Pagoda: A Hybrid Approach to Enable Efficient Real-time Provenance Based Intrusion Detection in Big Data Environments

Yulai Xie, *Member, IEEE,* Dan Feng, *Member, IEEE,* Yuchong Hu, *Member, IEEE,* Yan Li, *Member, IEEE,* Staunton Sample, Darrell Long, *Fellow, IEEE*

Abstract—Efficient intrusion detection and analysis of the security landscape in big data environments present challenge for today's users. Intrusion behavior can be described by provenance graphs that record the dependency relationships between intrusion processes and the infected files. Existing intrusion detection methods typically analyze and identify the anomaly either in a single provenance path or the whole provenance graph, neither of which can achieve the benefit on both detection accuracy and detection time. We propose Pagoda, a hybrid approach that takes into account the anomaly degree of both a single provenance path and the whole provenance graph. It can identify intrusion quickly if a serious compromise has been found on one path, and can further improve the detection rate by considering the behavior representation in the whole provenance graph. Pagoda uses a persistent memory database to store provenance and aggregates multiple similar items into one provenance record to maximumly reduce unnecessary I/O during the detection analysis. In addition, it encodes duplicate items in the rule database and filters noise that does not contain intrusion information. The experimental results on a wide variety of real-world applications demonstrate its performance and efficiency.

Index Terms—Provenance, Intrusion Detection, Big Data, Real-time

1 Introduction

Host-based intrusion detection has long been an important measure to enforce computer security. In today's world, the cyber attack has become a persistent, aggressive and disruptive threat. For instance, the APT (Advanced Persistent Threat) attack has gradually become a main threat in enterprise's environment [1, 2]. The "WannaCry" virus has attacked nearly 100 countries in the world [3] and resulted in huge economic losses in 2017 [4].

The traditional intrusion detection system typically uses system calls to analyze and identify host-based intrusion [5–10]. However, these methods are not widely used. Since they do not disclose how the intrusion happens, and thus the detection accuracy is not high. With the stealth and sophistication of modern attacks, it's critical to identify the causality relationships between the intruder and the damaged files. The existing mainstream methods focus on offline forensic analysis using provenance [11, 12] or audit logs [13–15]. However, typical attacks such as APT can remain stealthy for half a year after getting into the enterprise [16]. It is too late if sensitive data have been stolen before disclosing the intrusion source.

However, it is a challenge to accurately acquire the causality relationships of the intrusion behaviors and identify the intrusions in real time especially in today's big data environments, where the intruders' illegal behavior data are buried in massive data of different users and different applications. In previous work [17], we developed PIDAS, a provenance path based intrusion detection and analysis system. It uses provenance information but not traditional syscall as the data source for online intrusion detection. Provenance represents the history of an object, and records the dependencies of infected files, intrusion processes and network connections at the time of intrusion. By computing the anomaly degree of a certain length of a path that consists of a series of dependency relationships and comparing it to a predefined threshold, PIDAS can judge whether the intrusion has happened in real-time. The drawback of this method lies in that using only one path to detect intrusions cannot reflect the behavior of a whole provenance graph. Typically, the system cannot easily identify an intrusion where a virus stealthily infects all the paths with no serious damage. Though the administrator can reduce false alarms via analyzing the warning report, it is time-consuming and the administrator may make a wrong judgement.

There are also emerging works that use the whole provenance graph to detect intrusion. For instance, Lemay et al. [18] proposed a series of rule grammars to mine and judge the anomaly in the provenance graph of application behavior. Han et al. [19] used the provenance graph of a subset of a program's provenance records to model the application behavior in the PaaS cloud. Though these methods may have better detection accuracy than PIDAS, they have not been demonstrated on a variety of security-critical applications. In addition, analyzing the whole provenance graph usually involves traversing and processing a large number of provenance nodes and edges. This introduces a large runtime overhead and thus may not be realistic in big

- *Yulai Xie, Dan Feng and Yuchong Hu are with the School of Computer, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, P.R. China.*
E-mail: ylxie@hust.edu.cn, dfeng@hust.edu.cn, yuchonghu@hust.edu.cn
- *Yan Li is with TuneUp.ai in San Francisco Bay Area, CA, USA. Email: yanli@tuneup.ai*
- *Staunton Sample and Darrell Long are with Jack Baskin School of Engineering, University of California, Santa Cruz, CA 95064 USA. Email: sbsample@ucsc.edu, darrell@ucsc.edu*

data environments.

