WATSON: Abstracting Behaviors from Audit Logs via Aggregation of Contextual Semantics

Jun Zeng[†] Zheng Leong Chua[‡+] Yinfang Chen[†] Kaihang Ji[†] Zhenkai Liang[†] Jian Mao[§*]
[†]School of Computing, National University of Singapore
[‡]Independent Researcher
[§]School of Cyber Science and Technology, Beihang University
{junzeng, yinfang, kaihang, liangzk}@comp.nus.edu.sg czl@iiyume.org maojian@buaa.edu.cn

***Abstract*—Endpoint monitoring solutions are widely deployed in today's enterprise environments to support advanced attack detection and investigation. These monitors continuously record system-level activities as audit logs and provide deep visibility into security incidents. Unfortunately, to recognize behaviors of interest and detect potential threats, cyber analysts face a semantic gap between low-level audit events and high-level system behaviors. To bridge this gap, existing work largely matches streams of audit logs against a knowledge base of rules that describe behaviors. However, specifying such rules heavily relies on expert knowledge. In this paper, we present WATSON, an automated approach to abstracting behaviors by inferring and aggregating the semantics of audit events. WATSON uncovers the semantics of events through their usage context in audit logs. By extracting behaviors as connected system operations, WATSON then combines event semantics as the representation of behaviors. To reduce analysis workload, WATSON further clusters semantically similar behaviors and distinguishes the representatives for analyst investigation. In our evaluation against both benign and malicious behaviors, WATSON exhibits high accuracy for behavior abstraction. Moreover, WATSON can reduce analysis workload by two orders of magnitude for attack investigation.**

I. INTRODUCTION

Security incidents in large enterprise systems have been on the rise globally. We have been witnessing attacks with increasing scale and sophistication. Capital One reported that 106 million customers' credit card information was exposed due to unauthorized database access [6]. A recent Twitter attack has left dozens of high-profile accounts displaying fraud messages to tens of millions of followers [14]. To better prevent and respond to such attacks, endpoint monitoring solutions (e.g., Security Information and Event Management (SIEM) tools [5]) are widely deployed for enterprise security. These monitors continuously record system-level activities as audit logs, capturing many aspects of system's execution states.

When reacting to a security incident, cyber analysts perform a causality analysis on audit logs to discover the root cause of the attack and the scope of its damages [45], [46]. However, the amount of audit logs generated by a normal system is non-trivial. Even one desktop machine can easily produce over one million audit events per day [27], [50], let alone busy servers in cloud infrastructures. To overcome this challenge, recent research solutions scale up causality analysis by eliminating irrelevant system operations in audit logs [21], [34], [40], [50], [55], [61], [75], [82]. An alternative research direction aims to increase the efficiency of log query systems [27], [28], [32], [70]. Unfortunately, neither data reduction nor searching improvement leads to a substantial decrease in analysis workload. These solutions do not capture the semantics behind audit data and leave behavior recognization to analysts [53]. As a result, intensive manual effort is still involved in evaluating relevant yet benign and complicated events that dominate audit logs. Especially, a significant problem faced by analysts is a *semantic gap* between low-level audit events and high-level system behaviors.

Existing work strives to bridge this gap by matching audit events against a knowledge store of expert-defined rules that describe behaviors, such as tag-based policies [38], [39], query graphs [62], [84], and TTP (tactic, technique, and procedure) specifications [35], [63]. Essentially, these solutions identify high-level behaviors through tag propagation or graph matching. However, an expected bottleneck is the manual involvement of domain experts to specify such rules. For example, MORSE [39] needs experts to traverse system entities (e.g., files) and initialize their confidentiality and integrity tags for tag propagation. TGMiner [84] requires manual behavior labeling in training log sets before mining discriminative behavioral patterns and searching for their existence in test sets. Despite crucial role in audit log analysis, mapping events to behaviors heavily relies on expert knowledge, which may hinder its applications in practice.

Extracting representative behaviors from audit events for analyst investigation provides an efficient strategy to mitigate this problem. More concretely, we can use procedural analysis to automatically abstract high-level behaviors and cluster semantically similar ones, albeit without the labels explaining what they are. However, because repetitive/comparable behaviors have already been clustered, analysts only need to label the representatives from clusters, resulting in far fewer events to be investigated. Besides reducing manual workload in behavior analysis, automatic behavior abstraction also enables proactive analysis to detect unusual behavioral patterns in insider threats or external exploits. Particularly, any deviation of normal behaviors can be flagged efficiently for attack response.

[*]Corresponding author.[+]Research done at National University of Singapore.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24549
www.ndss-symposium.org

While behavior abstraction sounds promising, there are two main challenges to extract behaviors and infer their semantics: event semantics differentiation and behavior identification. Audit events record general-purpose system activities and thus lack knowledge of high-level semantics. A single event, such as process creation or file deletion, can represent different semantics in different scenarios. Furthermore, due to the large-scale and highly interleaving nature of audit events, partitioning events and identifying boundaries of behaviors are like finding needles in a haystack.

To address the above challenges, our first key insight is that the semantics of audit events can be revealed from the contexts in which they are used. Intuitively, we can represent behaviors by aggregating the semantics of their constituent events. With such representations, similar behaviors can be clustered together. In addition, we observe that the information flow of system entities provides a natural boundary of high-level behaviors. It can serve as guidance to correlate audit events belonging to individual behaviors.

In this paper, we present WATSON [1], an automated behavior abstraction approach that aggregates the semantics of audit events to model behavioral patterns. It does not assume expert knowledge of event semantics to perform behavior abstraction. The semantics is obtained automatically from the context of event usage in audit logs. We call this the *contextual semantics* of events. More specifically, WATSON first leverages a translation-based embedding model to infer the semantics of audit events based on contextual information in logs. Then, WATSON identifies events connected to related data objects (i.e., files and network sockets) and aggregates their semantics as the representation of high-level behaviors. Finally, WATSON clusters similar behaviors recorded in audit logs and distinguishes the representatives for analyst investigation.

To the best of our knowledge, WATSON is the first approach that automatically abstracts high-level behaviors from low-level log information. WATSON provides a quantitative representation of the semantics for both events and behaviors found in audit logs and a way to derive them automatically. Having a quantitative method to reason about behaviors and events gives analysts the ability to compare, sort, and cluster behaviors. It also enables the composition of multiple behaviors and even the synthesis or prediction of what a particular behavior should be. These capabilities can form the basis for designing new security solutions, such as abnormal behavior detection, or supporting existing solutions to select appropriate behaviors for deep inspection.

We prototype WATSON and evaluate its correctness and explicability using 17 daily routines and eight real-life attacks simulated in an enterprise environment. In addition, we use the public DARPA TRACE dataset [13] released by the DARPA Transparent Computing program to evaluate WATSON's efficacy in attack investigation. We note that WATSON is the first to abstract both benign and malicious behaviors for evaluation. Previous techniques [33], [35], [36], [39], [63] do not take benign behaviors into consideration because they mainly focus on attack detection. However, WATSON extracts high-level behaviors regardless of their security concerns. Experimental results show that WATSON accurately correlates system entities with similar usage contexts and achieves an average F1 score of 92.8% in behavior abstraction. Moreover, WATSON is able to reduce the amount of audit logs for analyst investigation by two orders of magnitude.

[1]Our approach name comes from Dr. Watson, who always provides trustful assistance for the detector, Sherlock Holmes.

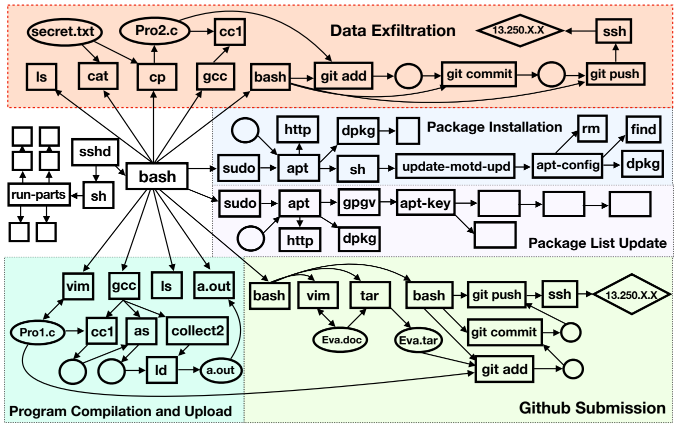


Fig. 1: The provenance graph for the motivation example. Nodes in the graph are system entities (rectangles are processes, ovals are files, and diamonds are sockets), and edges among nodes represent system calls. For readability, we present only a fragment of the graph and highlight high-level behaviors with colored boxes.

In summary, we make the following contributions:

- We present WATSON, the first approach to abstracting high-level behaviors from low-level logs without analyst involvement. Our approach summarizes behaviors using information flow as guidance and derives behavior semantics by aggregating contextual semantics of audit events.
- We propose the novel idea of inferring log semantics through contextual information. We provide a quantitative representation of behavior semantics and use it to cluster semantically similar behaviors and extract representatives.
- We prototype WATSON and conduct a systematic evaluation with both commonly-used benign behaviors and real-world malicious behaviors. The results show that WATSON is effective in abstracting high-level behaviors and reducing human workload in the analysis of logs.

II. BACKGROUND AND MOTIVATION

In this section, we first introduce audit log analysis and its challenges with a motivating example. We then analyze the problem of behavior abstraction with our insights, as well as describing the threat model.

A. Motivating Example

Scenario. Consider the following attack scenario, where an employee wants to exfiltrate sensitive data that he can access. As a software tester, his regular tasks include using `git` to synchronize code from the Github repository, using `gcc` to compile programs from source code, and using `apt` to install tested software dependencies. One day, he locates a sensitive document (`secret.txt`) and wants to exfiltrate it. To evade detection, he tries to mimic normal behaviors in his daily jobs.

He first copies the sensitive file to his working directory as `Pro2.c`. Then, he compiles the "program" using `gcc` and uploads it to a Github repository under his control. Note that the compilation of the file is unsuccessful because `Pro2.c` is not a legitimate source file. This strategy attempts to disguise the Data Exfiltration behavior as part of an ordinary program development activity and thus misguide analysts to flag it as a daily behavior.

Audit log analysis. System audit logs enable analysts to gain insight into cyber attacks with data provenance. Each audit event records an OS-level operation (i.e., system call) such as process execution, file creation, and network connection. Specifically, an event can be defined as a triple (*Subject*, *Relation*, *Object*), where *Subject* is a process entity, *Object* is a system entity (i.e., process, file, or network socket), and *Relation* is a system call function. In our motivating example, copying `secret.txt` events are represented as (`cp, read, secret.txt`) and (`cp, write, Pro2.c`). Note that system entities are associated with a set of attributes for identification, such as labels (e.g., PID and inode) and names (e.g., file path, process path, and IP address). Moreover, every individual event (e.g., a process writing a file) stands for an information flow between *Subject* and *Object*.

To facilitate attack causality analysis, the research community employs a provenance graph [17], [65] to allow tracking information flows efficiently in audit logs. In essence, the provenance graph is a common representation of historic causalities in the system at the OS level. Figure 1 shows an example of a provenance graph building upon the motivating example. The direction of an edge indicates how data transfer between system entities. In an investigation of a given security incident, analysts search through a provenance graph for pieces of information related to cyber-attacks. In Figure 1, analysts first perform backward tracking from an incident (i.e., file upload to an unknown Git repository) to identify its root cause. Then, analysts perform forward tracking on the found initial compromise point (i.e., insider login with `ssh`) to explore the ramifications of the same attack. When inspecting ancestries and progenies of a security incident through backward and forward tracking, analysts can reason about how the incident is caused and the high-level behaviors responsible for it.

B. Challenges

While capturing attack sequences and provenances, analysts would have to recognize not only malicious (e.g., Data Exfiltration) but benign behaviors (e.g., Program Compilation and Upload). Although a provenance graph provides an intuitive representation to visualize causal dependencies and remove irrelevant events, analysts still spend excessive time in investigating relevant but benign events due to the ubiquitous presence of normal activities on a daily basis.

