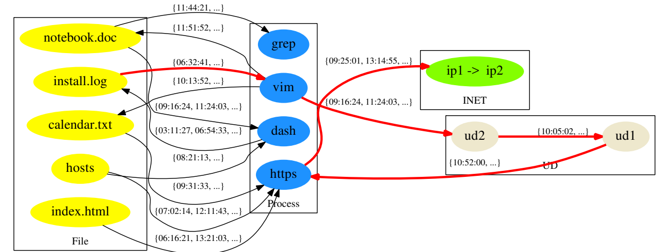


Figure 2: An example of compact graph model; the red path corresponds to an abnormal event sequence

Our graph model eliminates data redundancies. Formally, given the data in a time window, we construct a directed graph $G = (V, E, T)$, with: (1) V as a set of vertices, each representing an entity. For enterprise surveillance data (see Section 4.1), each vertex of V belongs to any of the following four types: files (F), processes (P), UDSockets (U), and INETSockets (I), namely $V = F \cup P \cup U \cup I$; (2) E as a set of edges. For each pair of entities (n_i, n_j), if there exists any system event between them, we construct an edge (v_i, v_j) in the graph, where v_i (v_j) corresponds to n_i (n_j); and (3) T as a set of time stamps. For any edge (v_i, v_j), it is possible that it is associated with multiple timestamps (i.e., the corresponding event happens multiple times). We use $T(v_i, v_j)$ to denote the set of time stamps on which this event has ever happened. Formally, $T(v_i, v_j) = \{e.t | e \in \mathcal{E}, v_i = e.n_b \quad \text{and} \quad v_j = e.n_d\}$. n_b (n_d) is the source (destination) entity of e. Given an event sequence of length ℓ, there is a corresponding path in G that includes ℓ vertices. In the rest of the paper, we will use event sequence and path interchangeably.

In this paper, we are interested in only those event sequences that happen in a given time interval, namely the timespan is less than or equal to Δt, where Δt is a given threshold. Figure 2 shows an example of the compact graph for enterprise surveillance data. Note that according to the types of interactions that are allowed in UNIX, G is not a complete graph. Instead, it only allows the edges between (1) process and file nodes, (2) process and process nodes,

(3) process and socket (both UDSockets and INETSockets) nodes, and (4) UDSocket and UDSocket nodes.

By removing the redundancy of attributes and events, our graph representation can significantly compress the original heterogeneous event data while preserving relevant information for intrusion detection. Our experiment results in Section 4 demonstrate that the graph model reduces the space cost significantly.

3.3 Candidate Path Search

The *candidate path searching* component searches for the paths in G that could correspond to the malicious event sequences whose length is no larger than ℓ. Considering that the graph can be densely connected, we impose the following *time order constraint* on the search procedure, demanding that for each path, its corresponding event sequence must follow the time order. Formally, a path $p = \{n_1, \ldots, n_{r+1}\}$ satisfies the *time order constraint* if $\forall i \in [1, r-1]$, there exists $t_1 \in T(n_i, n_{i+1})$ and $t_2 \in T(n_{i+1}, n_{i+2})$ such that $t_1 \leq t_2$. This condition enforces the time order in the corresponding event sequences.

A straightforward way to generate candidate paths is to apply the path pattern and time-order constraints in a *breadth-first search*. One scan of the system event graph G is sufficient to find all candidate paths. To reduce the space overhead, we calculate the anomaly score (as discussed in Section 3.4) for each candidate path once it is discovered, and only save the path in memory if it is in the top-k list.

3.4 Suspicious Path Discovery

It is possible that some candidate paths discovered by the *candidate path searching* component are not truly associated with system attacks. Hence it is necessary to identify those suspicious paths that are highly likely to be associated with abnormal event sequences among a large set of candidate paths.

A straightforward idea of the suspicious path discovery is to define their anomaly based on the frequency of the system entities that are involved. Those paths that involve rarely-used system entities are considered as suspicious. This is not correct as many intrusion attacks indeed only involve system entities that are popularly used in many events. Consider the enterprise surveillance graph in Figure 2 as an example. The red path shows a typical information leakage attack, via which the secret *install.log* file is leaked through the *vim* and *httpsd* entities. Apparently *vim* is the editor process and the *httpsd* process is a background daemon process that supports https service. Both entities are involved in many normal system events. The frequency-based anomaly approach cannot catch such intrusion attacks.

Continuing the example, we notice that, however, the interaction between *vim* and *install.log* entities is abnormal, as typically the *install.log* file is written by the processes such as *dash*, but not read by the *vim* process, which mainly serves as a file editor.

Therefore, our basic idea is to define the anomaly based on both the system entities and the interactions among them. Each path is assigned an *anomaly score* that quantifies the degree of anomaly. Next, we discuss how to calculate the anomaly scores.