To address the above problems, we propose Pagoda, a provenance based intrusion detection system that analyzes the anomaly degree of not only a single path, but also the entire provenance graph. It first looks for the intrusion path that may result in the intrusion. If the path has been found, then it does not have to traverse the provenance graph for further detection. Otherwise, it computes the anomaly degree of the provenance graph in three steps. First, it computes the anomaly degree of every path. Then it multiplies the anomaly degree of a path it by its path length to get the weight value of every path. Finally, it uses the sum of all these weight values to divide by the sum of the lengths of all the paths. This quickly identifies when an intrusion process inflicts damage on only a sensitive file or a small subset of files in the system. It also further improves the detection accuracy when all the files in the system have been stealthily damaged.

In addition, unlike the traditional approaches that employ GPU [20] or high performance CPU to improve the detection performance, Pagoda utilizes the widely used key-value memory database to speed up the detection process without adding the hardware cost by reducing the disk IO in detection process. On one hand, both the rule database and the provenance to be processed reside in memory. So the provenance acquirement and rule queries do not need to redirect to the disk. On the other hand, Redis aggregates multiple values with the same key into one value. This significantly reduces the provenance query time. Especially when we start with a process or file for ancestor queries in a provenance graph, we can fetch all the ancestors at one time.

Moreover, like many provenance systems (e.g., CamFlow [21]), Pagoda filters unnecessary provenance data to reduce the detection time. This also prevents noisy data generating false alarms. Typical noisy data includes daemon processes, pipe files and temporary files that are not likely to contain intrusion information. As Pagoda mainly uses the dependency relationships between different objects to drive the intrusion detection algorithm, it also omits some provenance data (e.g., environment variables and input parameters) to save the memory space. In addition, as we use an absolute path name to describe a file or a process, files in the same directory have a common prefix in their names. Pagoda uses dictionary encoding [22] technology to compress these duplicates to further reduce the space overhead.

The contributions of this paper are as follows:

- We propose Pagoda, a provenance-based intrusion detection system that takes into account the anomaly degree of both a single path and the whole provenance graph to achieve both fast and accurate detection in big data environments.
- We incorporate a novel design into Pagoda that uses a persistent memory database to store provenance and aggregate multiple similar provenance records into one record to maximumly reduce the unnecessary I/O and improve the provenance query efficiency in both the online detection and forensic analysis processes.
- To further save the memory space, we apply dictionary encoding to reduce the replicated items in the rule database. Moreover, we filter the noise provenance that is not likely to contain intrusion information or is not used for detecting intrusions in our method. Thus we improve the detection accuracy and reduce the detection time.
- We implement the system prototype and evaluate it on a series of real-world normal and vulnerable applications. The experimental results show that Pagoda significantly outperforms the classical syscall-based method [5] and the state-of-the-art (i.e., provenance path based method [17]) on a series of critical axes, such as detection accuracy, detection time, forensic analysis efficiency and space overhead.

Assumptions and Limitations: As the provenance collection process is in the kernel, we assume the operating system is trusted. In fact, there are some existing trusted computing platform [23] and sophisticated security schemes [24] that we can use to provide integrity, confidentiality and privacy guarantees to prevent undetected provenance modification. As provenance mainly stores in memory, we also assume the execution does not crash during detection. Even in the worst case, the provenance in memory can be batch-loaded to disk before the memory crashes.

Pagoda cannot track the intrusions that do not go through syscall interface because the intrusion behaviors in this case do not produce provenance. Almost all the user behavior in the computer system can generate system calls, such as reading or writing files, sending or receiving data to/from the network. Only a very few cases, such as memory leak, do not generate system calls. Typically, the data leak from the memory that results from the OpenSSL heartbleed vulnerability (CVE-2014-0160) does not produce provenance and thus cannot be captured. In addition, though Pagoda prunes provenance (e.g., temporary files or pipe files) to save space and speed up detection, there may still be a possibility that some intrusions are propagated via these noisy provenance.