Abstracting behaviors from audit events is an efficient strategy for analysts to navigate through a large number of events and focus on specific information. Essentially, behaviors represent an abstraction of audit data. Working on the level of behaviors can effectively reduce the analysis workload from the whole event space to behaviors of interests that draw attention in a specific scenario. Examples of such behaviors include Data Exfiltration, Backdoor Installation, and Program

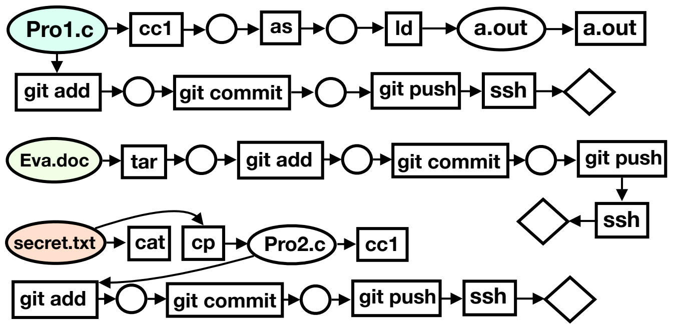


Fig. 2: Subgraphs starting from `Pro1.c`, `Eva.doc` and `secret.txt` correspond to the Program Compilation and Upload, Github Submission and Data Exfiltration behaviors. We color-code the source data object in behaviors and omit some edges (e.g., `vim` writes `Pro1.c`) for clarity.

Compilation. However, to automatically abstract high-level behaviors from low-level audit events, analysts face two major challenges:

- *Inferring semantics of OS-level audit events.* Audit events record detailed system execution states but lack knowledge of high-level semantics critical for behavioral pattern recognition. Specifically, two system entities with similar semantics may be named differently. For example, temporary files for IPC between C compiler (`cc1`) and assembler (`as`) can be named `/tmp/ccw4T8Hh.s` and `/tmp/cc0JjLYr.s`. On the other hand, system entities sharing the same names could indicate different intentions. In our motivating example, the Data Exfiltration behavior leverages `cc1` to mimic normal user activities, while the Program Compilation and Upload behavior uses `cc1` to compile the source code into assembly code. In order to uncover event semantics, existing work largely parses audit events with a knowledge store of expert-defined rules or models. However, the manual specifications could easily undermine the scalability of behavior abstraction due to the large scale of audit events, even in modestly sized systems.
- *Identifying behavior boundaries in large-scale audit events.* The volume of audit data is typically overwhelming, and audit events are highly interleaving. In our motivating example, even a single Package Installation behavior with `apt` generates over 50,000 events. In addition, all individual behaviors are causally connected, as shown in Figure 1. This makes it challenging for analysts to partition events and distinguish behavior boundaries. More notably, most of the events do not necessarily reflect behaviors but system routines. When compiling programs with `gcc`, the child process, `cc1`, reads a massive number of files such as C header files, but only operations accessing `Pro2.c` are particularly correlated with the Data Exfiltration behavior.

C. Problem Analysis

Given a large set of audit logs in a user login session, we aim to identify high-level (benign and malicious) behaviors and provide a quantitative representation of their semantics without analyst involvement. Furthermore, we also aim to cluster semantically similar behaviors and distinguish the representatives

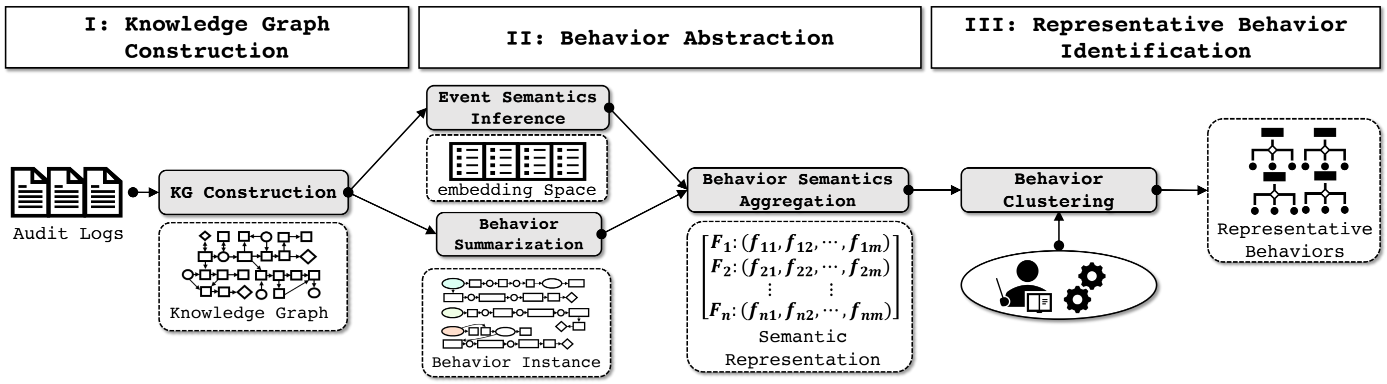

I: Knowledge Graph Construction
II: Behavior Abstraction
III: Representative Behavior Identification
Audit Logs
KG Construction
Knowledge Graph
Event Semantics Inference
embedding Space
Behavior Summarization
Behavior Instance
Behavior Semantics Aggregation
$F_1: (f_{11}, f_{12}, \cdots, f_{1m})$
$F_2: (f_{21}, f_{22}, \cdots, f_{2m})$
$F_n: (f_{n1}, f_{n2}, \cdots, f_{nm})$
Semantic Representation
Behavior Clustering
Representative Behaviors

Fig. 3: WATSON Overview.

for human inspection. In contrast to traditional approaches, where behavior abstraction relies heavily on domain knowledge, our goal is to achieve automated behavior abstraction using simple yet effective insights.

Our first insight comes from the observation that *the semantics of system entities and relations in audit events can be revealed from the context in which they are used.* For example, while compiling programs using `gcc`, C compiler (`cc1`) writes assembly code to a local temporary file (e.g., `/tmp/ccw4T8Hh.s`) which is later read by the assembler (`as`). We note that such temporary files are named randomly in different Program Compilation instances. From the context of how data is manipulated, they, however, go through identical system operations, i.e., created by `gcc`, written by `cc1`, read by `as`, and deleted by `gcc`. Thus, we can reason that these files may share similar semantics despite different identifiers. This matches our intuitive way of labeling them as IPC medium between `cc1` and `as`. Our core idea is to uncover the semantics of system entities and relations from their contextual information in audit events, such as by analyzing the correlation of their presence in events. A general approach to extracting such contextual semantics is to employ *embedding models*. The objective is to map system entities and relations into an embedding space (i.e., numeric vector space), where the distances between vectors capture the semantic relationships.

Now that we can interpret the semantic information of audit events. The next step would be to identify audit events belonging to individual behaviors. Our second insight is based on our observation that *high-level behaviors, which are primarily centered around an intended goal of a user, can be reflected as a series of system operations applied on data objects.* For example, compiling program `a.c` with `gcc` intends to translate source code into executable machine code. We can decompose the translation procedure into three operations: (1) *Compiling:* compile `a.c` into an assembly language file, `a.s`, (2) *Assembling:* convert `a.s` into a relocatable object file, `a.o`, and (3) *Linking:* combine `a.o` with multiple object files into an executable file, `a.out`. Specifically, WATSON defines the behavior as an intended goal of the user, while the means to achieve it as a *behavior instance*. Each behavior instance is a sequence of operations that the user performs to achieve the goal. These operations can be further modeled as data transfers and the behavior instance as a sequence of such data transfers. For example, we can summarize the Program Compilation behavior above by following how data transfer from `a.c` to `a.s`, `a.o`, and finally `a.out`. Based on this observation, we clearly abstract behaviors as sequences of data transfers that operate on related data. Particularly in audit logs, WATSON identifies behavior instances by leveraging information flows among audit events as guidance for tracking data transfers. When such flows exist, we group events as a *subgraph* and form an individual behavior instance. For example, during program compilation, the compiling, assembling, and linking operations are recorded as `cc1`, `as`, and `ld` events in audit logs. We can identify this behavior by following information flows from the source file in the `cc1` event to the executable file in the `ld` event. Figure 2 presents subgraphs used by WATSON to summarize behaviors in the motivating example.

With the knowledge of events and identification of behaviors, the semantic representation of behaviors can be naturally thought of as the aggregated semantics of their constituent events as they are defined as sequences of events.

D. Threat Model

We assume the underlying OS, the auditing engine, and monitoring data to be part of the trusted computing base (TCB). Ensuring the integrity of the OS kernel, endpoint monitors, or audit logs themselves is beyond the scope of this work. This threat model is shared among related work on system auditing [29], [36], [38], [39], [53], [63].

We also assume that behaviors go through kernel-layer auditing, and their operations are captured as system-call audit logs. Although attackers may attempt to perform malicious behaviors without executing any system call to hide their footprints, such behaviors appear to be rare, and the harm they can bring to the rest of the system is limited [76]. We focus on behaviors in single user sessions in this paper. Our insights are generally applicable to cross-session or cross-machine behaviors.

III. WATSON DESIGN

A. Approach Overview

The overall approach of WATSON is shown in Figure 3. It consists of three primary phases: Knowledge Graph Construction, Behavior Abstraction, and Representative Behavior Identification. WATSON takes as inputs system audit data,

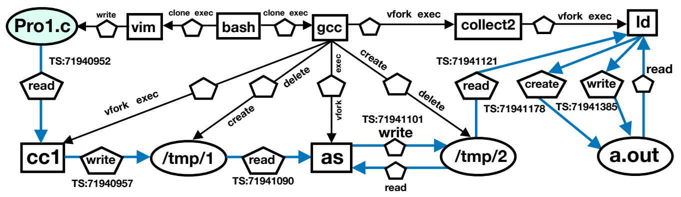


Fig. 4: Simplified version of KG for the Program Compilation. Pentagon denotes system relations. TS shows the timestamp attribute of *Relation* elements.

e.g., Linux Audit logs [9]. It summarizes behavior instances, uncovers their semantics, and finally outputs representative high-level behaviors.

Specifically, given audit logs in a user session as the input, the Knowledge Graph Construction module first parses logs into triples and constructs the log-based knowledge graph (KG). Then, the Event Semantics Inference module employs a translation-based embedding model to infer the contextual semantics of nodes in the KG. At the same time, the Behavior Summarization module enumerates subgraphs from the KG to summarize behavior instances. Combined with node semantics, the Behavior Semantics Aggregation module next enhances subgraphs to encode the semantics of behavior instances. Finally, the Behavior Clustering module groups semantically similar subgraphs into clusters, each specifying a high-level behavior. These cluster-based behavior abstractions can further be used to reduce the efforts of downstream tasks. We present the design details of WATSON in the following sections.

B. Knowledge Graph Construction

In order to analyze the contextual semantics of events, a unified representation is required to present heterogeneous events in a homogeneous manner. Instead of using a provenance graph based on the provenance data model (PROV-DM [12]), we propose a new representation based on knowledge graph (KG) to integrate heterogeneous information. This allows for the future capacity to capture relationships beyond just provenance [79], [80] (e.g., file meta-information such as permissions and owners).

Following the formal description of a KG [26] by Färber et al., we define log-based KG as a resource description framework (RDF) graph [64]. More concretely, the log-based KG is a set of numerous semantic triples. Each triple, corresponding to an audit event, consists of three elements that codify the semantic *Relation* between the *Head* and *Tail* in the form of (*Head, Relation, Tail*). Both *Head* and *Tail* can be any type of system entities, and *Relation* can take any system operation that is performed on *Tail*. However, note that types of system entities in triples are consistent with that in audit events, e.g., *Head* and *Tail* cannot be files or network sockets simultaneously. Figure 4 shows a knowledge sub-graph of the motivating example. Similar to a provenance graph, a KG encodes information flow that exists from *Head* to *Tail* in a triple. For example, the dependency chains (colored by blue) in Figure 4 illustrate the data transfers from `Pro1.c` to `a.out`.