First, we assign each system entity two scores, namely, a *sender* score and a *receiver* score. The sender (receiver, resp.) score measures the activeness that the entity serves as an information flow source (destination, resp.) For instance, the *install.log* file has a high receiver score but relatively low receiver score, as it is frequently modified whenever there is a package installation or upgrade, but it is rarely read. In contrast, the *hosts* file has a high sender score.

Both sender and receiver scores are computed by following the information flow in the system event graph G. In particular, given the graph G, we produce a $N * N$ square transition matrix A, where N is the total number of entities, and

$$A[i][j] = prob(v_i \to v_j) = \frac{|T(v_i, v_j)|}{|\sum_{k=1}^{N} |T(v_i, v_k)||}, \quad (1)$$

where $T(v_i, v_j)$ denotes the set of time stamps on which the event between v_i and v_j has ever happened.

Intuitively, $A[i][j]$ denotes the probability that the information flows from v_i to v_j in G. We denote A as

$$A = \begin{vmatrix} & P & F & I & U \\ P & 0 & A^{P\to F} & A^{P\to I} & A^{P\to U} \\ F & A^{F\to P} & 0 & 0 & 0 \\ I & A^{I\to P} & 0 & 0 & 0 \\ U & A^{U\to P} & 0 & 0 & A^{U\to U} \end{vmatrix}, \quad (2)$$

where 0 represents a zero sub-matrix. Note that the non-zero sub-matrices of A (Equation 2) only appear between processes and files, processes and sockets, as well as UDSockets and UDSockets, but not between processes, because the interaction between process and process does not come with information flow. This is what is allowed by the Unix system.

Let x be the sender score vector, with $x(v_i)$ denoting the node v_i's sender score. Similarly, we use y to denote the receiver score vector. To calculate each node (entity)'s sender and receiver scores, first, we assign initial scores. We randomly generate the initial vector x_0 and y_0 and iteratively update the two vectors by the following

$$\begin{cases} x_{m+1}^T = A * y_m^T \\ y_{m+1}^T = A^T * x_m^T \end{cases}, \quad (3)$$

where T denotes the matrix transpose. According to Equation 3, an entity v_i's sender score is the summation over the receiver scores of the entity to which v_i sends information to. The intuition is that if an entity sends information to a large number entities of high receiver scores, this entity is an important information sender, and it should have a high sender score. Similarly, an entity should have a high receiver score if it receives information from many entities of high sender scores. As a result, an entity v_i's receiver score is calculated by accumulating the sender scores of the entities from which v_i receives information.

From Equation 3, we derive

$$\begin{cases} x_{m+1}^T = (A * A^T) * x_{m-1}^T \\ y_{m+1}^T = (A^T * A) * y_{m-1}^T \end{cases}. \quad (4)$$

In Equation 4, we update the two score vectors independently. It is easy to see that the learned scores x_m and y_m depend on the initial score vector x_0 and y_0. Different initial score vectors lead to different learned score values. It is difficult to choose "good" initial score vector in order to learn the accurate sender and receiver scores. However, we find an important property in matrix theory, namely the *steady state property* of the matrix [9], to eliminate

the effect of x_0 and y_0 on the result scores. Specifically, let M be a general square matrix, and π be a general vector. By repeatedly updating π with

$$\pi_{m+1}^T = M * \pi_m^T, \tag{5}$$

there is a possible convergence state such that $\pi_{m+1} = \pi_m$ for sufficiently large m value. In this case, there is only one unique π_n which can reach the convergence state, i.e.,

$$\pi_n^T = M * \pi_n^T. \tag{6}$$

The convergence state has a good property that the converged vector is only dependent on the matrix M, but independent from the initial vector value π_0. Based on this property, we prefer that the sender and receiver vectors can reach the convergence state. Next, we discuss how to ensure the convergence.

To reach the convergence state, the matrix M must satisfy two conditions: *irreducibility* and *aperiodicity* [9]. A graph G is *irreducible* if and only if for any two nodes v_i, $v_j \in V$, there exists at least one path from v_i to v_j. The period of a node $v \in V$ is the minimum path length from v to v. The graph's period is the greatest common divisor of all the node's period value. A graph G is *aperiodic* if and only if it is irreducible and the period of G is 1.

As our system event graph G is not always a strongly connected, the iteration in Equation (4) may not reach the convergence state. To ensure convergence, we add a *restart matrix* R, which is widely used in random walk on homogeneous graph [21] and bipartite graph [23]. Typically, R is an $N * N$ matrix whose entries are all $\frac{1}{N}$'s. With R, we get a new transition matrix $\bar{A}$:

$$\bar{A} = (1-c) * A + c * R, \tag{7}$$

where c is a value between 0 and 1. We call c the *restart ratio*. With the restart technique, $\bar{A}$ is guaranteed to be an irreducible and aperiodic matrix. By replacing A with $\bar{A}$ in Equation (4), we are able to get the converged sender score vector x and receiver score vector y. We can also control the convergence rate by controlling the restart rate value. Our experiments show that the convergence often can be reached within 10 iterations.