2 Background and Motivation

We first give an overview of provenance and PASS [25] model, then introduce PIDAS [17].

2.1 Provenance

Provenance, also known as the origin of data, records how data was generated and how it comes to be its current state [26]. Provenance is commonly represented as a directed acyclic graph (DAG) [27], where the nodes represent objects, and the edges represent the dependency between the objects. For instance, a read system call on a file will construct a directed edge indicating that this read process depends on this file. In addition, each node has corresponding attributes to describe its semantics. For instance, a file may have attributes such as name and inode number, while a process can have attributes such as process name and ID.

As provenance has been widely used in a variety of areas (e.g., experimental document, search [28], and security [14]), a number of provenance collection and tracking systems have been built. The typical systems include SPADE [29], Story Book [30], TREC [31], PASS [25], LinFS [32], Hi-Fi [33], LPM [34], CamFlow [21], and Eidetic system [32]. These systems collect provenance in different layers. For instance, SPADE and Story Book collect provenance in the application layer, whereas PASS and LinFS can track kernel-level provenance. In addition, TREC is designed to collect provenance locally, SPADE can collect provenance in a distributed environment, whereas PASS can be extended to provide back-end storage support in both network-attached [35] and cloud [36] environments.

The intrusion detection system we develop in this paper is built on PASS which is a storage system to collect provenance information automatically. Provenance collection is transparent to the user layer and intruders do not know whether the system is monitoring their behavior. PASS intercepts the syscalls and transforms them to the causality-based provenance graphs, then stores the provenance graphs in key-value databases.

In the PASS model, the nodes in a provenance graph mainly include files, processes, and pipes. Though PASS collects provenance in the kernel, it also allows an application to generate its own provenance information. That means, the node can be an application-defined object. For each node, PASS assigns a unique pnode number to identity it, and the pnode number is monotonically increasing whenever a new node is created. Since an object may be written multiple times, the system also assigns a version number to each node to avoid the occurrence of the cycle in the provenance graphs.

2.2 PIDAS

Unlike traditional host-based intrusion detection methods that judge intrusion by finding the anomaly in the syscalls or UNIX shell commands, PIDAS [17] is the first work that identifies intrusion online by looking for provenance proof to detect whether a certain behavior is anomalous.

PIDAS employs a file-level provenance tracking framework (e.g., PASS) to collect provenance information generated by the normal behaviors of a program, filters the noisy provenance data (e.g., the temporary files or pipe files) and then divides the pruned provenance into a series of dependency relationships between files, processes and sockets. The frequently generated dependency relationships during multiple runs of a program will be put into a BerkeleyDB rule database called G.

Then, during the detection phase, PIDAS also filters the noisy provenance data that is not likely to contain intrusion information and then extracts all the dependency relationships from the intrusion behavior. If a dependency belongs to G, then the anomaly degree of this dependency is regarded as 0, otherwise it is 1. We make a depth-first search in the provenance graph of the intrusion behavior, and find the path whose length is L.

We calculate the path decision value P as follows:

$$P = \frac{\sum_{i=1}^{L} \text{anomaly degree of each edge}}{L}$$

We set the decision threshold as T. If $P > T$, the program behavior is judged as anomaly.

However, this method still has the following shortcomings:

a) Though provenance data has been reduced to eliminate the noisy data (e.g., the temporary files or pipe files), there still exist a lot of duplicates in the rule database, especially when a file or process is represented as an absolute path and occurs frequently in the key or value in the rule database.

b) Because the rule database and the provenance to be detected are stored in the form of database files on the hard disk, the disk I/O overhead will have a great impact on the detection time.

c) Only using a single provenance path to detect intrusion cannot reflect the behavior of a whole provenance graph, and thus the detection precision can be further improved. Typically, the intruders' behaviors in many cases (e.g., APT attack) are complicated, such as browsing multiple directories, tampering with the system files and sending various sensitive data outside. The provenance graph for describing these behaviors often involves many branches, and thus a single path cannot completely represent the behaviors of the intruder.