C. Event Semantics Inference

Understanding the semantics of audit events is the first step in abstracting high-level behaviors. In particular, an accurate understanding is predicated on a suitable representation and granularity whereby semantic meanings can be effectively compared. A common practice in prior work [25], [71], [72] is to formulate each log event as a basic unit for analysis. However, a single audit event includes three elements (*Head*, *Relation*, and *Tail*), where each element separately contributes to event semantics. Therefore, performing semantic analysis on the level of elements compared to events provides a more detailed view as the context of individual elements is made explicit. Working on the level of elements, we can obtain the semantics of an audit event through the consolidation of three constituent elements and the semantics of a behavior instance through the consolidation of the events that define it. Although there exists a trade-off between scalability and accuracy due to different granularity of semantic analysis, the choice of a computationally efficient embedding model allows us to preserve precision while handling the large number of events found in logs. As such, we select individual elements rather than audit events as the base unit in our semantic reasoning.

Since embedding models can learn the semantics of audit events from their contextual information with elements as the basis, the next question is how to map elements into an embedding space (i.e., vector space). In Natural Language Processing, word embedding has been used with much success to extract and represent the semantics of words [16], [74]. Inspired by the success of word embeddings in NLP, EKLAVYA [23] showed how it could also be applied to infer the semantics of binary instructions based on their *usage context*. This prompted us to ask a similar question. Does the contextual occurrence of elements in an audit event relate to their semantics? Take for example the triples (`cc1, read, a.c`) and (`cc1, read, b.c`). While `a.c` and `b.c` belong to different events, the usage context of both elements in the presence of (`cc1 read`) provides the hint that they might share similar semantics as program source files. Intuitively, we aim to convert each element into a vector where a small distance (e.g., L1/L2-norm distance) between elements signifies similar semantics while a large distance the opposite. For instance, we expect the distance between embeddings (i.e., numeric vectors) of `a.c` and `b.c` to be small. To achieve this goal, we propose employing the translation-based embedding model, TransE [20], to learn the mapping from elements to the embedding space.

In TransE, the translation in the embedding space describes the semantic relationship between *Head* plus *Relation* and *Tail*. Specifically, the embedding space has the property that given a triple (*Head, Relation, Tail*), the position of *Tail* is that of *Head* with a translation by *Relation* (i.e., *Head* + *Relation* $\approx$ *Tail*). Our guiding principle in selecting TransE is its translation-based model perfectly matching our understanding of contextual semantics in audit events. For example, consider the case of (`cc1, read, a.c`) and (`cc1, read, b.c`). Since TransE updates the embeddings of both `a.c` and `b.c` using `cc1` + `read`, they will be nearby in the embedding space, indicating similar semantics. In theory, the embedding model in TransE mirrors our expectation of element semantics and their similarities, and Section V-B experimentally demonstrates that TransE indeed learns the contextual semantics of elements that matches our domain knowledge.

In the embedding process, each element is first initialized as a symbol without regard to its numeric label or textual name.

To do so, we encode elements as one-hot vectors and use them as training inputs to the embedding model. Each one-hot vector is an n-dimensional vector where n is the number of unique elements in the universal set. For example, if the entire log set contains two triples involving six elements, five of which are unique, then the elements would be encoded into a 5-dimensional vector with the first being $[1,0,0,0,0]$, and the last being $[0,0,0,0,1]$. In terms of identifiers for elements, we use the executable name, argument, and PID for a process element, the absolute path for a file element, and the IP address and port number for a socket element. We do not directly employ element labels (e.g., PID and inode) for identification because they are recycled in a system and can easily cause a collision. For a *Relation* element, we use the system call number as the identifier due to its uniqueness. Note that element identifier bears no relation to domain-specific semantics during embedding.

In the training phase, TransE optimizes the embedding space of elements by minimizing the translational distance of a triple found in the KG (training triple) while maximizing that of a triple not found in the KG (corrupted triple). We generate corrupted triples by replacing either *Head* or *Tail* in a training triple with a random element and ensuring that the new triples do not exist in the KG. The loss function for the embedding model optimization is summarized as follows:

$$L = \sum_{(h,r,t)\in \boldsymbol{KG}} \sum_{(h',r',t')\notin \boldsymbol{KG}} (\|\boldsymbol{e_h}+\boldsymbol{e_r}-\boldsymbol{e_t}\| - \|\boldsymbol{e_{h'}}+\boldsymbol{e_{r'}}-\boldsymbol{e_{t'}}\| + \gamma),$$

where $\|\cdot\|$ denotes the L1-norm distance function. h, r, and t represent *Head*, *Relation*, and *Tail* elements. e_x denotes the embedding of element x. Note that for a given element, its embedding is constant no matter it acts as *Head* or *Tail* in a triple. Moreover, TransE uses Margin γ to encourage discrimination between training and corrupted triples. We refer interested readers to [20] for the detailed optimization procedure.

In summary, the result of TransE is an $n \times m$ embedding matrix, which maps n-dimensional one-hot encoded elements into an m-dimensional embedding space. To further infer the semantics of an audit event, we concatenate the embeddings of its constituent elements (*Head, Relation*, and *Tail*) and generate a $3m$-dimensional vector (192 dimensions in our case).

D. Behavior Summarization

The next step of behavior abstraction is identifying behavior instances from one user login session. We define a behavior instance as a sequence of audit events operated on related data and correlated by information flows. Accordingly, the problem of summarizing individual behavior instances can be reduced to extracting causally connected subgraphs with data objects (i.e., file and network socket) as the root in the session's KG. Note that unlike path-based approaches [36], [77], which decompose a provenance graph into overlapping paths for analysis, we partition the KG on the basis of subgraphs to represent behavior instances. This is because an individual path cannot preserve the complete context of behaviors representing multi-branch data transfers. For example, path-based approaches would fail to correlate all system operations belonging to the Data Exfiltration behavior in our motivating example because operations of the program compilation and github upload are located in separate paths.

In order to extract subgraphs that summarize behavior instances, we perform an adapted forward depth-first search (DFS) on the session's KG rooted at data objects. Figure 2 demonstrates three resulting subgraphs of behavior summarization in the motivating example. During graph traversal, we enforce the constraint that the timestamp of each following edge has to be monotonically increasing from all previous edges. This design can prevent false dependencies due to information flowing from a future event to a past event. Besides, we note that the ancestry of a system entity usually contains critical behavior contexts. For example, the process creating root data objects describes where they come from (e.g., downloaded by email clients). However, such ancestries are lost in the plain forward DFS because they belong to backward dependencies. Therefore, we further incorporate one-hop incoming edges of reached system entities during graph traversal. In addition, we do not bound the DFS based on the level of depth but rather domain-specific system entities (e.g., files read and written by numerous processes [39]). As only coarse-grained causal dependencies (system calls) are recorded in audit logs, the causality analysis suffers from the dependency explosion problem [49]. This also has an adverse effect on WATSON's ability to track data transfers and summarize behavior instances. While solving the general problem of dependency explosion is not within the scope of this work, we aim to mitigate its influence by applying heuristics to specify system entities (e.g., `.bash_history` and `firefox`) that potentially trigger dependency explosion as the termination condition in our DFS. To guarantee no behavior instance loss, we perform the adapted DFS on every single data object found in the KG except libraries that do not reflect the roots of user intended goals. Two behaviors are further merged if one behavior is the subset of the other.

In summary, we apply an adapted DFS algorithm to partition the session's KG into subgraphs, where each describes a behavior instance.

E. Behavior Semantics Aggregation

After behavior instance summarization, we next extract the semantics of behavior instances. Recall that each behavior instance partition is composed of audit events whose semantics has been represented with high-dimensional vectors using the embedding matrix. We then naturally derive the semantics of behavior instances by combining behavior instance partitions and the embedding matrix.

To obtain the semantic representation of a behavior instance, a naïve approach is to add up the individual vectors of its constituent events. However, this approach only works under the assumption that all constituent events contribute equally to the behavior instance semantics. In practice, this assumption usually does not hold due to how events have different *relative importance* to reflect behavior semantics and the influence of *noisy events*.

Relative Event Importance. Any given high-level task which a user performs consists of multiple smaller operations, but the importance or necessity of each operation may not be the same. While completing the desired task, users are typically required to execute boilerplate operations. Consider the compilation of a program in Figure 4. Users usually do not directly launch `gcc` to compile source code but first locate the source file using common utilities like `ls` and `dir`. Although such boilerplate operations reflect user activities, they do not uniquely associate with high-level behavior. Therefore, concerning the task of behavior abstraction, these boilerplate operations should be given less attention than operations directly representing behavior.

The key question is, how can we automatically ascribe the relative importance of each operation? Our insight stems from how operations are described in audit events of user sessions. We observe that behavior-unrelated events are more prevalent in sessions as they are repeated across different behaviors, whereas actual behavior-related events happen less frequently. Based on this observation, we propose using the frequency of events as a metric to define their importance. More specifically, we employ the Inverse Document Frequency (IDF) to determine the importance of a particular event to the overall behavior. IDF is widely used in information retrieval as a term weighting technique [59]. Its principle is to give more discriminative capacity to less-common terms in documents. In our particular scenario, each audit event and user session are viewed as a term and document. The equation to measure the IDF value is as follows:

$$w_{IDF}(e) = \log\left(\frac{S}{S_e}\right),$$

where e denotes an audit event, and S and S_e are the numbers of all the sessions and specific sessions that include event e. To summarize, each event in a behavior partition is assigned a weight using IDF, representing its importance to the behavior.

Noisy Events. The low-level and verbose nature of audit logs makes the presence of noisy events one of the primary challenges analysts struggle with. Reducing noise occurrence helps to improve WATSON's effectiveness. In this section, we will discuss two types of noisy events, *redundant* events and *mundane* events.

(1) *Redundant events*. In behavior instances, there are events when removed that do not change the data transfers. To identify these redundant events, we built on top of the shadow event [82], a concept that refers to file operations whose causalities have already been represented by other key events. We also incorporate domain knowledge in [50], [82] to enhance redundant event reduction. At a high level, analysts have listed specific files that do not introduce explicit information flows in causality analysis. For example, many processes create temporary files to store intermediate results during execution. Because such files exclusively interact with a single process during their lifetime, they do not affect data transfers nor contribute to behavior abstraction. Note that `/tmp/1` and `/tmp/2` in Figure 4 serving as IPC are not categorized as temporary files. All *redundant events* are considered noise and removed from behavior partitions.

(2) *Mundane events*. Another source of noise comes from file operations that are regularly performed for an action. We call them mundane events. Examples of such events are (`vim, write, .viminfo`) for file editing history cache and (`bash, read, /etc/profile`) for shell program setup. We classify mundane events as noisy events because they are associated with system routines rather than specific behaviors. Typically, mundane events have two characteristics, (a) they always occur for a given action, and (b) the order of their occurrence is fixed. In order to identify and filter them, we first enumerate all possible actions a program can perform. By one action, we refer to a sequence of events a program generates during its execution lifecycle in the system. Then, given sequences of events for each program, we summarize events always occurring in a fixed pattern as mundane events. Essentially, we formulate the mundane event identification problem as the longest common subsequence (LCS) searching problem. Given event sequences of a program, we extract the LCS among them as the mundane events. Similar to (1), all mundane events are removed from behavior partitions. We note that NODEMERGE [75] first proposes to identify mundane events (i.e., data access patterns) for data reduction. However, it focuses on file reading operations (e.g., load libraries and retrieve configurations) in the process initialization action. In contrast, WATSON targets all types of file operations in more general actions (e.g., `vim` creates and writes `.viminfo` to cache where users leave off editing files).

After weighting events with IDF and removing noisy events, we derive the semantic representation of behavior instances by pooling its constituent vectorized events. We have attempted different pooling approaches for implementation, such as addition, bi-interaction, and global average pooling. The addition pooling is eventually utilized as we observe that simply summing the semantics of events has already integrated the semantic information of events effectively.