Given a path $p = (v_1, \ldots, v_{r+1})$, based on the sender and receiver score, the anomaly score is calculated as

$$Score(p) = 1 - NS(p), \tag{8}$$

where $NS(p)$ is the regularity score of the path calculated by the following formula:

$$NS(p) = \prod_{i=1}^{r} x(v_i) * A(v_i, v_{i+1}) * y(v_{i+1}), \tag{9}$$

where x and y are the sender and receiver vectors, and A is calculated by Equation 2. In Equation (9), $x(v_i) * A(v_i, v_{i+1}) * y(v_{i+1})$ measures the normality of the event (edge) that v_i sends information to v_{i+1}. Intuitively, any path that involves at least one abnormal event is assigned a high anomaly score. Consider the example of the suspicious path (the red path) in Figure 2. As the *install.log* file has a low sender score, and it is rarely accessed by the *vim* process, the information transition probability between *install.log* and *vim* is low. Therefore, the event sequence is assigned with a high anomaly score.

For each path $p \in C$, we calculate the anomaly score by Equation 8. However, it is easy to see that longer paths tend to have higher anomaly scores than the shorter paths. To eliminate the score bias from the path length, we *normalize* the anomaly scores so that the scores of paths of different lengths have the same distribution. Let $\mathcal{T}$ denote the normalization function. We use the Box-Cox power transformation function [20] as our normalization function. In particular, let $Q(r)$ denote the set of anomaly scores of r-length paths before normalization. For each score $q \in Q(r)$, we apply

$$\mathcal{T}(q, \lambda) = \begin{cases} \frac{q^\lambda - 1}{\lambda} & : \lambda \neq 0 \\ \log q & : \lambda = 0 \end{cases} \tag{10}$$

where λ is a normalization parameter. Different λ values yield different transformed distributions. Our goal is to find the optimal λ value to make the distribution after normalization as close to the normal distribution as possible (i.e., $\mathcal{T}(Q, \lambda) \sim N(\mu, \sigma^2)$).

Next, we discuss how to compute the optimal λ. First, we assume that such λ exists to make $\mathcal{T}(Q, \lambda) \sim N(\mu, \sigma^2)$. The density of a normalized scores is

$$Prob(\mathcal{T}(q, \lambda)) = \frac{\exp(-\frac{1}{2\sigma^2}(\mathcal{T}(q, \lambda) - \mu)^2)}{\sqrt{2\pi}\sigma}. \tag{11}$$

The profile logarithm likelihood of the normalized distribution is

$$\mathcal{L}(Q, \lambda) = -\frac{n}{2}\log(\sum_{i=1}^{n}\frac{(\mathcal{T}(q_i, \lambda) - \mathcal{T}(\bar{q}, \lambda))^2}{n}) + (\lambda - 1)\sum_{i=1}^{n}\log q_i, \tag{12}$$

where $\mathcal{T}(\bar{q}, \lambda) = \frac{1}{n}\sum_{i=1}^{n} T(q_i, \lambda)$.

To minimize the margin between the normalized distributions and a Gaussian distributions, we find a λ that maximizes the log-likelihood. A possible solution is to take derivatives of $\mathcal{L}(Q, \lambda)$ on λ, and pick λ that makes $\frac{\partial \mathcal{L}}{\partial \lambda} = 0$. The *suspicious path discovery* component returns those paths of top-k normalized anomaly scores as suspicious paths.

3.5 Suspicious Path Validation

To further validate the discovered suspicious paths, we calculate the *t-value* between the two groups of paths: all candidate paths C, and the set of discovered suspicious paths $\mathcal{S}$. The t-test returns a confidence score that determines whether the difference between the two sets of paths is significant. If the confidence score is greater than 0.9 with $p - value$ smaller than 0.05, all paths in $\mathcal{S}$ are considered as abnormal paths that are relevant to intrusion attacks. Otherwise, we treat those paths as normal and do not raise alerts.

The suspicious path validation component prevents GID from sending false alarms when there is no attack at all.

3.6 Complexity Analysis

In this section, we discuss the complexity of GID. In the *candidate path* search component, we execute *BFS* for each vertex. Thus, the complexity is $O(|V|(|V|+|E|))$, where $|V|$ and $|E|$ denote the number of vertices and edges in the graph G. In the *suspicious path discovery* component, calculating the sender and receiver scores using Equation 3 takes $O(|E|t)$ time, where t is the number of iterations needed for convergence. In practice, the graph is typically sparse, and so the number of edges is of the same order as the number of vertices, N. Therefore, overall the complexity of GID is is $O(N^2)$.

