Few-Shot Learning-Based Cyber Incident Detection with Augmented Context Intelligence

Fei Zuo*, Junghwan Rhee*, Yung Ryn Choe†, Chenglong Fu‡, Xianshan Qu*
*University of Central Oklahoma, †Sandia National Laboratories, ‡University of North Carolina at Charlotte
{fzuo, jrhee2, xqu1}@uco.edu, yrchoe@sandia.gov, chenglong.fu@uncc.edu

***Abstract*—In recent years, the adoption of cloud services has been expanding at an unprecedented rate. As more and more organizations migrate or deploy their businesses to the cloud, a multitude of related cybersecurity incidents such as data breaches are on the rise. Many inherent attributes of cloud environments, for example, data sharing, remote access, dynamicity and scalability, pose significant challenges for the protection of cloud security. Even worse, cyber threats are becoming increasingly sophisticated and covert. Attack methods, such as Advanced Persistent Threats (APTs), are continually developed to bypass traditional security measures. Among the emerging technologies for robust threat detection, system provenance analysis is being considered as a promising mechanism, thus attracting widespread attention in the field of incident response. This paper proposes a new few-shot learning-based attack detection with improved data context intelligence. We collect operating system behavior data of cloud systems during realistic attacks and leverage an innovative semiotics extraction method to describe system events. Inspired by the advances in semantic analysis, which is a fruitful area focused on understanding natural languages in computational linguistics, we further convert the anomaly detection problem into a similarity comparison problem. Comprehensive experiments show that the proposed approach is able to generalize over unseen attacks and make accurate predictions, even if the incident detection models are trained with very limited samples.**

***Index Terms*—Incident Detection, Anomaly Detection, Cyber Threat, Cloud Security, Few-Shot Learning.**

I. Introduction

Cybersecurity incidents are emerging incessantly nationwide. Commercial organizations, government bodies, and even educational institutions can all be potential targets for cyberattacks. For example, an investigation [1] conducted on 550 organizations revealed that 83% of them had more than one data breach, and the cost of a data breach averaged USD 4.35 million. Furthermore, over the recent years, a noteworthy trend has been the sharp increase in the number of attacks on cloud environments per organization, "*which shot up by 48% in 2022 compared with 2021*" [2]. Even worse, cloud environments are demonstrably vulnerable to Advanced Persistent Threats (APTs), which can be covert over a prolonged period of time but difficult to defend. Therefore, in this paper, we focus on threat detection towards cloud incidents.

In the arsenal of threat detection, system provenance analysis is believed to possess great potential in detecting cyber threats because system-level data can not only represent complex dependencies within a system, which is crucial for understanding potential threats, but also correlate with attack scenarios, providing a valuable historical context that helps in predicting and preventing future attacks. The two interrelated system entities, along with the operation between them, collectively constitute an event, which is a basic unit in provenance data for tracking and recording system-level behaviors. We thus extract events from a provenance graph and regard them as informative features for describing the characteristics of a cyber incident.

More specifically, in an event, the entity that issues an operation is modeled as a *subject*, while the other entity that passively undergoes an operation is modeled and referred to as an *object*. Inspired by this, we analogize the semantic description of an event to a *sentence* in linguistics. To this end, we particularly develop a semiotics extraction method to capture the event's semantics with enriched expressivity. Then, we further adopt the embedding method in Natural Language Processing (NLP) to generate a numeric representation for every event's description, which will lay a solid foundation for the subsequent threat detection.

Furthermore, we notice that with the tremendous success of machine learning in the recent decade, leveraging related advancements to assist and automate system provenance analysis has become more of a need than an option. Industry statistics show that in practical incident response applications, fully deployed AI-driven systems "*were able to identify and contain a breach 28 days faster than those that didn't, saving USD 3.05 million in costs*" for the organizations [1].

It is generally acknowledged that training high-quality machine learning models typically requires massive amounts of data. However, well-labeled large datasets are usually a precious or even scarce resource in the field of cybersecurity. One widely seen challenge in collecting cyber incident data is that the actually expected distribution of samples is not balanced because system events in practical scenarios are mostly benign. Even worse, this imbalance is quite likely to result in a high false positive rate, which would be less useful for many security-sensitive tasks. Not surprisingly, researchers consider existing threat detection methods "*may not be robust enough to distinguish malicious behavior from benign ones accurately*" [3]. Thus, what we are looking for is a more effective threat detection technique towards realistic incidents in cloud environments, which can cope with not only various known attacks but also previously unseen ones.

In the field of machine learning, Few-Shot Learning (FSL) aims at enabling models to generalize over unseen data and make accurate predictions, even if they are trained with very

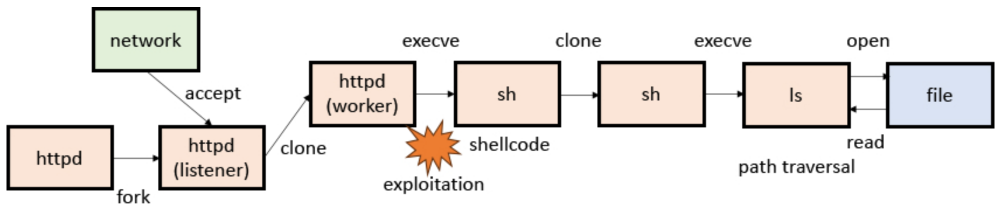


Fig. 1: A provenance graph example (CVE-2021-41773 case).

limited samples. In light of the great progress in FSL, we convert anomaly detection within the context of incident response into a semantic comparison problem. Our intuition is that normal and attack behaviors can be mapped to two different clusters in a certain feature space. The FSL-based neural network model is thus used as a kernel function, which maps the data points from the original space to a feature space where the intra-cluster distance is relatively larger than the inter-cluster distance. Based on a well-trained model, we are able to predict whether a previously unseen behavior is an attack or not, by comparing it with a known case.