In conclusion, the Behavior Abstraction phase takes a log-based KG as input and generates vector representation of behavior instances in a $3m$-dimensional embedding space.

F. Behavior Clustering

As described in Section II-C, behavior instances are variations on how high-level behaviors can be realized. In other words, a behavior can be thought of as one cluster of similar instances. It naturally follows that the behavior signature is the most representative instance (e.g., centroid) in the cluster. In this way, analysts only need to investigate a few auto-selected signatures for behavior matching rather than the whole cluster space. Given the vector representation of behavior instances, WATSON calculates their semantic relationships using cosine similarity as follows:

$$\mathrm{S}\left(F_m, F_n\right) = \frac{F_m \cdot F_n}{\|F_m\| \times \|F_n\|} = \frac{\sum_{e_i \in \boldsymbol{F_m}} \sum_{e_j \in \boldsymbol{F_n}} e_i \cdot e_j}{\sqrt{\sum_{e_i \in \boldsymbol{F_m}} (e_i)^2} \times \sqrt{\sum_{e_j \in \boldsymbol{F_n}} (e_j)^2}},$$

where F_m and F_n refer to the vector representation of two behavior instances, and $S(F_m, F_n)$ representing the cosine similarity score is positively correlated with behavior semantic similarity. This equation intuitively explains the effectiveness of using addition to pool embeddings of events in a behavior instance. Since F_i and F_j are the summations of their respective constituent events, cosine similarity, in effect, compares the similarities of individual events in two instances.

WATSON uses Agglomerative Hierarchical Clustering Analysis (HCA) algorithm to cluster similar behavior instances. Initially, each behavior instance belongs to its own cluster. HCA then iteratively calculates cosine similarities between clusters and combines two closest clusters until the maximum similarity is below the merge threshold (0.85 in our case). We select Centroid Linkage as the criterion to determine the similarity between clusters. In other words, cluster similarity estimation depends on centroids (arithmetic mean position) in clusters.

Once the clusters are identified, a behavior signature for each cluster would be extracted based on the instances' representativeness. WATSON quantifies the representativeness of each instance in a cluster by computing its average similarity with the rest of instances. The instance with the maximum similarity is picked out as the signature. By distinguishing the behavior signature, we expect to see a substantial analysis workload reduction as semantically similar behaviors have been clustered before human inspection. Take for instance our motivating example. WATSON groups multiple Program Compilation and Upload instances together into one cluster and only reports the representative for analyst investigation. It is noteworthy that the Data Exfiltration behavior fails to compile the illegitimate source code and thus loses the data transfers from the source file to an executable file. As a result, it is not clustered together with the Program Compilation behavior even though they use identical utilities (`gcc` to compile programs and `git` to upload files).

In summary, the Representative Behavior Identification phase clusters similar behavior instances and distinguishes representative behaviors as signatures.

IV. IMPLEMENTATION

We prototype WATSON in 9.2K lines of C++ code and 1.5K lines of Python code. In this section, we discuss important technical details in the implementation.

Log Input Interface. WATSON takes system audit data as inputs. We define a common interface for audit logs and build input drivers to support different log formats, such as Linux Audit [9] formats (auditd [8], auditbeat [7], and DARPA dataset[2]). Our drivers can be extended to support other audit sources, i.e., CamFlow [68] and LPM [17] for Linux, ETW for Windows, and Dtrace for FreeBSD.

Modular Design. Modularity is one of our guiding principles in WATSON design. Each module can be freely swapped out for more effective solutions or application-specific trade-offs in terms of performance vs. accuracy. Take the Event Semantics Inference module for example. In our implementation, TransE is used to learn the embedding space of audit events for its memory and time efficiency despite the limitation on the types of relations it can encode [81]. If WATSON users wish, TransE can be easily replaced with other embedding algorithms (e.g., TransR [51]) without affecting WATSON's functionality.

Knowledge Graph Construction. To construct a log-based KG, WATSON first sorts audit events in chronological order. Then, it translates each event into a KG-based triple by using the system entities as the *Head* and *Tail*, and the system call function as the *Relation*. To interpret rules for triple translation, we manually analyze 32 commonly-used system calls, including (1) process operations such as *clone*, *execute*, *kill*, and *pipe*; (2) file operations such as *read*, *write*, *rename*, and *unlink*; (3) socket operations such as *socket*, *connect*, *send*, and *receive*. After parsing audit events into triples, a relational database (PostgreSQL [11]) is used to store the built KG. Note that we compute the 64-bit hash value of system entity identifiers defined in Section III-C as the primary key in the database. Besides, all the properties of system entities (e.g., process name) and relations (e.g., timestamps) are preserved as attributes of elements in triples.

Parameter Settings. We implement the embedding model in the Event Semantics Inference module with Google Tensorflow [15]. The model is optimized with SGD optimizer, where the margin, batch size, and epochs are fixed at 1, 1024, and 500, respectively. In terms of hyperparameter, we apply a grid search: the learning rate and embedding size are tuned amongst {0.005, 0.010, 0.015} and {32, 64, 128, 256}. Similarly, for behavior clustering, the merge threshold is searched in {0.7, 0.75, 0.80, 0.85, 0.9}. In light of the best F1 score in our experiments, we show the results in a setting with learning rate as 0.010, embedding size as 64, and merge threshold as 0.85. Note that the merge threshold is a configurable parameter, and analysts can customize it according to particular scenarios. For example, the threshold can be decreased to satisfy high true positive or increased to maintain low false positive.

Behavior Database. We observe that behaviors are recurrent in the system. New sessions always include behaviors that appear previously. Therefore, to avoid repetitive behavior analysis, we label WATSON-generated behavior signatures with domain-specific descriptions and store them in our database. The embedding of each behavior quantifying semantics is preserved as an attribute for behavior objects. Once a new behavior instance appears, WATSON first computes its cosine similarities with all stored signatures. If no similarity is above the merge threshold, analysts manually investigate its semantics; Otherwise, its semantics is retrieved by querying the similar behavior signature in the database.

V. EVALUATION

In this section, we employ four datasets and experimentally evaluate four aspects of WATSON: 1) the explicability of inferred event semantics; 2) the accuracy of behavior abstraction; 3) the overall experience and manual workload reduction in attack investigation; and 4) the performance overhead.

A. *Experimental Dataset*

We evaluate WATSON on four datasets: a benign dataset, a malicious dataset, a background dataset, and the DARPA TRACE dataset. The first three datasets are collected from `ssh` sessions on five enterprise servers running Ubuntu 16.04 (64-bit). The last dataset is collected on a network of hosts running Ubuntu 14.04 (64-bit). The audit log source is Linux Audit [9].

In the benign dataset, four users independently complete seven daily tasks, as described in Table I. Each user performs a task 150 times in 150 sessions. In total, we collect 17 (expected to be $4 \times 7 = 28$) classes of benign behaviors because different users may conduct the same operations to accomplish tasks. Note that there are user-specific artifacts, like launched commands, between each time the task is performed. For our

[2]To achieve platform independence, audit logs in DARPA datasets are represented in a Common Data Model (CDM) format [43].

TABLE I: Overview of tasks with scenario descriptions. Column 3 shows the ground-truth behaviors when completing the tasks.

Task	Scenario Description	Behavior			
		U1	U2	U3	U4
Program Submission	Upload machine learning programs to a GPU server	vim + scp	vi + scp	emacs+ scp	nano + scp
Code Reference	Download and compile online programs for reference	wget + gcc	elinks + gcc	wget + python	elinks + python
Dataset Download	Download and uncompress public datasets	wget + gzip	wget + bzip	elinks + unzip	wget + bzip
Program Compilation	Write a C/C++ program and testify its functionalities	vim + gcc	vim + g++	vim + gcc	vim + gcc
FTP Server Login	Use ssh or ftp services to sign in a FTP server	ftp	ssh	ftp	ftp
Package Installation	Run apt application to install software packages	apt update/install	apt update/install	apt update/install	apt update/install
Package Deletion	Run apt application to remove software packages	apt purge	apt purge	apt purge	apt purge

TABLE II: Overview of attack cases in our malicious dataset with scenario descriptions.

Attack Cases	Scenario Description	Reference
Data Theft	A malicious script is mistakenly downloaded by a normal user and when being executed it exfiltrates sensitive information on the local machines	PRIOTRACKER [53]
Illegal Storage	An attacker creates a directory in another user's home directory and uses it to store illegal files	Taser [30]
Content Destruction	An insider tampers with classified programs and documents	Taser [30]
Backdoor Installation	An attacker compromises a FTP server, invokes a remote bash shell, and installs a backdoor program to obtain permanent access to the server	Protracer [57]
Passwd-gzip-scp	An attacker steals user account information from /etc/passwd file, compresses it using gzip and transfers the data to a remote machine using ssh service	Xu et al. [82]
Wget-gcc	Malicious source code is downloaded, compiled and executed	Xu et al. [82]
Configuration Leakage	A downloaded txt file exploits the code executable vulnerability in vim to collect machine configuration for future compromise preparation	CVE-2019-12735 [2]
Passwd Reuse	An administrator reads encrypted user password from /etc/shadow file, decodes it with John, and uses the plaintext to log in on other applications	Passwd Reuse [10]

benign dataset, there are 55,296,982 audit events, which make up 4,200 benign sessions.

In the malicious dataset, following the procedure found in previous works [2], [10], [30], [53], [57], [82], we simulate[3] eight attacks from real-world scenarios as shown in Table II. Each attack is carefully performed ten times by two security engineers on the enterprise servers. In order to incorporate the impact of typical noisy enterprise environments [53], [57], we continuously execute extensive ordinary user behaviors and underlying system activities in parallel to the attacks. For our malicious dataset, there are 37,229,686 audit events, which make up 80 malicious sessions.

In the background dataset, we record behaviors of developers and administrators on the servers for two weeks. To ensure the correctness of evaluation, we manually analyze these sessions and only incorporate sessions without behaviors in Table I and Table II into the dataset. For our background dataset, there are 183,336,624 audit events, which make up 1,000 background sessions.

The DARPA TRACE dataset [13] is a publicly available APT attack dataset released by the TRACE team in the DARPA Transparent Computing (TC) program [4]. The dataset was derived from a network of hosts during a two-week-long red-team vs. blue-team adversarial Engagement 3 in April 2018. In the engagement, an enterprise is simulated with different security-critical services such as a web server, an SSH server, an email server, and an SMB server [63]. The red team carries out a series of nation-state and common attacks on the target hosts while simultaneously performing benign behaviors, such as ssh login, web browsing, and email checking. For the DARPA TRACE dataset, there are 726,072,596 audit events, which make up 211 graphs. Note that we analyze only events that match our rules for triple translation in Section IV.

[3]For Illegal Storage and Content Destruction attacks, we leverage different vulnerabilities (CVE-2019-13272 [3] and CVE-2019-12181 [1]) to implement privilege escalation because the prior exploits are out-of-date.

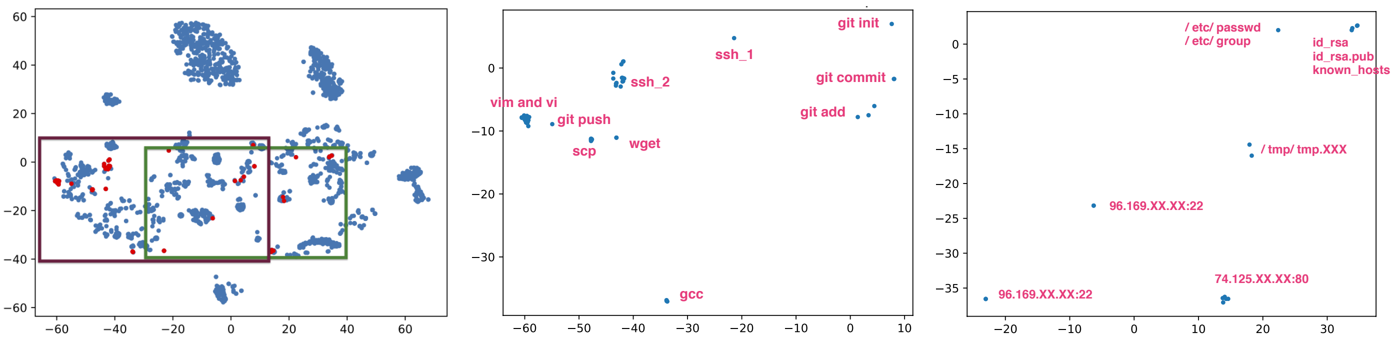
We test WATSON's explicability and accuracy on our first three datasets as we need the precise ground truth of the event semantics and high-level (both benign and malicious) behaviors for verification. We further explore WATSON's efficacy in attack investigation against our malicious dataset and DARPA TRACE dataset because the ground truth of malicious behaviors related to attack cases is available to us.