4 EXPERIMENTS

4.1 Experiment Setup

Dataset. We use a real-world system monitoring dataset in our experiments. The data was collected from an enterprise network composed of 33 UNIX machines, in a time span of three consecutive days (*i.e.*, 72 hours). The sheer size of the data set is around 157 Gigabytes. We consider four different types of system entities: (1) *files*, (2) *processes*, (3) *Unix domain sockets* (UDSockets), and (4) *Internet sockets* (INETSockets). Each type of entities is associated with a set of attributes and a unique identifier. Two types of events (i.e., interactions between the system entities) are considered in this paper: (1) file accessed by the processes; and (2) communication between processes. According to the design of modern operating systems, sockets function as the proxy for different processes to communicate. Typically, two processes that execute on the same host communicate with each other via UDSockets, while processes on different hosts communicate with each other by INETSockets. Thus, on a single host, the interactions exist between the following types of entities: (1) processes and files, (2) processes and sockets (both UDSockets and INETSockets), and (3) UDSockets and UDSockets. In total, there are around **440 million** system events. These events are related to $410,166$ processes, $1,797,501$ files, $185,076$ UDSockets and $18,391$ INETSockets.

Testbed and Parameters. We implement our algorithm in Java and run it on a PC with a 2.5GHz CPU and 8GB RAM. We set the *time window size* as one hour, namely, we are interested in catching intrusions which occur within an hour. We use the *tumbling window* model to process the stream data for simplicity. By default, we set $\ell = 5$. Various k values are used in order to thoroughly evaluate the detection accuracy. Regarding the restart ratio c, the detection accuracy reaches a plateau as c grows from 0.5 to 0.9, which indicates that GID is insensitive to the choice of c [22]. In the experiment, we set $c = 0.6$.

Based on the default setting, for each one-hour time window, GID returns the most suspicious event sequences whose lengths are no larger than 5.

Attack Description. There are 10 different types of attacks with various lengths from 3 to 5. For each type of attacks, we tried 10 attack scenarios at different time slots throughout the data collection period, which results in total 300 event sequences that correspond to intrusion attacks into the data. All the 10 types of attacks exploit event sequences to transmit sensitive information to an unauthorized party [7, 17]. Due to the space limits, here we only list the three major types of attacks.

- **Type 1.** This attack targets at the */selinux/mls* file, which defines the MLS (Multi-Level Security) classification of files within the host. In general, the */selinux/mls* file should be kept secret to all users except for the security administrator, as it exposes the security rules of a computer system and enables the attackers to find potential vulnerabilities. By the intrusion attack, the attacker first exploits the *ssh* process to access */selinux/mls* file. If the file access is successful, the file content is sent to an external host (i.e., the attacker).
- **Type 2.** This attack targets at the */etc/passwd* file, which stores the password digest of all users as well as the user group information. First, the attacker tries to access the */etc/passwd* file by the *gvfs* process, which enables easy access from a remote host via FTP. Then the attacker tries to send the file via an *INETSocket*.
- **Type 3.** In this attack, the remote intruder employs the *bash* process to scan a sensitive file */var/log/apt/history.log*. This file stores detailed installation messages. The attackers are interested in it as they can intrude the host by exploiting the vulnerabilities of the installed software. The sensitive information is leaked via an *INETSocket*.

Type 1 and 2 attacks are the essential initial intrusion steps committed by the Snowden attack, while Type 3 attacks correspond to the botnet attack where the zombie computer gathers and delivers the unauthorized information to a command and control (C&C) server.

Baseline. We compare our algorithm with a number of state-of-the-art algorithms and the variations of GID. We briefly introduce these baseline approaches:

- *OutRank* [16]: This approach leverages graph-based approach to detect anomalies from a set of objects. The transition probability is defined as the similarity between a pair of objects. The values in the dominant eigenvector are used to determine the anomaly degree of each object. This approach suffers high computational overhead due to the similarity computation of all the pairs.
- *NGRAM* [3]: This method has been widely studied for the identification of attacks and malicious software. This method builds the profiles of normal system behaviors, and labels those events in the testing data that do not appear in the normal profiles as abnormal ones. In our experiments, we use the first 4 hours monitoring data as the training set.
- *iBOAT* [6]: This method has shown its effectiveness in suspicious trajectory discovery in GPS traces. It defines the abnormal events as those whose corresponding paths have low confidence score in the dataset. In the experiments, we set the threshold value to be 0.5, which is already the lowest confidence we can set to get the largest number of attacks detected.
- *PAGE*: This approach exploits the famous PageRank [21] algorithm to compute the entity score and calculates the anomaly score for *single events* based on the entity scores. In this approach, the direction of information flow is ignored. Also, instead of assigning both the sender and receiver score to any entity as in GID, this approach only calculates a single score for each entity based on the steady state distribution of the Markov chain process. An event is reported as "abnormal" if its anomaly score reaches the threshold.
- *GID -UNNORMAL* In this approach, we intentionally avoid the normalization of the anomaly scores of paths with different length. We compare GID -UNNORMAL to demonstrate the effectiveness of the normalization in improving the detection accuracy.