Following the aforementioned pipeline, we develop a cyber threat detection system for incident response. First, we adopt an innovative semiotic extraction method, which can accurately capture the key semantics of system call-based program behaviors. To generalize cyber threat intelligence over unseen attacks, we present a new few-shot learning technique applied to provenance data, which converts an anomaly detection problem into a similarity comparison problem. We have explored a systematic strategy to address security threats in an increasingly challenging cloud environment. To this end, realistic cyber-attacks on cloud applications are investigated in this work. At last, wee empirically demonstrated that few-shot learning is applicable to operating system behavior datasets. The comprehensive experiments show that our proposed approach is capable of generalizing over previously unseen attacks and making accurate classifications, even if there exist very limited training samples in each attack scenario.

II. Background

In incident response research and practice, provenance data reflects activities at the level of system entities, which offers great potential for tracking the dependencies across system events and further exposing attack sequences. System-level provenance can usually be recorded in the form of graphs, where vertices and edges represent entities and the operations amongst entities respectively.

Figure 1 shows a provenance graph example that exposes an attack scenario in an Apache HTTP server. Upon a connection from the network with an `accept` system call, a worker process of the web server (`httpd`) is invoked. Then due to the exploit code from an attack, this server allows the shellcode which invokes the `ls` process that lists files as a demonstration of arbitrary command execution.

The system entities refer to the components within a system responsible for producing, modifying, or processing information and resources. Examples of entities include processes, files, and network objects, which are indicated by orange, blue, and green nodes in Figure 1, respectively. For various operating systems, similar types of entities can be found. Therefore, we can apply a similar approach to different operating systems.

The operations between two system entities are described using a system call, which is a lower-level service interface invoked by software to use the privileged services or resources provided by the kernel. For instance, process operations can include `execve`, `fork` and `clone` system calls. Besides, `open`, `close`, `read`, and `write` system calls are examples of file operations. Lastly, operations such as `connect` and `accept` system calls are invoked by network activities.

When taking both operations and the entities that issue such operations into consideration, all these elements come together to form an event. Moreover, multiple events can manifest causality dependency if there exist direct or indirect data and control flows across them. Provenance data is often represented as a graph because graphs can intuitively depict dependencies between events or their chronological order. However, considering that hackers launch multi-stage attacks, extracting attack-relevant events from a vast number of system events over a long time span and linking them is non-trivial. As a result, the generated provenance graph often contains significant noise, posing challenges for threat detection. Therefore, it should be noted that **the proposed approach focuses on system events in provenance data**.

III. Problem Definition and Basic Assumptions

A. Problem Statement and Objectives

In the operating system of a cloud environment, the target that we are interested in monitoring is processes. A process p is an executing instance of a program A. The behavior of a process in the system is recorded in the form of provenance data consisting of a large number of events. Given a new process instance p of A, our aim is to infer whether it implicitly involves a malicious attack by analyzing its provenance data. In particular, our research objective is to seek an effective threat detection technique for realistic incidents in cloud, which can handle previously unseen attacks. Intuitively, capturing and modeling the behaviors of each program by a provenance graph seems feasible, but challenges still lie ahead of us. In practice, we need to address the following two challenges.

Challenge 1: Representativeness and Generalizability. In modern computer systems, the programs that might be

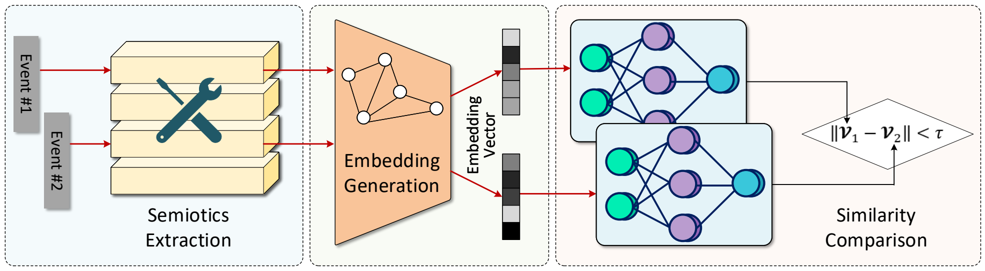


Fig. 2: System overview.

used are diverse. Exhaustively enumerating every possible program and empirically constructing provenance graphs to indicate the presence or absence of malicious attacks requires domain expertise and substantial human effort. In addition, cloud environments are complex and dynamic. For the same program, provenance graphs from a similar workload may look identical, but if the input workloads differ, the resulting provenance graphs may also differ. Therefore, it is uncertain whether stereotype provenance graphs established based on experience or limited simulations can cover all possible scenarios. Finally, malicious attacks can also be mixed and hidden in normal operations. Therefore, attackers can generate a complex provenance graph by adding a large number of normal events, attempting to obscure the malicious events within. As a result, treating the graph as a whole for anomaly detection becomes undesirable.