In general, our experimental behaviors for abstraction are comprehensive as compared to behaviors in real-world systems. Particularly, the benign behaviors are designed based upon basic system activities [84] claimed to have drawn attention in cybersecurity study; the malicious behaviors are either selected from typical attack scenarios in previous work or generated by a red team with expertise in instrumenting and collecting data for attack investigation.

B. Explicability of Event Semantics Inference

We measure what semantics WATSON learns for audit events both visually and quantitatively: Visually, we use t-SNE to project the embedding space into a 2D-plane giving us an intuition of the embedding distribution; quantitatively, for each triple, we compare the training loss in the TransE model against our knowledge of event semantics and their similarities.

Embedding of system entities. Each element's semantics in events is represented as a 64-dimensional vector, where the spatial distance between vectors encodes their semantic similarity. To visualize the distance, we apply t-SNE to project high-dimensional embedding space into a two-dimensional (2D) space while largely preserving structural information



(a) (b) (c)

Fig. 5: t-SNE visualization: (a) shows the embedding space of 20 sampled sessions. Each dot denotes a system entity or relation. The red (left) and green (right) boxes are where (b) and (c) locate; (b) and (c) point out 53 process entities and 25 data entities.

TABLE III: The semantic distance ("surprise" factor) in six audit events (KG-based triples). B and M indicate whether an event is benign or malicious.

Head	Relation	Tail	Distance $\|h + r - t\|$	B/M
bash	execve	vim	0.4152	B
/usr/include/stdio.h	read	cc1	0.4273	B
wget	send	172.26.X.X	1.6401	B
.ssh/id_rsa	read	ssh	0.8078	B
vim	execve	bash	6.2447	M
bash	send	172.26.X.X	4.2952	M

among vectors. To manage the complexity, we randomly sampled 20 sessions from the first three datasets for visualization, obtaining a scatter plot of 2,142 points. Figure 5a shows the 2D visualization of the embedding space. Points in the space are distributed in clusters, suggesting that events are indeed grouped based on some metric of similarity.

We further select eight programs (`git`, `scp`, `gcc`, `ssh`, `scp`, `vim`, `vi`, and `wget`) to investigate process element embeddings. Figure 5b shows a zoom in view of Figure 5a containing 53 elements corresponding to the eight programs. For clarity, the elements are labeled with process names and partial arguments. Note that identity information is erased during one-hot encoding and thus does not contribute to semantics inference. While most elements of the same program are clustered together, there are a few interesting cases supporting the hypothesis that the embeddings are actually semantic-based. For example, `git` has a few subcommands `(push, add, commit)`. `Git push` is mapped closer to `scp` and `wget` instead of `git commit` and `git add`. This agrees with the high-level behaviors where `git push` uploads a local file to a remote repository while `git add` and `git commit` manipulate files locally. Another interesting example involves `ssh` where two different clusters can be identified, but both represent `ssh` connection to a remote host. Upon closer inspection, we notice that these two clusters, in effect, correspond to the usage of `ssh` with and without X-forwarding, reflecting the difference in semantics.

Similarly, Figure 5c shows the t-SNE plot of 25 data object elements. As expected, socket connections are separated from files and clustered through different network ports. Note that the relatively large distance between socket elements of port 22 is an artifact of the projection by t-SNE. They are actually close in the pre-projected embedding space as measured using cosine similarity. Moreover, WATSON groups `id_rsa`, `id_rsa.pub`, and `known_hosts` together, suggesting that they share similar functions as identity authentication. However, `id_rsa` and `id_rsa.pub` have different semantics as they represent private and public key, respectively. TransE does not identify their differences because both are only accessed by `ssh` during SSH Login. One potential approach to distinguishing them is to involve additional information such as file permissions for event semantics inference.

Event Semantics. We calculate the translational distance in KG-based triples to quantify the event semantics learned by WATSON. This distance can be intuitively thought of as a measure of how "surprising" one event is, with a lower value being commonly seen and a higher value being rarely seen. Some existing approaches [36], [77] leveraged the rareness of events to measure their anomaly scores. To explore whether our event semantics is consistent with such heuristics, we also consider whether events are benign or malicious while analyzing semantic distances. We classify one event as malicious if it plays a critical role in an attack campaign. This experiment is performed on 1,000 random events in our malicious dataset. The statistical result shows that the distances in malicious events are always larger than those in benign events. This matches our domain knowledge as malicious events should be considered "surprising" for analysts. Due to space limitations, we present six events as examples in Table III. Specifically, (`vim, execve, bash`) and (`bash, send, 172.26.X.X`) in the Configuration Leakage behavior result in much higher distance than other benign events.

In summary, WATSON learns semantics that consistently mirrors our intuitive understanding of event contexts.

C. Accuracy of Behavior Abstraction

To evaluate WATSON's accuracy in behavior abstraction, we use behavior signatures that WATSON learns to predict `ssh` sessions in our first three datasets with similar behaviors. To this end, we select 25 behaviors, including 17 benign routines in Table I and eight malicious attacks in Table II, as the abstraction candidates. Due to considerable background noise, audit events regarding these behavior candidates only constitute 0.2% of the entire log volume. For each behavior

TABLE IV: Evaluation results for behavior abstraction in sessions. Columns 2 reports the number of events in raw audit logs (AL), KG (after noise reduction), and behavior signatures (BS). Column 3-5 elaborate the abstraction accuracy of WATSON on 25 different behaviors in 5,280 sessions. -U# in column 1 denotes specific users in Table I.

Behavior	AL	KG	BS	Recall	Precision	F1
Program Submission-U1	3256	101	13	92.7%	49.7%	64.7%
Program Submission-U2	3178	98	13	94.0%	50.3%	65.5%
Program Submission-U3	3854	117	28	88.0%	94.3%	91.0%
Program Submission-U4	3208	97	10	89.3%	95.7%	92.4%
Code Reference-U1	3336	131	24	96.7%	91.8%	94.2%
Code Reference-U2	4274	141	27	82.7%	84.3%	83.5%
Code Reference-U3	2915	92	6	93.3%	95.2%	94.2%
Code Reference-U4	3141	105	10	92.0%	93.1%	92.5%
Dataset Download-U1	3361	111	10	98.7%	99.3%	99.0%
Dataset Download-U2	3471	121	10	95.3%	96.6%	95.9%
Dataset Download-U3	3635	116	13	92.7%	97.2%	94.9%
Program Compilation-U1	3229	137	24	96.7%	98.6%	97.6%
Program Compilation-U2	3962	210	103	96.0%	93.2%	94.6%
FTP Server Login-U1	3126	97	5	100%	100%	100%
FTP Server Login-U2	3086	101	8	100%	100%	100%
Package Installation-U1	55610	1243	480	95.3%	97.9%	96.6%
Package Deletion-U1	19595	312	165	90.6%	98.6%	87.5%
Data Theft	409949	9013	26	100%	100%	100%
Illegal Storage	203994	6906	41	100%	100%	100%
Content Destruction	550281	17524	33	100%	100%	100%
Backdoor Installation	310783	40417	75	90.0%	100%	94.7%
Passwd-gzip-scp	280451	5906	13	100%	83.3%	90.1%
Wget-gcc	357449	15928	39	90.0%	100%	94.7%
Configuration Leakage	1144963	101225	116	80.0%	100%	88.9%
Passwd-Reuse	465098	31834	9	100%	100%	100%
Average	153968	9283	52	94.2%	92.8%	92.8%

candidate, we first randomly select one session of it and then generate the corresponding behavior signature. Next, we use the signature to predict similar behaviors in the remaining 5,279 sessions. Recall in Section III-F that two behaviors are considered similar if their cosine similarity is beyond the merge threshold (0.85). The performance metrics are measured using Recall, Precision, and F1 scores. Intuitively, they provide a measure of true-positive rate, false-positive rate, and general accuracy, respectively. We refer true positives to sessions correctly predicted with the behavior candidate and false positives to sessions incorrectly predicted with the behavior candidate.

Table IV provides a summary of the experimental results. WATSON demonstrates promising results (an average F1 score of 92.8%) in behavior abstraction. Even for the complicated behavior, the Code Reference-U2, WATSON still accomplishes an F1 score of 83.5%. This is because by leveraging contextual information of audit events, WATSON can reason the semantics of behavior instances accurately. For example, when compiling programs with `gcc`, child processes like `cc1` and `as` will create, read, and write temporary files for IPC. Such temporary files are first initialized randomly in the embedding space. Through their contextual relationships with `cc1` and `as`, WATSON infers that they share similar semantics as the IPC medium between `cc1` and `as`, and thus enhances their proximity in the embedding space. Consequently, WATSON boosts the similarity of Program Compilation instances, although they access different temporary files at first glance.

Another interesting observation is that the precision is, in most cases (20/25 in Table IV), higher than or equal to the recall. The relatively low recall shows that the semantics of behavior instances can be affected by noisy events even after using IDF weighting and noise removal. That said, the high precision indicates that the impact caused by such noise is limited. In particular, we discover that behavior instances suffering from noisy events can drift away from the original cluster. However, they will most likely form a new cluster as false negatives instead of joining the existing one as false positives. Furthermore, the average precision (92.8%) is lower than the average recall (94.2%), although the precision is, in most cases, higher than the recall. This primarily results from two exceptional cases, the Program Submission-U1 and Program Submission-U2 (PS1 and PS2 for short). PS1 and PS2 significantly pull down the average precision because they are clustered together and continuously recognized as false positives to each other. In fact, it is reasonable to predict PS2 given PS1 or vice versa as their only difference is using `vim` or `vi` to edit documents. Consequently, WATSON still achieves behavior abstraction with low false positives.

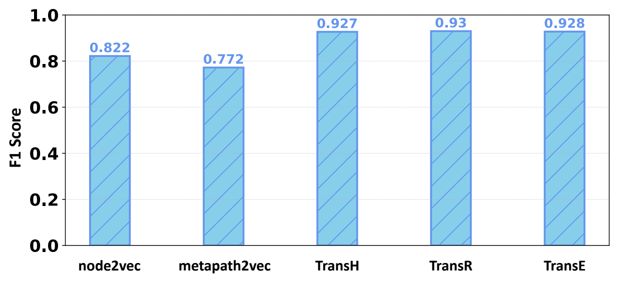
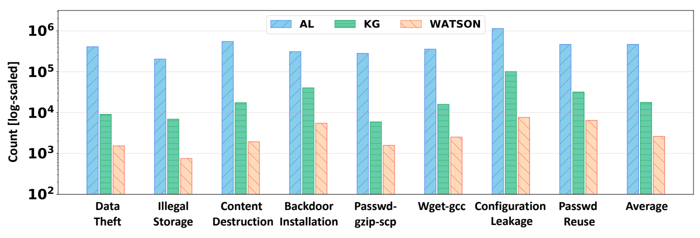


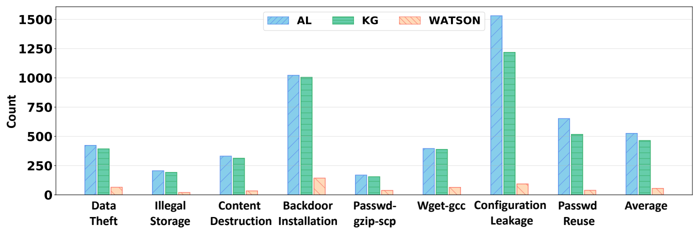
Fig. 6: F1 scores of five different embedding methods in behavior abstraction.