Evaluation Metrics. We compare GID with the baselines in terms of detection accuracy and time performance.

- Accuracy: The accuracy is measured using *true positive rate (TPR)* and *false positive rate (FPR)*. Intuitively, the *TPR* defines the fraction of intrusion attacks that are detected during the

test. *FPR*, on the other hand, describes the fraction of normal event sequences that trigger an alert in the test. We plot the ROC curve based on the TPR and FPR obtained under various k values.

- Time: We measure the time consumed to detect abnormal event sequences. In order to deploy GID into enterprise systems, we expect a low detection latency.
- Memory usage: We quantify the memory consumed by GID to simultaneously monitor and detect intrusion attacks for all the 33 machines. This is an important performance indicator in real applications.

Experiment Settings. We evaluate GID in the following settings:

(1) *Static:* We fetch the events in the monitoring data collected in the 10-th hour, and execute the detection algorithms on these events offline. In specific, the monitoring data is fed to the detection algorithms all at once. All the required data is stored in memory. There exist in total **8 million** system events. We launched 12 attacks during the 10-th hour. We aim at evaluating the performance and parameter sensitivity of GID before testing it on the streaming data.

(2) *Streaming:* The monitoring data is delivered to GID in a streaming fashion, i.e, one system event is processed at a time. GID updates the graph (in particular, the edge weights) and the sender and receiver scores of all the entities according to the incoming events. A snapshot of the entity scores is retained every hour for evaluation. We use this setting to simulate the application of GID in enterprise security system.

4.2 Static Evaluation

Detection Accuracy. We compare the detection accuracy of GID with the baseline approaches. To quantify the detection accuracy, we choose different k values and compare the detected alerts with the ground truth event sequences related to attacks. Based on the result, we plot the ROC curves in Figure 3. The result demonstrates the effectiveness and accuracy of GID in detecting the attacks. The accuracy provided by GID is much better than the baselines. Specifically, among altogether 12 attacks, only two of them exploit system entities that are not included in the normal profile. Thus *NGRAM* can only identify these two attacks. OutRank fails to detect the majority of real attacks, such as Type 2 and 3. This is because the involved system entities, i.e., *gvfs* and *history.log*, commonly fall inside system events. *iBOAT* introduces a large number of false positives, as the confidence scores between certain processes and files follow a polarized distribution. GID -*UNNORMAL* introduces a bias for long sequences, as they typically have higher anomaly scores. However, as the number of long sequences is limited, the accuracy is still acceptable. *PAGE* does not consider the information flow direction and only targets on single events. Thus, even though history.log is rarely read, it fails to detect the Type 3 attack, as the file is often updated. Moreover, it triggers a large amount of false positive alerts.

In practice, it is unrealistic to expect users to provide an accurate estimation of the length ℓ of real attack sequences. Therefore we measure the impact of the choice of ℓ on the detection accuracy. As the length of ground truth event sequences is at most 5, we vary ℓ from 5 to 10, and report the ROC curves in Figure 4. We observe that even though a larger ℓ tends to hamper the accuracy, the effect is quite negligible especially when $\ell \geq 8$. This is because

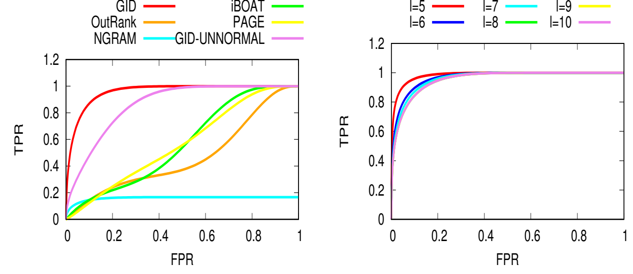


Figure 3: ROC curve over static data

Figure 4: ROC curve *w.r.t.* various ℓ

the number of long event sequences is quite small. For example, the number of sequences of length $l = 5$ is $13,675$; while that for $\ell = 6$ is only 625. Therefore, given the limited number of long sequences, the detection accuracy of GID is not very sensitive to the choice of ℓ. This makes our approach more plausible in practical scenarios.

Time Performance. We compare the time performance of GID with the baseline approaches. The result is displayed in Figure 5. We omit the time performance of *PAGE* and *GID-UNNORMAL* as they are very close to that of GID. GID is significantly more efficient than *OutRank*. The main complexity of *OutRank* comes from computing the pairwise similarity between all the event sequences. Among the baseline approaches, *NGRAM* is the most efficient as the normal profile is pre-computed and requires no update. It only needs to find the event sequences and check the existence in the normal profile. However, it still takes longer time than GID due to the large size of the normal profile. *iBOAT* takes more time than GID due to the confidence computation.