To address this challenge, we decompose a provenance graph into events, which is a basic unit for recording system-level behaviors. The benefits of this approach are evident. Different APT attacks in a cloud environment share commonalities at the level of system calls. Therefore, malicious events captured in one attack scenario can be generalized to other unseen attack scenarios. Again, from the attack example of Apache HTTP server (Figure 1), we observed that remote shell execution is issued by `httpd`, indicating exploitation. However, these events should not occur when normal workloads are input. On top of that, the object of examination is no longer the entire graph, but individual events. Hence, methods that use normal operations to obscure malicious attacks at the graph level are no longer effective. After using appropriate embedding methods, benign events form clusters in the high-dimensional feature space, which are distinct from the clusters formed by adversarial events.

Challenge 2: Feature Preparation for Anomaly Detection. The performance of traditional machine learning methods heavily relies on feature engineering, where domain knowledge plays a significant role in creating and selecting features from raw data. Moreover, current powerful system provenance data recorders, e.g., `Sysdig` and `Auditd`, are able to provide abundant fields to describe an event, for instance, but not limited to timestamps, system call names, process IDs (PIDs), file names, IP addresses, port numbers, and user IDs (UIDs), etc. Therefore, which features to select and how to evaluate the impact of each feature on the performance of anomaly detection in different attack scenarios depend on expertise and manual effort. Not only that, many fields in provenance data are descriptive text and therefore unstructured, such as file paths or process names. These unstructured data can be further divided into two categories. The first category consists of meaningful entities that correspond to specific tasks in the system. The other category consists of non-representative entities that are temporarily generated during intermediate steps, such as temporary files. Obviously, we need to design different response strategies for these two categories of entities. To sum up, how to prepare input features to represent system events for anomaly detection models is non-trivial.

To address this challenge, we view a system event as a sentence in natural language. The nodes and edges in a graph correspond to the subject, predicate, object, or complement of a sentence, which describes a system-level behavior. Furthermore, we can project a sentence to a numerical vector in the feature space using those advanced neural networks-based embedding methods. A key advantage of neural networks is their ability to learn directly from the raw data with minimal need for feature engineering. Relying on the embedding method for accurate semantic capture, in the feature space, sentences with similar semantics have closer vector representations, while different sentences are farther apart. For example, "`java execve sh`" is supposed to possess a numerical vector representation approximating that of "`httpd execve sh`". As a result, we are able to develop distance-based classifiers as our anomaly detection models.

B. Threat Model and Assumptions

Our threat model has several assumptions on adversary activities of our interest. First, we assume the security attack involves the change of the operating system calls as behavior to use system resources or services. While it is possible to make trivial attacks without involving system calls, they would have very limited impact because most serious impacts such as privilege changes, files, network activities seen in privilege escalation and data leak require system calls. The attacks without any system calls are hence out of focus.

Second, adversaries may use software exploits to cause compromise of software and subsequent damages. However, we assume the integrity of the security event recorder so that the system calls of the adversary events can be properly recorded. Similar to event detection and response (EDR)

TABLE I: Types of system calls used for semiotics.

Event type	System call names
Process events	`fork, vfork, clone, exec, kill`, and variants
File events	`open, close, read, write, unlink, dup, rename, chmod`, and variants
Network events	`listen, connect`, and variants

solutions, the recorded events are remotely transferred and the integrity of the OS kernel and the data recorder are assumed to be protected.

IV. Methodology

In this section, we propose a new cyber threat intelligence framework for the analysis and detection of cloud incidents. Figure 2 shows the overview of our proposed approach, which consists of three major components, i.e., semiotics extraction, embedding generation, and similarity comparison.

A. Key Insights

Our proposed methods are motivated by several key insights on the provenance data collected from realistic cloud incidents.

Program behavior can be abstracted into text. Programs utilize common OS interfaces that have a uniform format. Hence, we can standardize the description of program behavior into generic sentences.

***Insight* 1:** We embed system event details as natural language sentences that are composed of subject entities (i.e., programs), the type of operations acted by the subject entities as a predicate (e.g., system calls), and object entities that are the target of the operations by the subject entities. The object component includes ❶ processes, ❷ files, and ❸ network entities (e.g., IP addresses).

Certain process events lack sufficient details. Certain system events may have limited context of behavior because of the way software handles code. One example is interpreters such as `bash`, `python`, and `java`. These programs rely on common software binaries even though the actual software code differs. Such differences in program code are not determined from the main executable names (i.e., interpreters' names) because they are provided as parameters to them. Our insight was that assisting the model in such lacking contexts can drastically improve the performance of machine learning based solutions.

***Insight* 2:** In linguistics, an object complement is a grammatical construction that provides additional information about a direct object in a sentence. Drawing an analogy to this, we improved the system events description by supplementing extra information such as process executable names.

Non-representative files are widespread. Operating systems use a lot of "temporarily generated" information, which is used in intermediate steps. Such information is generally not important, as it is eventually deleted after a certain period or upon reboot, and it can introduce unnecessary complexity into the machine learning process. These files can be identified due to specific locations or naming schemes.

***Insight* 3:** We leveraged multiple patterns to recognize and normalize non-representative files. In addition, we use an algorithm to detect hash-like names based on a high diversity of file names.