3 Design and Implementation

We will first describe the design goals of our provenance-based intrusion detection system, then we elaborate the details on design and implementation of this system.

3.1 Design goals

Our intuitive design objective mainly comes from the practical user requirements of host-based intrusion detection system: detection accuracy, real time, and low overhead.

a) Detection accuracy

Obviously, the first and most important goal is to detect with high accuracy. We will mainly consider two aspects: detection rate and false alarm rate. The former shows the percentage of intrusion behaviors that are correctly classified. The latter reveals the percentage of normal behaviors that are reported as intrusions.

b) Real time

In today's big data era, increasingly large data has been generated and should be processed in time. Especially for an intrusion detection system, finding intrusion or anomaly from data flow with multi-source has become a great challenge. When using provenance or audit logs to record the user or application behavior, it is important to process the data in a timely and efficient manner.

c) Low overhead

The overhead includes three aspects in this paper: 1) disk space overhead used to build various provenance databases; 2) memory overhead incurred by building memory databases and storing and processing provenance data; 3) The performance overhead brought by the running of intrusion detection algorithm, i.e., the intrusion detection algorithm should have a minor impact on the normal program running.

3.2 Overall architecture

Figure 1 shows the architecture of Pagoda. It consists of six components, namely, *Provenance collection, Provenance pruning, Provenance storage and maintenance, Rule building and deduplication, Detection process* and *Forensic analysis*. The

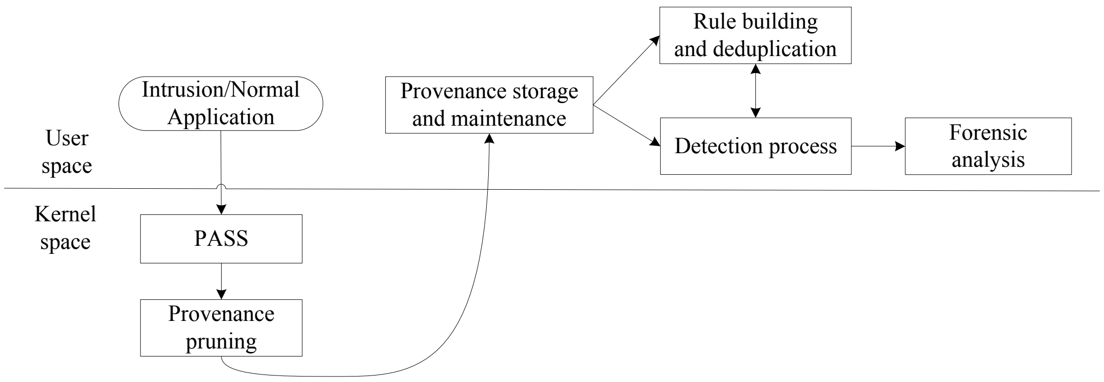

Rule building and deduplication
Intrusion/Normal Application
Provenance storage and maintenance
User space
Detection process
Forensic analysis
Kernel space
PASS
Provenance pruning

Fig. 1: Architecture of Pagoda

Provenance collection component is responsible for monitoring the behaviors of the normal/intrusion applications, intercepting the system calls invoked by them and translate these system calls to causality-based provenance records. Then the *provenance pruning* module omits the provenance records that are not related to intrusion detection to improve the detection accuracy and save the storage space simultaneously. The *Provenance storage and maintenance* component uses key-value memory database (e.g., Redis [37]) to store rule database and run the provenance-based intrusion detection algorithm to make real-time detection. The *Rule building and deduplication* module constructs the rule sets for intrusion detection and removes the duplicated strings to make the rule database as small as possible. The *Detection process* component judges whether the intrusion has happened according to the rule sets and also updates the rule sets according to the detection results. At last, the *Forensic analysis* module looks for the system vulnerability and intrusion sources by making forward and backward queries.

3.3 Provenance collection and pruning

Pagoda utilizes PASS to collect provenance of intrusion/normal applications. It can also use other provenance tracking frameworks (e.g., SPADE, LinFS) which intercept syscall and generate file-level provenance.