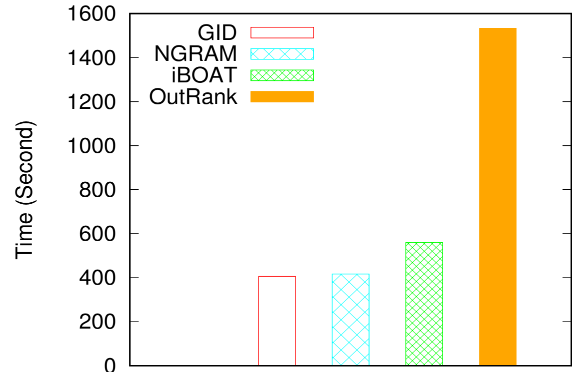


Figure 5: Time performance over static data

4.3 Streaming Evaluation

Detection Accuracy. The frequency at which snapshots of the entity scores are updated can have a dramatic impact on the detection accuracy. Let W denote the period to update the snapshot. Intuitively, a smaller W leads to more dynamic updates in the normal profile in locating anomalies from incoming event sequences. In Figure 6, we evaluate the detection accuracy of GID with regard to various W. It is obvious that GID yields the best accuracy when the update period W is small. But it is also worth noting that when

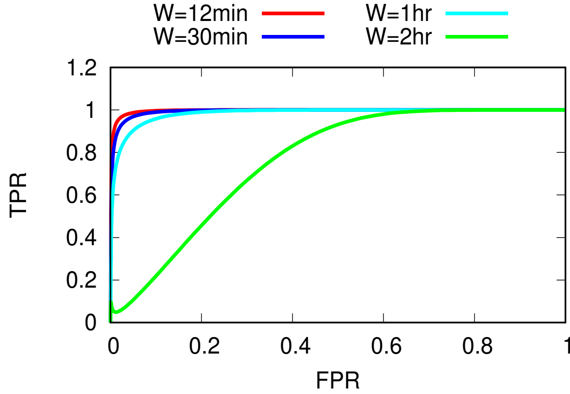


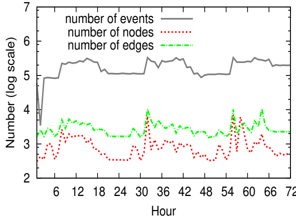
Figure 6: ROC curve *w.r.t.* update period

W is smaller than 1 hour, the benefit of decreasing W can be trivial. Given the fact that smaller W induces more overhead in updating the snapshot, we figure out that GID reaches the best balance between detection accuracy and update overhead when $W = 1hr$. Therefore, in the experiment, we update the snapshot every 1 hour.

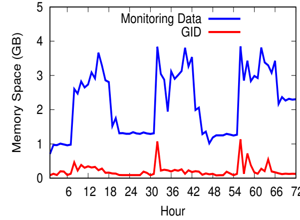
Memory Usage. To show the compactness of the graph model, we measure the size of the constructed graph.

	Process	File	INETSocket	UDSocket	Edge
Avg.	117.3	191.36	0.93	41.42	1668.4
Max	1468	23290	130	6735	58555

(a) Avg/max. number of system entities and edges



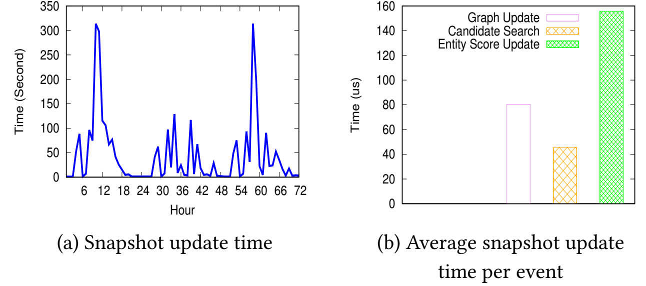
(b) Graph size vs. number of events



(c) Memory usage comparison of GID and the monitoring data

Figure 7: Graph compactness

In Figure 7 (a), we report the size of our system event graph in terms of the number of system entities and edges. Specifically, we construct a graph per host per hour. Thus based on the monitoring data for 33 machines and 3 days, there are 72*33 = 2, 376 graphs. On average, each graph contains around 351 nodes with four different types and less than 1.7K edges. Even for the worst case, the graph is still within the size of 60K edges. In Figure 7 (b), we show the average size of the graphs for each hour in a 72-hour time window. The number of nodes and edges are measured by averaging the size of the 33 graphs (for 33 hosts) in one hour. The results show that the graph can indeed reduce the space dramatically. One interesting observation is that the graph reaches its largest size at the 10-th, 34-th and 58-th hour. These 3 peak hours correspond to the 10am of Day 1, Day 2 and Day 3 respectively. Normally 10am is the busiest time of system logging as it is when most employees arrive at office. We also directly compare the memory space consumed by GID and the size of the monitoring data in each snapshot. The result is displayed in Figure 7 (c). The memory usage of GID is around one tenth of that demanded by the monitoring data. It demonstrates the graph model compactly compresses the massive heterogeneous monitoring data. Therefore, GID maintains the scalability to be deployed for real-world enterprise application.