Next, the abstraction accuracy has no direct relationship with the size of behavior signatures. For example, although the Package Installation produces the largest signature, nearly orders of magnitude larger than others, they manage to achieve an F1 score of 96.6%, surpassing most cases. However, the Package Deletion also generates a relatively large size signature but achieves one of the lowest accuracy (87.5%), which even falls far behind the average. A plausible explanation is that WATSON recognizes behaviors by representative patterns, which do not necessarily coincide with spatial scale. The Apt Update and Install using `gpgv` to authenticate keys is a unique behavioral pattern. On the other hand, the Apt Purge uses `update-motd` to examine the number of packets needed for updates. However, this pattern is also in the Apt Update and Install, which lowers its distinguishability. Moreover, the abstraction accuracy does not rely on task types either. For example, the Code Reference task simultaneously has 94.2% and 83.5% accuracy for the Code Reference-U1 and Code Reference-U2 behaviors. This suggests that it is system operations that decide the recognizability of behaviors instead of the task itself.

Finally, we observe that WATSON exhibits high accuracy at categorizing malicious behaviors. For all eight attack scenarios, WATSON can achieve, on average, 95.0% and 97.9% in terms of recall and precision. In other words, 4 out of 80 malicious sessions are missed, and 2 out of 5,200 benign sessions are falsely predicted with malicious behaviors. Analysts can further improve the recall and finally detect all 80 malicious sessions by decreasing the merge threshold in HCA. Although malicious behaviors can be seamlessly blended in background activities by performing daily routines, most of them, if not all,



(a)



(b)

Fig. 7: Statistics of analysis workload (a) and behaviors (b) in our malicious dataset under raw audit logs (AL), KG (noise removed) and behavior abstraction (WATSON). Each attack case corresponds to ten sessions, and we report the average results.

possess distinct operations. Take for example the Configuration Leakage and Backdoor Installation attacks. It is not an everyday occurrence that `vim` collects system configurations and sends them out remotely, or `ProFTPD` invokes a privileged bash and downloads executable files. Accordingly, WATSON can recognize malicious behaviors and separate them from benign ones.

D. Comparison of Different Embedding Methods

To demonstrate the effectiveness in behavior semantics inference, we compare TransE with four other embedding methods, namely node2vec [31], metapath2vec [24], TransH [81], and TransR [51]. Before training these embedding models, we have transformed our encodings of audit events to their accepted input formats. For example, we define three types of meta paths (i.e., file to process to file, socket to process to socket, and process to process to process) for random walks in the metapath2vec model.

- **node2vec**: This method defines the context of a node in the graph based on its local neighborhood. It follows the intuition that nodes from the same network community should be mapped closely together in an embedding space.
- **metapath2vec**: This method treats meta-path based random walks in a graph as natural language sentences in a corpus. It then feeds these sentences into a skip-gram model to learn node embeddings.
- **TransH**: To address the issue of TransE when modeling relations that translate one entity to various entities (i.e., 1-to-N problem, and similarly, N-to-1 and N-to-N problems), this method extends TransE by introducing an additional hyper-plane to learn relation embeddings.
- **TransR**: Unlike TransE and TransH, which assume that entity and relation embeddings are within a shared space, this method builds entity and relation in separate spaces. To train embeddings, it first projects entities from the entity space to relation space and then calculates translations among projected entities.

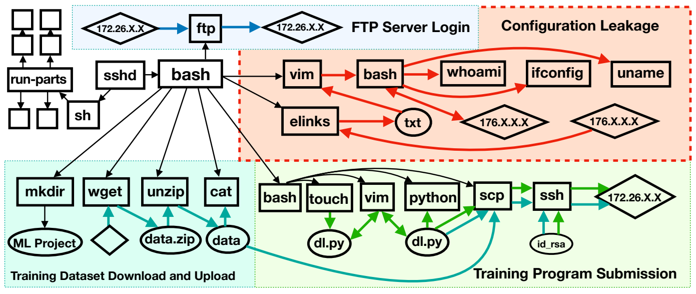
Figure 6 summarizes the behavior abstraction accuracy of different embedding methods on our first three datasets. Translation-based embedding methods (i.e., TransE, TransH, and TransR) consistently outperform the node2vec and metapath2vec, which well justifies our design choice of using a translation-based model to infer contextual semantics. Specifically, the node2vec learns the semantics of elements based on the behaviors they belong to. Its principle is that elements from the same behavior share similar roles and thus should have similar embeddings. However, for audit data, system entity elements of a behavior are not necessarily similar. For example, in the Program Compilation behavior, `vim` and `gcc` are semantically irrelevant to each other. Moreover, the metapath2vec leverages meta-path based random walks to generate heterogeneous neighborhoods for different types of elements (e.g., processes and files). The downside is that it does not consider relation elements when training the embedding space for system entity elements. However, we note that relations are critical to infer system entity semantics. For example, (`bash, read, /etc/passwd`) and (`bash, delete, /etc/passwd`) indicate completely different semantics for the `bash` elements. As such, we hypothesize that this explains why node2vec and metapath2vec are unable to achieve a comparable abstraction accuracy as compared to translation-based methods.

Within three translation-based embedding methods, TransR slightly outperforms TransE and TransH through separating the entity and relation embedding space. Notwithstanding, it incurs a much larger runtime overhead. In our experiment, the model training time of TransE, TransH, and TransR are 2.13, 3.47, and 5.70 hours, respectively. Different translation-based methods demonstrate a trade-off between computational efficiency and predictive accuracy. The fact that TransE and TransR achieve almost the same accuracy, but TransE is around three times faster suggests that TransE is more scalable for long-term log analysis.

E. Efficacy in Attack Investigation

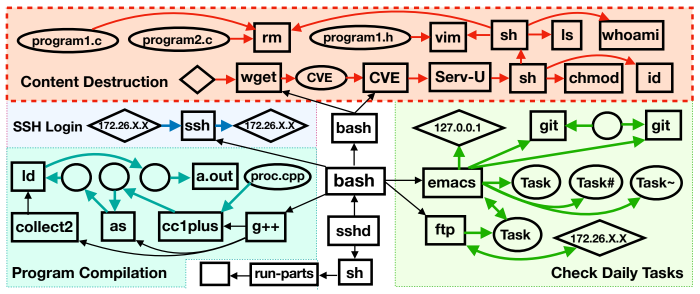
We explore WATSON's efficacy by empirically measuring the reduction of analysis workload in attack investigation. In this paper, analysis workload is quantified as the number of events an analyst would have to go through to identify all behaviors in a session. More specifically, events of an investigation before and after using WATSON refer to raw audit logs and the sum of events for each behavior signature identified. Although analysts do not necessarily search through all related events to recognize a behavior, this metric provides a reasonable way to demonstrate the proportion of reduced events due to behavior abstraction. We evaluate WATSON against our simulated malicious dataset, which includes eight attack scenarios in Table II, as well as the DARPA TRACE dataset, which includes five real-life APT attacks.

1) Evaluation on malicious dataset: Figure 7b summarizes the resulting analysis workload reduction in our malicious



FTP Server Login
Configuration Leakage
Training Dataset Download and Upload
Training Program Submission

(a)



(b)

Fig. 8: Case studies: (a) Configuration Leakage. (b) Content Destruction.

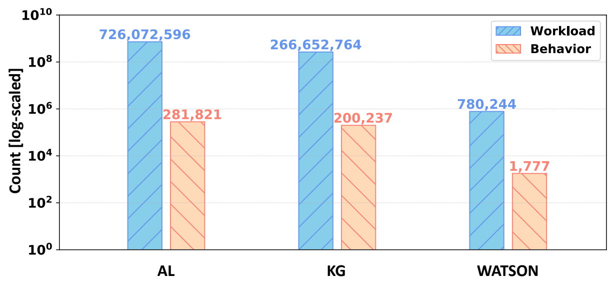


Fig. 9: Statistics of analysis workload and behaviors in DARPA TRACE dataset under raw audit logs (AL), KG (noise removed) and behavior abstraction (WATSON).

dataset. As a comparison, we also present intermediate workload reduction by removing noisy events defined in Section III-E. Combining the results in Table IV with Figure 7b, we notice that WATSON significantly decreases analysis workload without sacrificing the accuracy of attack investigation. In the analysis of eight attack scenarios in 80 malicious sessions, WATSON shows two orders of magnitude (up to 284 times, with 134 times on average) workload reduction. We also see a massive reduction in the number of behaviors, as shown in Figure 7a. This is predictable as there are always more behavior instances than representative behaviors in a session, and WATSON can group them to prevent duplicate investigations.

To gain further insights, we study behavior abstraction of two attack cases from Table II: the Configuration Leakage and Content Destruction. Their audit events are visualized in Figure 8a and Figure 8b. We color-code information flows summarizing behavior instances.

Configuration Leakage. In this attack case, an attacker leverages the code executable vulnerability in `vim` to collect machine configuration for future compromise preparation. The background scenario is that one AI engineer in an enterprise intends to develop an image recognition model. She first downloads the training dataset from the company website as it cannot be found on the internal file server. Next, she implements and evaluates the learning model by submitting data and codes to a GPU server specialized for machine learning. Meanwhile, she uses `elinks` to search online about model optimization. Unfortunately, one malicious `text` file is mistakenly downloaded for her reference. The file successfully bypasses the prefix check in `vim`, invokes a `bash` to collect machine configuration, and further transfers the information remotely. To improve image prediction accuracy, the engineer frequently seeks model optimization techniques online, which introduces a bunch of noisy events.

As shown in Table IV, WATSON accurately recognizes the Configuration Leakage from other launched common behaviors. It can distinguish this behavior due to two reasons: (1) Collecting and transferring machine configurations is a unique behavioral pattern compared with daily routines like material download and code submission; (2) The (`vim`, `fork`, `bash`) triple is assigned high importance weight due to its low frequency in the system. Any behavior instance containing it would significantly deviate from the rest. Furthermore, WATSON clusters redundant behaviors like the Training Program Submission and the Online Material Reference, thus efficiently reducing analysis workload by avoiding duplicate inspection.

Content Destruction. This attack is an insider threat. Through a downloaded malicious payload, an attacker first exploits the Serv-U FTP local escalation vulnerability to invoke a privileged `bash`. After a successful initial compromise and foothold establishment, he discovers a directory of classified projects. However, the firewall blocks remote file transfer, so he decides to randomly tamper with completed programs. Since the knowledgeable insider is aware of the deployed IDS, he simulates an extensive number of ordinary activities to disguise himself as a regular developer, such as compiling programs and receiving daily tasks. Therefore, although IDSs generate an alert for this session, the attacker still wastes analysts intensive labor and time to pinpoint all behaviors and reconstruct attack scenarios. Traditionally, an analyst would have to investigate all the behaviors in a session to verify the truth of alerts reported by IDSs and explore other potential threats. To do so, a total of 550,281 audit events, where most are noisy information, are manually inspected. Fortunately, because most noisy behaviors incorporated by the insider to bury attack footprints are redundant, WATSON can efficiently assemble them and save analysis workload by 284 times. For instance, multiple Program Compilation instances are substituted with one behavior signature for analysis. We find WATSON particularly effective in insider threat investigation, which primarily credits to its capability of clustering benign behaviors.

2) Evaluation on DARPA TRACE *dataset:* According to the ground truth in the DARPA TC program report, DARPA TRACE dataset contains five attack scenarios, namely Fire-

TABLE V: Overview of attack cases in DARPA TRACE dataset with scenario descriptions. Column 3 shows the root data objects of malicious behaviors. Column 4 and Column 5 reports the number of nodes and edges in behaviors.