B. Semiotics Extraction

We use operating system calls for our behavior modeling. While our experiment is mainly focused on Linux, our method is general and the same mechanism can be applied to other operating systems which also have system calls as seen in related work [4]–[7].

1) Program behavior abstraction: Table I shows a list of system calls collected, extracted, and used in our work. Process events include process creation system calls such as `fork`, `vfork`, and `clone` and the `exec` system call that replaces the program image inside a process is also used. `kill` system calls are used to terminate processes. File events refer to file behavior such as file creation, file deletion, file open, file close, file read, file write, rename, duplication etc. For instance, `open`, `close`, `read`, `write` are common examples. In Unix-like operating systems, almost everything is accessible as a file. Therefore, most system behavior can be tracked by utilizing file activities. Over the years, operating systems have added support for variant system calls. One example is the `*at` system calls (e.g., `openat`) which enable file operations relative to the referred directory descriptor in a parameter. Another example is `p*` system calls (e.g., `popen`) which are used to operate a process with a pipe. We only list the major system calls here. Their variants are available depending on the versions of the Linux kernels. Network events for both the client side (e.g., `connect`) and the server side (e.g., `listen`) are considered.

Modeling system events as sentences. A system call is always executed on behalf of a particular process, which serves as the identity of that particular system call action. Hence, we model a system event in the following triplet form, that is composed of a `Subject` (a process initiating a system call), `Predicate` (a system call type), and `Object` (a target upon which the system call is applied). For instance, in the example below, we have two system event descriptions where the `ps` program operates on the `stat` file.

```
(Subject)  (Predicate) (Object)
 ps         open        stat
 ps         close       stat
```

Data enhancement for certain program types. In Unix-like operating systems, a program identity is determined by an executable loaded by the `exec` system call. While this method works for most programs, there are exceptions where program executables are not representative.

1) **Interpreter-based scripting shells:** Shells with scripting capabilities use the script interpreter as the main executable. For example, Linux shells (e.g., `bash`, `csh`,

zsh) use the main script file as the first parameter allowing it to be used together with the executable file to represent the program.

2) **Software platforms using interpreters:** Multiple software languages use interpretation for code execution. Well known examples include `Python`, `Perl`, `Ruby`, and `JavaScript`. Programs falling into this category will be recognized by its interpreter executable name and the script file name.
3) **Single software-based[1] virtual machines:** Virtual machines execute an executable after obtaining the code file as one of the command-line parameters. For instance, `Java`, `Scala`, and `Kotlin` use the Java Virtual Machine (JVM) to run their executable files.

These cases commonly describe the program's identity as one of the parameters referring to the program file. Consequently, we handled such cases by appending the program file parameter to the behavior description sentence, which is analogical to an object complement in a natural language sentence.

New modeling of system events as sentences. Based on the initial modeling introduced earlier, we further improve it with the following new ideas in this work. The new augmented sentence format thus obtained is

```
(Subject) (Predicate) (Subject complement)
(Object) (Object complement)
```

This new sentence format is grounded in multiple aforementioned insights. The "`Subject complement`" and "`Object complement`" are two newly added optional components.

- **Subject complement:** The data enhancement for shells, interpreters, and single-software based virtual machines are supported with supplemental parameters of program execution. Such information is listed as one of the subject complement after applying multiple normalization techniques. Plus, we simplify the program arguments by removing flags and normalizing hash-like values.
- **Object complement:** We applied normalization to the object of each system event as shown in Algorithm 1 (See Section IV-B2). It uses multiple normalization techniques including hash detection to reduce the diversity of tokens that are not likely to be reused.

More concretely, we present several examples demonstrating how this augmented semantic information improves system event representations.

```
# (Subject): java
# (Predicate): clone
# (Subject complement): executable parameter
   extension = java 8983 /opt/solr/server/logs
   start.jar
# (Object): <NA>

BEFORE: java clone <NA>
AFTER:  java clone java 8983 /opt/solr/server/logs
        start.jar <NA>
```

Listing 1: Example 1 of augmented system event expression.

[1]We used the term, "single software-based" virtual machine to differentiate it from the virtual machines that execute an OS such as `virtualbox`.

Listing 1 shows the first example, where the augmented subject complement adds new information to the process context. The initial sentence (`BEFORE`) only had the program name (`java`), the system call name `clone`, and invalid target, `<NA>`, indicating that object information is not used for this type of system call (`clone`). The new format (`AFTER`) shows that, ffter filtering out the option fields starting with '`-`', several command-line parameters are listed, and the main Java JAR file, `start.jar`, appears.

```
# (Subject): dockerd
# (Predicate): openat
# (Object): .tmp-config.v2.json07205514
# (Object complement):
#   .tmp-config.v2.json072055514 -> <TMP>

BEFORE: dockerd openat .tmp-config.v2.json07205514
AFTER:  dockerd openat <TMP>
```

Listing 2: Example 2 of augmented system event expression.

As shown in Listing 2, the second example illustrates the normalization process. The original form (`BEFORE`) included a temporary file name, `.tmp-config.v2.json07205514`, which is not likely to be reused and thus is not helpful for subsequent machine learning task. The new format (`AFTER`) shows that the Docker daemon (`dockerd`) opens a temporary file that is normalized to `<TMP>`.