(a) Snapshot update time

(b) Average snapshot update time per event

Figure 8: Time performance over streaming data

Time Performance. When executing GID on the streaming data, the main computational bottleneck stems from the snapshot update. The candidate path searching and anomaly score calculation can be easily performed on the fly. However, the periodical snapshot update can be time consuming, especially when the events arrive at a higher velocity. We show the time performance of snapshot update in Figure 8 (a). In most cases, the snapshot update is efficient (within 2 minutes). However, at the peak hours when the incoming events explode, the update can take up to 8 minutes. Considering that the snapshot only needs to be updated every hour, the time overhead of graph update is affordable. We further analyze the update time for each incoming event. Recall that each incoming event triggers four operations: updating the graph, re-calculating the entity score, searching for candidate paths, and anomaly score calculation. Given the sender and receiver scores, it is extremely efficient to calculate the anomaly score for each candidate path. Thus, we only measure the time consumed by the first three operations and report the result in Figure 8 (b). We observe that for each event, the time overhead is negligible. On average, the operations triggered by each event only takes 0.28 millisecond. Therefore, GID can be scaled up to 4, 000 events per second.

5 RELATED WORK

5.1 Anomaly Detection

Existing methods for intrusion detection can roughly be categorized into two classes, namely anomaly detection and misuse detection [11]. Anomaly detection approaches define and characterize normal/abnormal behaviors of the system, while the misuse detection approaches monitor explicit patterns, with the intrusion patterns known in advance. In this paper, we focus on anomaly detection methods. Based on the data representation, we put the existing work of anomaly detection into the following two categories.

Event-based anomaly detection monitors and analyzes the process events of a computing system. Traditionally, system calls serve as a good basis for event-based analysis, as short sequences of system calls are a good discriminator for several types of intrusions [3]. [3] builds the profile of k-grams from normal system call traces.

An alert is thrown if a new system call trace is significantly different from the normal profile. [18] extends this work by taking the system call argument values into consideration. [15] considers two events with high posterior probability in the normal training data to be a good predictive pattern of normal status. In the detection phase, any violation of such pattern is recognized to be abnormal. In all these methods, a purely normal and exhaustive set of training data is essential for constructing a robust normal profile, which substantially degrades the practicability.

In contrast, **graph-based** anomaly detection models the information flow in a computer system using directed graphs, and extracts abnormal substructures from it. Based on *minimum description length* (MDL) principle, [8, 19] discover those small but rarely-happened substructures in the procedure of compressing the graph. However, the MDL principle does not fit the highly dynamic computer system graph. To overcome this problem, [23] explores the anomalies based on the community structure in an evolutionary graph. Because the concentration is limited to the graph structure, a wealth of information to describe an attack, including event timestamps and entity attributes, is disregarded. As a result, the discovered anomalies may not necessarily relate to a cyber attack. [5] defines various notions of the rarity in order to discover novel links from a graph. However, even though efficient, the straightforward rarity measurement may not be able to catch sophisticated cyber attacks.

5.2 Similarity Search in Graphs

Graph similarity search plays a key role in information retrieval and recommendation systems. [21] initiates the research on similarity search in graphs by proposing a random web surfer model to evaluate the importance of each webpage. If a random surfer stops at a webpage with high probability after a sufficiently large time, this page is of great importance. To avoid the rank sinks such as circles with no out-edges, a restart matrix is taken into consideration to model the behavior that the surfer periodically gets bored and jumps to a random page. Following this work, similarity search strategies have been successfully applied to different settings such as personalized recommendation and information retrieval [4, 10]. But most of them focus on homogeneous graphs. Limited papers make a bold attempt to heterogeneous graphs. Among them, [23] utilizes graph partition and relevance search to detect anomalies in undirected bipartite graph. [1] exploits the cyclic structure to rank nodes in cyclic multipartite graphs. PathSim [24] extracts top-k similar entities to an input entity from a heterogeneous network. The similarity between two entities is measured based on the Jaccard similarity of paths that are consistent with the meta-path. A big difference between GID and these methods is that the search criteria of GID is the anomaly score that reflects the typical system behavior in terms of information flow, but not similarities.