Attack Cases	Scenario Description	Root	Node	Edge
Firefox Backdoor	A malicious ad server exploits Firefox to execute an in-memory payload. This provides a remote console to exfiltrate sensitive information. A cache process is exploited and displayed similar behaviors as the compromised Firefox.	passwd	233	688
Browser Extension	The attacker exploits a target host via a vulnerable browser-plugin pass-mgr. The compromised plug-in downloads and executes a malicious program, which scans ports for internal recon and exfiltrates sensitive information.	gtcache	895	1750
Pine Backdoor	Pine is compromised by a malicious executable to scan ports for internal recon and establish a connection to the attacker's machine.	tcexec	67355	67453
Phishing Executable	The attacker sends a malicious executable as an e-mail attachment to exploit a vulnerability in Pine. However, the attack fails to run as expected even though the user manually downloads and executes the executable.	tcexec	22	23

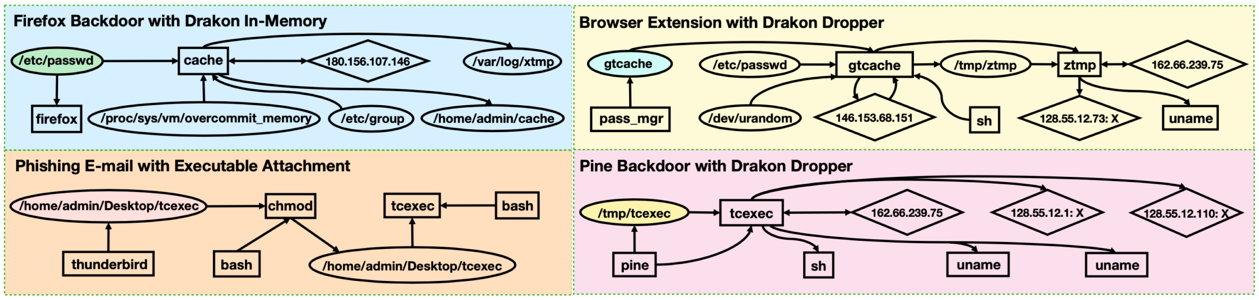


Fig. 10: Malicious behaviors of four APT attacks in DARPA TRACE dataset.

fox backdoor with Drakon in-memory (Firefox Backdoor for short), browser extension with Drakon dropper (Extension Backdoor for short), Pine backdoor with Drakon dropper (Pine Backdoor for short), phishing email link, and phishing email with executable attachment (Phishing Executable for short). We note that the phishing email link attack is by nature not visible in the system-call audit logs as there is no subsequent system operation on the victim's machine after the user visits the phishing website and enters credential information [39]. Therefore, we have omitted this attack from behavior analysis and investigate only the remaining attacks described in Table V. The details of these attacks' roots are shown in Table VI.

Figure 9 presents the reduction of analysis workload and behaviors for attack investigation in the DARPA TRACE dataset. As we can see, WATSON significantly decreases the analysis workload by around 930 times. Besides, the reduced behaviors indicate that WATSON can cluster semantically similar behaviors and help security analysts to stay focused on the inspection of representative behaviors. To better understand WATSON's efficacy, we then look into the behavior clusters related to APT attacks in Table V. Our analysis reveals that for three (i.e., Extension Backdoor, Pine Backdoor, and Phishing Executable) of four APT attacks, WATSON successfully summarizes the corresponding malicious behaviors in Figure 10, which match the ground truth provided. Moreover, our summarized malicious behaviors are generally comparable to attack scenarios constructed by existing work (e.g., MORSE [39]) that also investigates the DARPA TRACE dataset. Upon closer inspection of the remaining Firefox Backdoor attack, we discover that the data provenance of `cache` process (PID 26317) is missing in the original dataset. Accordingly, WATSON only captures the second part of the attack: a `cache` process exfiltrates sensitive information remotely.

Moreover, we observe that high-level behaviors of successful APT attacks are all abstracted as separate clusters. That is, no benign behavior is falsely classified as attacks. As an example of the Extension Backdoor attack in Figure 10, the attacker first compromises a vulnerable password manager extension in Firefox to implant an on-disk malicious program, `gtcache`. Then, the attacker executes the program to exfiltrate sensitive information (e.g., `/etc/passwd`) to the public network and perform a port scan of target hosts (e.g., `128.55.12.73`) on the internal network. We note that scanning ports and exfiltrating data are quite different from running common utilities in terms of contextual semantics. Specifically, the port scan for internal reconnaissance typically involves thousands of network connections to local network hosts. As a result, any behavior containing the port scan is labeled with different semantics from daily behaviors and constitutes a separate cluster. Another discovery is that the Extension Backdoor attack would be clustered together with the Pine Backdoor attack if we decrease the cluster merge threshold from 0.85 to 0.75. This is expected because both APT attacks leverage a command and control agent to scan network ports — a particularly unique behavioral pattern. It is also worth noting that the cluster of Phishing Executable attack includes normal behaviors unrelated to APT attacks. This matches our domain knowledge as Phishing Executable attack fails to exploit Pine's vulnerabilities and just demonstrates the user downloading and executing an email attachment, which is normal compared to regular email checking behaviors.

We further study what benign behaviors are clustered by WATSON. For reasons of space, we present two benign clusters, Syslog Rotation by Gzip (Syslog Rotation for short) and SSH Login with MOTD (SSH Login for short), in Figure 11. Syslog Rotation behavior is a common system routine that

TABLE VI: Root data objects of attack cases in DARPA TRACE dataset. Column 2 and 3 shows the name and UUID of roots. Column 4 represents the graph where WATSON locates malicious behaviors in Figure 10.

Attack Cases	Name	UUID	Graph
Firefox Backdoor	/etc/passwd	C13A910B-8966-7C95-549F-6EACF06F2429	ta1-trace-e3-official.json.125
Browser Extension	/etc/firefox/native-messaging-hosts/gtcache	17498F61-1D2A-DEB2-F6E5-EB447ABF4A60	ta1-trace-e3-official-1.json.3
Pine Backdoor	/tmp/tcexec	7169B097-1601-297F-2F6E-CEF5924F1C68	ta1-trace-e3-official-1.json.4
Phishing Executable	/home/admin/Desktop/tcexec	BBC43AE7-8DF9-49DD-44A0-030EEC564E84	ta1-trace-e3-official-1.json.4

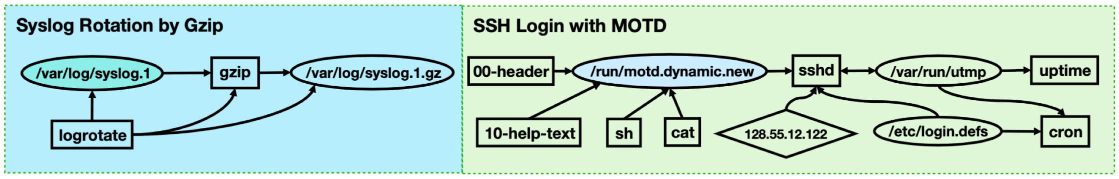


Fig. 11: Syslog Rotation and SSH Login behaviors in DARPA TRACE dataset.

compresses system logs (`/var/log/syslog.1`) to prevent them from growing too large on disk. In our experiment, WATSON summarizes five Syslog Rotation behaviors in one cluster with no false positive. The signature of Syslog Rotation cluster includes eight nodes and ten edges. SSH Login behavior displays the Dynamic Message Of The Day (MOTD) when a user logs in to the system through `ssh`. WATSON in total captures 16 of SSH Login behaviors in one cluster with four false positives. The signature of SSH Login cluster includes 13 nodes and 198 edges. Note that we cannot verify whether WATSON misses any true-positive benign behavior as the ground truth is unknown to us.

In summary, our results show that WATSON can cluster similar behaviors to reduce analysis workload while not losing accuracy in attack investigation. It is also worth mentioning that even though WATSON effectively abstract behaviors, large-scale benign behaviors may still cause trouble for analysts as individual behavior signatures from benign clusters require manual inspection. One approach to mitigating this problem is to borrow ideas from anomaly detection systems. For example, we can extract the signatures from a comprehensive benign dataset and report only deviations to analysts.

F. System Performance

We measure WATSON's performance overhead for behavior abstraction in our malicious dataset and the DARPA TRACE dataset. The scale and magnitude of these two datasets are generally comparable with that of daily user data. The on-disk sizes of a session in the malicious dataset and a graph in the DARPA TRACE dataset are on average 420 MB and 3 GB, respectively. We conduct the experiments on a Linux server with Intel(R) Core(TM) i9-9900X CPU @ 3.50GHz and 64GB memory. The Operating System is Ubuntu 18.04.4 LTS.

In this setting, WATSON abstracts behaviors from a malicious session and a DARPA graph within 35 and 170 seconds, respectively. We do not include the runtime overhead of KG construction, as it largely depends on audit logs' volume. However, we show that WATSON can parse 40k audit events in the default auditbeat format and build the KG within one second. Construction of the KG in the DARPA TRACE dataset format [43] is faster, operating at about 70k per second. Our current implementation loads audit events from disk and runs the experiments on a single machine using a single thread. The system efficiency can be further improved by main memory storage [38] and distributed graph processing [58]. Moreover, the constructed KG and abstracted behaviors represent the majority of WATSON's runtime memory overhead. While analyzing our malicious dataset (33 GB) and the DARPA TRACE dataset (635 GB), the memory consumption increases up to 2.6 GB. The storage overhead mainly comes from the audit events' embeddings and behavior database, which is on average 18 MB (10 MB for embeddings and 8 MB for behaviors) for a malicious session and 121 MB (78 MB for embeddings and 43 MB for behaviors) for a DARPA graph. We note that storage overhead does not increase linearly with the increase of sessions or graphs because events are recurrent on the system.

VI. DISCUSSION

In this section, we will introduce some of the design choices, implications, and possible extensions to this work.

Benefit for Related Solutions. Related security tools on log analysis require intensive manual efforts to develop knowledge in understanding audit events. WATSON acts as an "assistant" to these solutions for analysis workload reduction. In fact, half of the efforts of defining domain knowledge can be performed automatically. We can use procedural analysis to identify behavioral patterns and cluster similar ones via a quantitative representation of behavior semantics. Therefore, it is relatively easy for an analyst to investigate the representative behavioral patterns (i.e., behavior signatures) from clusters and provide domain-specific labels. Specifically, behavior signatures in our behavior database can be taken as the inputs to related security tools. For example, if TGMiner [35] applies WATSON to extract behavior signatures before formulating behavior queries, significant analysis efforts would be saved from labeling behaviors of interest in training logs, given that benign or redundant behaviors have been clustered. With the assistance of WATSON, MORSE [39] only needs to initialize tags for representative system entities (e.g., one of IPC files) rather than the whole entity space. In addition, WATSON uncovers event semantics indicating how "surprising" an event is for security analysis. This helps to reduce human workload in defining TTP specifications, as "unsurprising" (e.g., benign) events are identified before analysts extract attack patterns.

Embedding Space Retraining. Common to most learning approaches using embedding techniques [67], WATSON needs to retrain the embedding space periodically due to semantics shifts and the inclusion of previously unseen data. That said, our choice of a computationally efficient embedding model (TransE) helps to mitigate the overheads incurred by such retraining. Empirically in our experiments, we observe that WATSON typically retrains the embedding model on one daily session within 25 seconds. Especially, we can further leverage NLP techniques [19], [44], [69] to learn the semantics of "out-of-vocabulary" (unseen) audit events from their morphological information (e.g., file path) so that WATSON does not necessarily retrain the whole embedding space.

Robustness of Behavior Abstraction. To evade the behavior abstraction, an attacker may attempt to obfuscate a behavior by intentionally introducing irrelevant events. However, the impact of such events on behavior semantics is limited. In Section III-E, we design two strategies (*relative importance* and *noise events*) to improve WATSON's robustness for behavior abstraction. Specifically, while WATSON aggregates behaviors' contextual semantics, irrelevant events would be assigned low importance weights or even removed as noise events. Another potential approach to deobfuscating behaviors is to incorporate additional side information (e.g., semantically-rich arguments of audit events) into WATSON's KGs. We believe this can give WATSON more capabilities to sift through uninteresting events for security analysis. We acknowledge that more advanced techniques could be used to mimic normal behaviors. Nevertheless, mimicry attack detection itself is an open research problem [76] and beyond the scope of this study.