2) Data normalization of non-representative data: Modeling raw data requires careful arrangement of the data because it is well-known that non-representative (e.g., volatile noise that may not be used twice) can cause undesired side effects to machine learning performance. For example, the exact names are not likely to appear in the test run. The resulting out-of-vocabulary issue often causes language models to misunderstand or misinterpret the meaning of the input. Therefore, these identifiers are better to be normalized for our machine-learning tasks. To this end, we propose to use several mechanisms based on reasonable assumptions. We recognize such files and replace them using a constant label such as `<TMP>`, `<PIPE>`, or `<HASH>` depending on categories. Algorithm 1 specifically demonstrates our processing strategy.

Normalization of non-representative files based on operating system knowledge: One assumption is that we can utilize well-known knowledge about operating systems. For instance, most operating systems have well-known common directory locations used by kernels (e.g., `/bin`, `/tmp`) as described in operating manuals. Certain directories in UNIX variants such as `/run/`, `/dev/` and `/proc/` are all utilized to contain operating system internal states such as devices, and a list of program processes. Such states have high volatility or randomness that may cause out-of-vocabulary issues in machine learning approaches. Thus, the files in such directories are normalized by replacing them with `<TMP>`. Any file used for a pipe (in Unix everything is a file) is generalized as `<PIPE>` as well. For brand-new operating systems where pre-knowledge is incomplete, this knowledge can be easily figured out by a system administrator with a one-time effort.

Algorithm 1 An algorithm for normalization of a token

```
Require: token $t$, and IsHighVarietyCharacterName($t$)
 1: if $t$ is in the directories reserved for temporary files or has the .tmp extension then
 2:     return <TMP>
 3: end if
 4: if $t$ is in the OS internal state directories such as /run/, /dev/, or /proc/ directory then
 5:     return <TMP>
 6: end if
 7: if $t$ is a pipe then
 8:     return <PIPE>
 9: end if
10: if $t$ is a hash-like identifier then
11:     return <HASH>
12: end if
13: return $t$
```

Normalization of temporary files: Temporary files are generated by programs or operating systems for short-term use. They are another source of non-representative data, since such file names may not be persistent. Our assumption to handle this data is that such knowledge is also well-known or the knowledge can be easily obtained with one-time effort. For instance, `/tmp` in UNIX, `C:\Users\Username\AppData\Local\Temp`, or `C:\Windows\Temp` in Windows are non-regular (i.e., specially designated) directories for temporary files. Algorithm 1 illustrates our practice in handling non-representative file names of targets whose names are randomly determined. In addition, the file extensions for temporary files (e.g., `.tmp`) are also used.

Normalization of the identifiers with a high variety of characters: Another source of randomness comes from the identifiers that alternate numbers (`0-9`), alphabets (`a-zA-Z`), and special characters (e.g., `@,-,_,$,%`). We observed that that certain programs generate hash-like filenames (e.g., `75619cbc-879c-4076-8539-181392588ced`) because they use certain algorithms such as UUID, MD5, SHA that alternate alphabets and numbers for filenames. To mitigate the negative impact caused by non-representative data, any file or directory name recognized as a hash-like literal is normalized using `<HASH>`. At a high level, our determination method uses generally two different strategies. First, the files with well-known extensions such as `.conf` (configuration files), `.jar` (Java Archive file), and `.so` (libraries) are ruled out from being considered because they may be used consistently over multiple executions with the same names even with complex names. Second, for the rest of the filenames, if they exhibit the characteristic of frequent transitions among letters, digits, and special characters, they will be recognized as a hash-like identifier.

In a nutshell, our normalization algorithms opportunistically generalize the names of non-representative file and directory names based on operating system knowledge and data with simple pattern matching. If their names are not matched, simply the raw names will be used without being replaced by generalized names. They would not cause runtime errors or incorrect results.

C. Embedding Generation

The provenance record contains a large amount of unstructured data. To facilitate subsequent machine learning tasks, embedding methods are required. In natural language processing, embedding refers to the process of representing text as a fixed-length numerical vector in a continuous vector space. We have observed that multiple embedding techniques have been proposed for words [8], sentences [9] and documents [10]. This vector representation not only extracts the semantic meaning of the text but also can be further fed into a neural network. From the semiotics extraction stage, we have obtained textual descriptions for each system event. Therefore, in the embedding generation stage, we particularly adopt `Sent2Vec` [9] to generate numeric representations for these events. This method leverages n-gram features and an efficient unsupervised objective to capture the contextual information of sentences. As a result, we use a 50-dimensional vector to represent an event. Higher dimensions often have the capacity to encode richer information, but they also typically entail greater computational complexity. Considering the relatively small vocabulary size and the short sentence length in our application, we empirically choose 50 as the dimension of embeddings.

D. Similarity Comparison

In Few-Shot Learning, Siamese networks stand out particularly in the area of detecting similarities among multiple comparable items, and they have already been applied in security applications [11], [12]. We thus propose a Siamese-network-based threat detection method that converts the anomaly detection problem into a semantic comparison problem. In the training phase, we prepare three types of event pairs: ❶ both events in a pair are benign; ❷ both events in a pair are adversary; ❸ one event is benign while the other is adversarial. Benign and adversary events inherently come from two different clusters in the feature space, and the intra-cluster distance is relatively larger than the inter-cluster distance. Consequently, the ground truth assigned to the first and second types of event pairs is a binary label false, i.e., the two events in a pair are considered similar. By contrast, the ground truth assigned to the third type of event pair is true, indicating that the two events in the pair are from two distinct clusters.