6 CONCLUSION

In this paper, we investigate the problem of detecting intrusions, especially suspicious event sequences, in enterprise systems. Different from traditional methods that focus on detecting single entities/events, we propose GID, a graph-based method to capture the interaction behavior among different entities and identify abnormal event sequences. An event sequence is evaluated to be suspicious if any entity functions differently from its role. In this way, even the abnormal activities only involve ordinary entities, we are still able to catch such anomalies. We implement and deploy our approach to a real enterprise security system, and evaluate the proposed algorithm in extensive experiments. The experiment results convince us of the effectiveness and efficiency of our approach.

REFERENCES

[1] Niels Becker. 2013. *Ranking on multipartite graphs. Diploma Thesis. Ludwig Maximilian University of Munich, Munich.*

[2] Richard Bellman. 1961. *Adaptive control processes: a guided tour.* Princeton University Press.

[3] Marco Caselli, Emmanuele Zambon, and Frank Kargl. 2015. Sequence-aware intrusion detection in industrial control systems. In *Proceedings of the Workshop on Cyber-Physical System Security.* 13–24.

[4] Soumen Chakrabarti. 2007. Dynamic personalized pagerank in entity-relation graphs. In *Proceedings of the International Conference on World Wide Web.*

[5] Hans Chalupsky et al. 2003. Unsupervised link discovery in multi-relational data via rarity analysis. In *Proceedings of the International Conference on Data Mining (ICDM).* 171–178.

[6] Chao Chen, Daqing Zhang, Pablo Samuel Castro, Nan Li, Lin Sun, Shijian Li, and Zonghui Wang. 2013. iBOAT: Isolation-based online anomalous trajectory detection. *IEEE Transactions on Intelligent Transportation Systems* (2013).

[7] Abhishek Das, Gokhan Memik, Joseph Zambreno, and Alok Choudhary. 2010. Detecting/preventing information leakage on the memory bus due to malicious hardware. In *Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 861–866.

[8] William Eberle, Jeffrey Graves, and Lawrence Holder. 2010. Insider threat detection using a graph-based approach. *Journal of Applied Security Research* 6, 1 (2010), 32–81.

[9] JP Jarvis and Douglas R Shier. 1999. Graph-theoretic analysis of finite Markov chains. *Applied Mathematical Modeling: A Multidisciplinary Approach* (1999).

[10] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *Proceedings of the International Conference on World Wide Web (WWW).* 271–279.

[11] Anita K Jones and Robert S Sielken. 2000. Computer system intrusion detection: a survey. *Computer Science Technical Report* (2000), 1–25.

[12] V Jyothsna, VV Rama Prasad, and K Munivara Prasad. 2011. A review of anomaly based intrusion detection systems. *International Journal of Computer Applications* 28, 7 (2011), 26–35.

[13] Ponemon L. 2014. Cost of data breach study: global analysis. *Poneomon Institute Sponsored by Symantec* (2014).

[14] Shih-Wei Lin, Kuo-Ching Ying, Chou-Yuan Lee, and Zne-Jung Lee. 2012. An intelligent algorithm with feature selection and decision rules applied to anomaly intrusion detection. *Applied Soft Computing* 12, 10 (2012), 3285–3290.

[15] Matthew V Mahoney, Philip K Chan, and Muhammad H Arshad. 2003. *A machine learning approach to anomaly detection.* Technical Report. *Florida Institute of Technology.*

[16] HDK Moonesignhe and Pang-Ning Tan. 2006. Outlier detection using random walks. In *International Conference on Tools with Artificial Intelligence (ICTAI).*

[17] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar R Weippl. 2011. Dark clouds on the horizon: using clouds storage as attack vector and online slack space.. In *USENIX Security Symposium.* San Francisco, CA, USA, 65–76.

[18] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. 2006. Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC)* 9, 1 (2006), 61–93.

[19] Caleb C Noble and Diane J Cook. 2003. Graph-based anomaly detection. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD).* 631–636.

[20] Jason W Osborne. 2010. Improving your data transformations: applying the Box-Cox transformation. *Practical Assessment, Research & Evaluation* 15 (2010).

[21] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: bringing order to the web.* Technical Report. *Stanford Digital Library Technologies Project.*

[22] Jia-Yu Pan, Hyung-Jeong Yang, Christos Faloutsos, and Pinar Duygulu. 2004. Automatic multimedia cross-modal correlation discovery. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD).*

[23] Jimeng Sun, Huiming Qu, Deepayan Chakrabarti, and Christos Faloutsos. 2005. Neighborhood formation and anomaly detection in bipartite graphs. In *Proceedings of the International Conference on Data Mining (ICDM).* 418–425.

[24] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. 2011. Pathsim: meta path-based top-k similarity search in heterogeneous information networks. In *Proceedings of the International Conference on Very Large Databases (VLDB).*