VII. Related Work

Causality Analysis. Causality analysis is an orthogonal but important problem relating to behavior abstraction. King and Chen [45] first introduce building a dependency graph on system audit logs to track back from a given security incident and locate its root cause. King et al. [46] improve the causality tracking by capturing forward and cross-host dependencies. A large number of research efforts have been further made to mitigate the dependency explosion problem and high storage overhead in causality analysis. Recent work has proposed fine-grained unit partition [49], [54], [56], [57], [73], dynamic tainting [66], [83], modeling-based inference [47], [48], record-and-reply [41], [42], and universal provenance [37] techniques to achieve more precise causality tracking. Another line of research strives to decrease overall log volume for analysis by graph compression [21], [34], [38], [40], [75] and data reduction [50], [57], [82]. Although the scope of WATSON is different from these solutions, its effectiveness relies on accurate causality analysis when correlating data transfers to summarize behaviors.

Behavior Abstraction. Abstracting behaviors as graph patterns or causal dependencies has proved useful in understanding OS-level activities and detect potential threats and risks. TGMiner [84] mines discriminative graph patterns from behaviors of interest and use them as templates to identify similar behaviors. Based on cyber threat intelligence reports, POIROT [62] extracts query graphs for APT behaviors and presents an alignment algorithm to search for their existence in provenance graphs. HOLMES [63] and RapSheet [35] view multi-stage attacks as a chain of causal events that match TTP specifications. SLEUTH [38] and MORSE [39] propose tag policies to model information leakage behaviors. Compared with prior work, we abstract behaviors as embeddings (numeric vectors) based on contextual information. Our findings suggest that this quantitative representation of behaviors can preserve behavior semantics and enable advanced behavior analysis (e.g., similar behavior clustering).

Embedding-based log analysis. Extensive literature exists on applying embedding techniques for other log analysis tasks. Such tasks include anomaly-based IDS [22], [25], [33], [52], [60], malware identification [18], [77], [78] and cyberattack evolution understanding [72]. Much prior work uses machine learning models such as neural networks, word embedding, and n-grams to embed logs into vectors. For example, DeepLog [25] is a neural network-based approach that leverages a long short-term memory (LSTM) to learn execution patterns in streams of normal log entries. PROVDETECTOR [77] applies a neural word-embedding model, doc2vec, to quantify behaviors of processes running on the system. Similarly, ATTACK2VEC [72] leverages a temporal word-embedding model to quantify the context in which cyberattack steps are exploited over time. On the contrary, WATSON first proposes employing a translation-based embedding, TransE, to uncover the contextual semantics of audit events. Our principle of using TransE is its translational distance matching our knowledge of event usage context. UNICORN [33] presents a graph sketching algorithm to summarize long-running system executions. Essentially, it makes use of statistical properties of graphs to represent semantic information, which is different from our contextual semantics learning approach.

VIII. Conclusion

Abstracting high-level behaviors from low-level audit logs is a key task in security response. It helps to bridge the semantic gap between audit events and system behaviors and thus reduce human efforts in log analysis. In this paper, we propose an automated approach, WATSON, to abstract behaviors from audit logs. WATSON addresses two primary challenges in event semantics inference and behavior summarization and aggregation. Specifically, WATSON leverages contextual information in log-based knowledge graphs to enable semantics inference. To distinguish representative behaviors, WATSON provides a vector representation of behavior semantics and uses it to cluster semantically similar behaviors. We evaluate WATSON against both behaviors simulated from real-life cyber attacks as well as behaviors of an adversarial engagement organized by DARPA. Our experimental results show that WATSON can accurately abstract both benign and malicious behaviors and dramatically reduce manual workload in attack investigation.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments. Thank Xiang Wang and Zhaoying Li for discussion and feedback on earlier drafts of this paper. We also thank Yankai Jiang and Jiahao Liu for their help on our experiments. Some of the experiments were conducted using the infrastructure of National Cybersecurity R&D Laboratory, Singapore. This research was supported in part under the grant "Asian Institute of Digital Finance" supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative, by National Research Foundation, Singapore under its International Research Centres in Singapore Funding Initiative and Ministry of Education Humanities and Social Science project, China (No.16YJC790123), by Beijing Natural Science Foundation (No.4202036), and National Natural Science Foundation of China (No.61871023). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect views of the funding agencies.

References

[1] CVE-2019-12181. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12181.

[2] CVE-2019-12735. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12735.

[3] CVE-2019-13272. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13272.

[4] DARPA Transparent Computing. https://www.darpa.mil/program/transparent-computing.

[5] Endpoint Monitoring and Forensics. https://logrhythm.com/products/endpoint-monitoring.

[6] Information on the Capital One Cyber Incident. https://www.capitalone.com/facts2019.

[7] Lightweight shipper for audit data. https://www.elastic.co/beats/auditbeat.

[8] Linux audit daemon. https://github.com/linux-audit/audit-userspace.

[9] Linux Kernel Audit Subsystem. https://github.com/linux-audit/audit-kernel.

[10] Password Reuse Attacks. https://www.compasscyber.com/blog/password-reuse-attacks.

[11] Postgresql Relational Database. https://www.postgresql.org.

[12] PROV-DM: The PROV Data Model. https://www.w3.org/TR/2013/REC-prov-dm-20130430.

[13] Transparent Computing Engagement 3 Data Release. https://github.com/darpa-i2o/Transparent-Computing/blob/master/README-E3.md.

[14] Twitter hack. https://www.theguardian.com/technology/2020/jul/15/twitter-elon-musk-joe-biden-hacked-bitcoin.

[15] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[16] Gabor Angeli, Melvin Jose Johnson Premkumar, and Christopher D Manning. Leveraging linguistic structure for open domain information extraction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 344–354, 2015.

[17] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security Symposium*, 2015.

[18] Konstantin Berlin, David Slater, and Joshua Saxe. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pages 35–44, 2015.

[19] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[20] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *NeurIPS*, 2013.

[21] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Distributed provenance compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 203–218, 2017.

[22] Ting Chen, Lu-An Tang, Yizhou Sun, Zhengzhang Chen, and Kai Zhang. Entity embedding-based anomaly detection for heterogeneous categorical events. In *IJCAI*, 2016.

[23] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *USENIX Security Symposium*, 2017.

[24] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *ACM KDD*, 2017.

[25] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *ACM CCS*, 2017.

[26] Michael Färber, Frederic Bartscherer, Carsten Menne, and Achim Rettinger. Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago. *Semantic Web*, 9(1):77–129, 2018.

[27] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. SAQL: A stream-based query system for real-time abnormal system behavior detection. In *USENIX Security Symposium*, 2018.

[28] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. AIQL: Enabling efficient attack investigation from system monitoring data. In *USENIX ATC*, 2018.

[29] Ashish Gehani and Dawood Tariq. Spade: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*, pages 101–120. Springer-Verlag New York, Inc., 2012.

[30] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. The taser intrusion recovery system. In *SOSP*, 2005.

[31] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *ACM KDD*, 2016.

[32] Jiaping Gui, Ding Li, Zhengzhang Chen, Junghwan Rhee, Xusheng Xiao, Mu Zhang, Kangkook Jee, Zhichun Li, and Haifeng Chen. APTrace: A responsive system for agile enterprise level causality analysis. In *IEEE ICDE*, 2020.

[33] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. In *NDSS*, 2020.

[34] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS*, 2018.

[35] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *IEEE Security and Privacy*, 2020.

[36] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS*, 2019.

[37] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In *NDSS*, 2020.

[38] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. Sleuth: Real-time attack scenario reconstruction from cots audit data. In *USENIX Security Symposium*, 2017.

[39] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *IEEE Security and Privacy*, 2020.

[40] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, Evan Downing, et al. Dependence-preserving data compaction for scalable forensic analysis. In *USENIX Security Symposium*, 2018.

[41] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM CCS*, 2017.

[42] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *USENIX Security Symposium*, 2018.

[43] Joud Khoury, Timothy Upthegrove, Armando Caro, Brett Benyo, and Derrick Kong. An event-based data model for granular information flow tracking. In *USENIX TaPP*, 2020.

[44] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *AAAI*, 2016.

[45] Samuel T King and Peter M Chen. Backtracking intrusions. In *SOSP*, 2003.

[46] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[47] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. In *ASPLOS*, 2016.

[48] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. MCI: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, 2018.

[49] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[50] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. In *ACM CCS*, 2013.

[51] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *AAAI*, 2015.

[52] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *ACM CCS*, 2019.

[53] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[54] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In *ACM ACSAC*, 2015.

[55] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX ATC*, 2018.

[56] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security Symposium*, 2017.

[57] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[58] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*. ACM, 2010.

[59] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

[60] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *ACM KDD*, 2016.

[61] Noor Michael, Jaron Mink, Jason Liu, Sneha Gaur, Wajih Ul Hassan, and Adam Bates. On the forensic validity of approximated audit logs. In *ACM ACSAC*, 2020.

[62] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. POIROT: Aligning attack behavior with kernel audit records for cyber threat hunting. In *ACM CCS*, 2019.

[63] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *IEEE Security and Privacy*, 2019.

[64] Eric Miller. An introduction to the resource description framework. *Bulletin of the American Society for Information Science and Technology*, 25(1):15–19, 1998.

[65] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *USENIX ATC*, 2006.

[66] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[67] Erik Ordentlich, Lee Yang, Andy Feng, Peter Cnudde, Mihajlo Grbovic, Nemanja Djuric, Vladan Radosavljevic, and Gavin Owens. Network-efficient distributed word2vec training system for large vocabularies. In *ACM CIKM*, 2016.

[68] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime analysis of whole-system provenance. In *ACM CCS*, 2018.

[69] Yuval Pinter, Robert Guthrie, and Jacob Eisenstein. Mimicking word embeddings using subword rnns. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017.

[70] Omid Setayeshfar, Christian Adkins, Matthew Jones, Kyu Hyung Lee, and Prashant Doshi. Graalf: Supporting graphical analysis of audit logs for forensics. *arXiv preprint arXiv:1909.00902*, 2019.

[71] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events through deep learning. In *ACM CCS*, 2018.

[72] Yun Shen and Gianluca Stringhini. ATTACK2VEC: Leveraging temporal word embeddings to understand the evolution of cyberattacks. In *USENIX Security Symposium*, 2019.

[73] Sriranjani Sitaraman and Subbarayan Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In *IEEE IWIA*, 2005.

[74] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *NeurIPS*, 2014.

[75] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. Node-merge: template based efficient data reduction for big-data causality analysis. In *ACM CCS*, 2018.

[76] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM CCS*, 2002.

[77] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, C Gunter, et al. You are what you do: Hunting stealthy malware via data provenance analysis. In *NDSS*, 2020.

[78] Shen Wang, Zhengzhang Chen, Xiao Yu, Ding Li, Jingchao Ni, Lu-An Tang, Jiaping Gui, Zhichun Li, Haifeng Chen, and S Yu Philip. Heterogeneous graph matching networks for unknown malware detection. In *IJCAI*, 2019.

[79] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. Kgat: Knowledge graph attention network for recommendation. In *ACM KDD*, 2019.

[80] Xiang Wang, Xiangnan He, Fuli Feng, Liqiang Nie, and Tat-Seng Chua. Tem: Tree-enhanced embedding model for explainable recommendation. In *ACM WWW*, 2018.

[81] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *AAAI*, 2014.

[82] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM CCS*, 2016.

[83] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *ACM CCS*, 2007.

[84] Bo Zong, Xusheng Xiao, Zhichun Li, Zhenyu Wu, Zhiyun Qian, Xifeng Yan, Ambuj K Singh, and Guofei Jiang. Behavior query discovery in system-generated temporal graphs. In *VLDB*, 2015.