Furthermore, we adopt a lightweight and shallow network design for each sub-network of the Siamese architecture, comprising three layers of fully-connected feed-forward neural networks with ReLU activation functions. The purpose herein is to explore how well the threat detector performs even when it only uses a simple network design. Once the neural network with fine-tuned weights is constructed through training, we can determine if a previously unseen event is adversarial or not by evaluating the discrepancy between it and a known event.

TABLE II: The number of system events extracted from provenance data

	C01∼C05	C06	C07	C08	C09	C10	C11
Events in *ben.* scenarios	4,341	272	324	1,620	336	339	3,678
Events in *adv.* scenarios	4,516	438	237	2,527	477	399	5,041
Overlap between *ben.* and *adv.*	2,629	178	190	1,202	309	324	2,958
Events in *adv.* after refinement	1,887	260	47	1,325	168	75	2,083

The optimization objective of our model training is to minimize a contrastive loss [13], which is a distance-based loss. This loss function is widely used in few-shot learning to describe the degree of similarity between two samples. Namely, two similar inputs have a small distance and two dissimilar inputs instead have a relatively large distance.

Our evaluation shows that, even with a relatively small training dataset and a network with very few layers, our threat detection solution can still distinguish previously unseen malicious incidents from benign ones.

V. Evaluation

A. Dataset

The performance of our technique was evaluated using a publicly accessible research dataset ProvSec [14], which includes provenance analysis data collected from a cloud environment involving 11 attack scenarios. They reflect real cyber attacks in cloud based on CVE entries and the corresponding proof-of-concept (PoC) exploit code. In the following description, we use C01∼C11 to represent each category of attacks.

When the attack is on, the corresponding provenance data of a specific adversary scenario is recorded. We accordingly extract events and preliminarily consider them as adversarial. Otherwise, if the attack has not been launched, the events extracted from the corresponding provenance data are regarded as benign. It is unsurprising that there exist some events that may be on both sides. The intersection between the benign events and the preliminary adversary events is not sufficiently iconic to indicate an attack behavior. Therefore, we remove those overlapped patterns from the original adversary events to obtain a refined dataset. The details of events per category extracted from provenance data are shown in Table II.

Furthermore, we prepare both similar and dissimilar pairs in the dataset to train a Siamese-network-based model. The two events in a dissimilar pair are from different clusters, i.e., one is benign while the other one is adversarial. In contrast, the two events in a similar pair are from the same clusters. Namely, they must be either both benign or both adversarial. Table III shows the number of pairs per category, where the sizes of similar and dissimilar pairs are balanced. Additionally, among the similar pairs, half are benign pairs while the other half are adversarial.

B. Evaluation Metrics

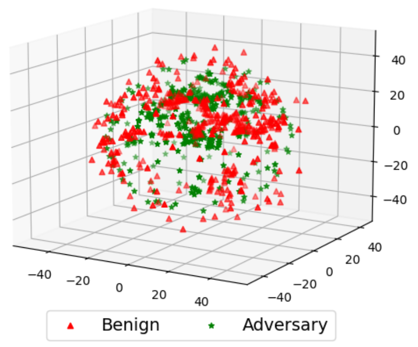
$Precision$, $Recall$, F_1 $score$, and $Accuracy$ are commonly used metrics in the field of machine learning to evaluate the performance of classification models. $Precision$ is a measure of how many of the positively predicted instances are actually true positives. High precision indicates that the model has a low rate of false positives. False positive is the number of predicted similar pairs that actually are not. $Recall$, aka True Positive Rate (TPR), is the measurement describing how robust the model is in identifying malicious attacks. A high $Recall$ means the method can effectively distinguish a benign event from an adversary one. F_1 $score$ is the harmonic mean between $Precision$ and $Recall$. A higher F_1 $score$ implies a lower false positive rate as well as a lower false negative rate. $Accuracy$ is the ratio of the number of correct predictions to the total number of input samples. Thus, a high $Accuracy$ means that the model performs well overall.

TABLE III: The datasets consisting of similar and dissimilar event pairs

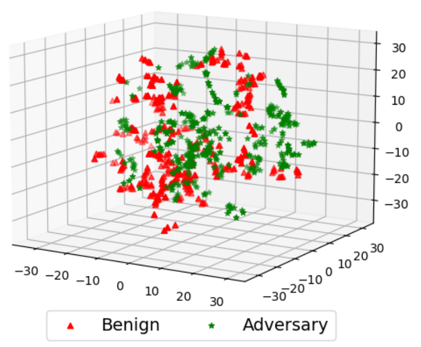
Dataset	D01∼D05	D06	D07	D08	D09	D10	D11
Source	C01∼C05	C06	C07	C08	C09	C10	C11
# of pairs	3,380	516	92	2,648	332	148	4,164

C. Embedding Visualization and Model Interpretability

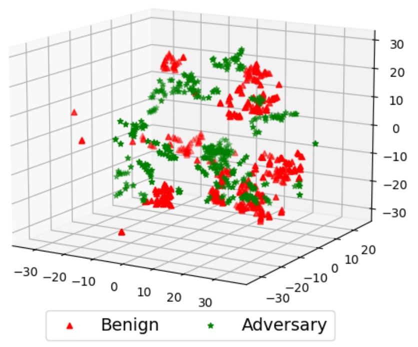
Intuitively, if the numerical representations for events provided by the embedding model are semantically meaningful, the degree of similarity between events can be often reflected in the embedding space by the proximity or distance. Therefore, we examine the interpretability of event embeddings generated by Sent2Vec [9] through visualization. In detail, we randomly selected 1,000 samples from D01∼D05, D08, and D11, respectively. The reason we choose these three categories is that there are too few samples in the other sets. We project the event embeddings to a three-dimensional space using t-SNE [15]. As Figure 3 shows, the red points represent benign samples while and green ones represent adversarial samples. It should be noted that the numbers of benign and adversarial samples are balanced. It turns out that benign and adversarial samples are not trivially separable, especially when mixing different attacks together, as Figure 3(a) shows. One explanation is that we compress the high-dimensional space into a three-dimensional space, and due to information loss, the separating hyper-planes between groups that could originally be distinguished are no longer apparent. On the other hand, we can still observe small cliques formed by similar events, which is particularly evident in Figure 3(c). This implies that the existing event embeddings can be used as a feasible starting point. Based on this, we can take advantage of the subsequent FSL-based neural network model to further amplify the distance between benign and adversary events in a feature space. At the same time, the distance between two



(a) D01∼D05



(b) D08



(c) D11

Fig. 3: Visualization of initial event embeddings based on Sent2Vec [9].

benign events, and the distance between two adversarial events can both be reduced.

D. *Effectiveness*

As previously mentioned, to address real-world cyber incidents in cloud environments, we are particularly interested in an effective threat detection system capable of detecting previously unseen APT attacks. Ideally, the knowledge acquired from a few known attack scenarios is supposed to generalize over other unseen attacks. To validate our hypothesis, we divided the data into 7 subsets based on different attacks, as shown in Table III. After that, we conducted four groups of experiments, from Group 1 to Group 4, where the size of the training set gradually increases in each group. Models trained on known attacks will be used to test unseen attacks. Specifically, we will use the metrics described in Section V-B to evaluate the effectiveness of the attack detection models.

We perform the evaluation in multiple groups as we change the size of the training set and the testing set. Table IV summarizes our main results. Group 1 uses five cases from D01 to D05. We also varied the group by including additional data cases in the training set. Groups 2, 3, and 4 respectively have six, seven, and eight cases in the training set. The remaining unused cases in the training stage are evaluated as the testing set. The FSL-based method we proposed has shown promising results overall, achieving an average accuracy of 91.7% across 18 tests.

E. *Case Study*

In general, we observed the FSL model performs differently depending on the evaluated data cases as it has more cases in the training set. Some cases like D09 have better accuracy, precision, and recall in the Group 4 experiment with 8 cases in the training. This is typically in line with our expectations, *"as the model sees more data, it can predict better"*.

Also, we noticed that the performance on the D10 category is often worse than that on other categories. Our manual examination found that the D10 case, an SQL injection case, exhibits more distinct behavior compared to the other attack cases causing less consistent behavior. Other cases include similar behavior, such as command injections and new shell invocations due to the shell code, which are represented as new program invocations even though they are performed by different programs with different exploits. This unique behavior of this particular case made its behavior far from other cases causing unusual behavior.

TABLE IV: Evaluation on unseen attacks

Group	Train	Test	Accuracy	Precise	Recall	F_1
G1	D01∼D05	D06	88.2%	93.0%	82.6%	87.5%
	D01∼D05	D07	91.3%	95.2%	87.0%	90.9%
	D01∼D05	D08	92.9%	88.9%	98.2%	93.3%
	D01∼D05	D09	94.0%	90.1%	98.8%	94.3%
	D01∼D05	D10	85.8%	87.3%	83.8%	85.5%
	D01∼D05	D11	94.7%	91.2%	99.0%	95.0%
G2	D01∼D06	D07	91.3%	93.2%	89.1%	91.1%
	D01∼D06	D08	95.1%	92.1%	98.7%	95.3%
	D01∼D06	D09	95.5%	98.8%	92.7%	95.6%
	D01∼D06	D10	82.4%	87.5%	75.7%	81.2%
	D01∼D06	D11	96.2%	94.3%	98.3%	96.3%
G3	D01∼D07	D08	93.2%	88.7%	98.9%	93.5%
	D01∼D07	D09	94.0%	89.2%	100%	94.3%
	D01∼D07	D10	83.8%	85.7%	81.1%	83.3%
	D01∼D07	D11	96.0%	93.4%	98.9%	96.1%
G4	D01∼D08	D09	96.4%	93.3%	100%	96.5%
	D01∼D08	D10	83.1%	90.2%	74.3%	81.5%
	D01∼D08	D11	96.6%	95.3%	98.0%	96.6%

VI. Related Work

A. *Provenance Analysis*

Using provenance in intrusion analysis and detection has been explored by a large body of work [16]. Dependence tracking analysis [17] has been used to analyze a large volume of data effectively. Provenance tracking has been done in different data granularity. BEEP [18] and ProTracer [19] use units that are execution partitions of application code which is common in event-handling loops. MPI [20] uses user input on data structures to define execution partitions. PrioTracker [4] proposed priority-based causality tracking using rareness score and fanout score as indications of unusualness. Bates et al. [21] proposed Linux Provenance Modules (LPM), a kernel-based framework and data loss prevention system for sensitive data. PalanTir [22] uses a processor tracing (PT) hardware technique to enable finer-grained instruction level tracking.

KAIROS [23] proposed a graph neural network-based encoder and decoder to learn the temporal evolution of the provenance graph's structural changes. We analyze the provenance data using few-shot learning algorithm with the goal of detecting anomalies from the behavior of few samples.

B. Attack detection

Using provenance data for attack detection is a branch of active research. Multiple attack detection approaches have been proposed based on dependency graphs. HOLMES [24] proposed an approach to detect and summarize APT attack campaign stages. SLEUTH [25] proposed a tag-based attack detection system to prioritize behavior. Hossain et al. [26] improved a tag-based system reducing false alarms significantly in the detection of APT-style attacks. NODOZE [5] uses a network diffusion algorithm that propagates anomaly scores across dependency graphs to calculate anomaly scores. Hassan et al. [6] propose another graph-based scoring scheme for an alert triage system with path preferences and graph reduction schemes. In this work, we propose using Few-Shot Learning on the provenance data due to the growing diversity of attack mechanisms. The presented results shed light on a promising direction for scenarios where adversary data is inadequate.

VII. CONCLUSION

System provenance analysis is believed to be a promising mechanism. Due to its remarkable ability to track dependencies across system events in a cyber incident, this potent technique can play a crucial role in the analysis of incidents as well as in detecting attacks. While various attack patterns are possible, having a limited set of attack datasets becomes a challenge. In this paper, we propose a new approach that applies few-shot learning to attack detection based on system events. Our evaluation of a set of 11 different attack scenarios shows a promising result of 91.7% accuracy on average from 18 experiments with varied training and testing sets, demonstrating that the threat prediction on different attacks with few data samples is possible.

ACKNOWLEDGMENT

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This article describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the article do not necessarily represent the views of the U.S. Department of Energy or the United States Government. This work was supported through contract CR-100043-23-51577 with the U.S. Department of Energy.

REFERENCES

[1] IBM Security, "Cost of a data breach report," https://www.ibm.com/reports/data-breach, 2022, accessed: 2023-05.

[2] Check Point Research, "Cyber security report," https://resources.checkpoint.com/cyber-security-resources/2023-cyber-security-report, 2023, accessed: 2023-05.

[3] Z. Li, Q. A. Chen, R. Yang, Y. Chen, and W. Ruan, "Threat detection and investigation with system-level provenance graphs: a survey," *Computers & Security*, vol. 106, 2021.

[4] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *NDSS*, 2018.

[5] W. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting threat alert fatigue with automated provenance triage," in *NDSS*, 2019.

[6] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *IEEE S&P*, 2020.

[7] Y. Tang, D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li, "NodeMerge: Template based efficient data reduction for big-data causality analysis," in *CCS*, 2018.

[8] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *NeurIPS*, 2013.

[9] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-gram features," in *NAACL*, 2018.

[10] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International Conference on Machine Learning*, 2014.

[11] C. Fu, X. Du, Q. Zeng, Z. Zhao, F. Zuo, and J. Di, "Seeing is believing: Extracting semantic information from video for verifying iot events," in *WiSec*, 2024.

[12] F. Zuo, B. Yang, X. Li, and Q. Zeng, "Exploiting the inherent limitation of L0 adversarial examples," in *RAID*, 2019.

[13] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *CVPR*, 2006.

[14] M. Shrestha, Y. Kim, J. Oh, J. Rhee, Y. R. Choe, F. Zuo, M. Park, and G. Qian, "ProvSec: Open cybersecurity system provenance analysis benchmark dataset with labels," *International Journal of Networked and Distributed Computing*, vol. 11, no. 2, pp. 112–123, 2023.

[15] L. Van der Maaten and G. Hinton, "Visualizing data using t-SNE." *Journal of Machine Learning Research*, vol. 9, no. 11, 2008.

[16] M. Zipperle, F. Gottwalt, E. Chang, and T. Dillon, "Provenance-based intrusion detection systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, 2022.

[17] S. T. King and P. M. Chen, "Backtracking intrusions," in *The 19th ACM Symposium on Operating Systems Principles*, 2003.

[18] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *NDSS*, 2013.

[19] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *NDSS*, 2016.

[20] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *26th USENIX Security Symposium*, 2017.

[21] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the Linux kernel," in *24th USENIX Security Symposium*, 2015.

[22] J. Zeng, C. Zhang, and Z. Liang, "Palantír: Optimizing attack provenance with hardware-enhanced system observability," in *CCS*, 2022.

[23] Z. Cheng, Q. Lv, J. Liang, Y. Wang, D. Sun, T. Pasquier, and X. Han, "KAIROS: Practical intrusion detection and investigation using whole-system provenance," in *IEEE Symposium on Security and Privacy (SP)*, 2024.

[24] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: Real-time apt detection through correlation of suspicious information flows," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[25] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *26th USENIX Security Symposium*, 2017.

[26] M. N. Hossain, S. Sheikhi, and R. Sekar, "Combating dependence explosion in forensic analysis using alternative tag propagation semantics," in *IEEE Symposium on Security and Privacy (SP)*, 2020.