Pagoda prunes noisy provenance from two aspects. First, similar to PIDAS, Pagoda does not preserve the provenance of objects that only reside on disks for a short term. Typically, these objects include temporary files or pipe that occur during program execution. They just bridge the information transformation between different entities (e.g., files or processes), but are not likely to store the intrusion information. Second, to further save storage space and improve detection efficiency, Pagoda does not collect the whole-system provenance, but only chooses the key data that is used for detecting intrusions from the provenance stream. For instance, both PASS and PIDAS collect a variety of provenance items, e.g., attributes of an object, like NAME, TYPE, ENV(i.e., environment variable), ARGV (i.e., argument input of a process), PID, EXECTIME; the dependency relationships between objects, such as INPUT, GENERATEDBY, FORKPARENT, RECV, SEND. But for Pagoda, only the name of the objects and the dependencies between them are required for detecting intrusions. Pagoda does not need to preserve the type of object, the environment variables of the system or any other information. For this, we add a filter (i.e., Provenance pruning component) in the framework to eliminate the noisy data and make the detection more efficient.

3.4 Provenance storage and maintenance

To reduce detection time, Pagoda stores provenance in non-volatile memory databases. Originally, PASS stores provenance in many log files or BerkeleyDB databases that reside on disks. This will induce a large number of disk Input/Output operations that slow down the provenance query and intrusion detection process. Therefore, Pagoda stores rule database in an emerging and widely used memory key-value database called Redis. Thus any updates in the rule database will only be in memory.

Pagoda uses a series of Redis databases to store provenance as shown in Table 1. Each node is uniquely identified by a pnode number. NameDB builds the mappings from the pnode number of a node to its name. RuleDB contains the frequently generated relationships between the name of a node and its parent node during the training process. In addition, ParentDB and ChildDB are used as index databases to store the relationships between the pnode number of a node and its parent nodes or child nodes respectively. Note that we store multiple provenance dependency relationships into one item in Redis to further reduce the provenance graph query time. In a provenance graph that describes intrusion behavior, there exist many edges that start from the same nodes. For instance, an intrusion process may access many files, and will form many edges from this process to each file. The traditional key-value database (e.g., BerkeleyDB) will store each edge as a key-value item in the database. It takes a long time to locate all of these items in the graph query process. Pagoda stores these kinds of dependency relationships into one item in Redis where the key is the process and the value is a collection of files. This will improve detection performance and reduce the number of items in the rule database simultaneously.

As provenance is in Redis, it will not disappear in case of power crash. The provenance in memory can be batch-loaded to logs on disk periodically. This further enforces the reliability of provenance.

TABLE 1
Provenance database

Database	Key	Value
NameDB	Pnode number	Pnode name
ChildDB	Parent pnode	Child pnodes
ParentDB	Child pnode	Parent pnodes
RuleDB	Child name	Parent name

3.5 Rule building and deduplication

The building of a rule database in Pagoda is similar to PIDAS, i.e. the system obtains provenance information of the normal user behavior, and extracts the dependency relationships to create a rule database with the following steps:

1. To accurately track and obtain provenance information of the normal behaviors of a program, we run it M times.
2. For each time of running, the program behavior can be described as a provenance graph which consists of a series of dependency relationships which we denote as $\text{Dep}_1, \text{Dep}_2, ..., \text{Dep}_n$. Dep_i represents a directed relationship between two objects. Typical objects include files, processes, and sockets. If object A depends on another object B, we denote it as $\text{Dep}_i = (A, B)$, of which A is the child and B is the parent.
3. We set a threshold T_1 and count the number of times N_i that Dep_i appears in M times of program running. If $N_i > T_1$, we put one copy of the corresponding Dep_i into the rule database G, $G = \{\text{Dep}_i | N_i > T_1\}$.

We use an absolute path to describe the name of an object in the rule database. The absolute path can provide exact location to find a file, especially when we need to remove a malicious file. However, this can also bring in a lot of duplicate information. The first case is that some strings are exactly the same. Typically, if we put both "$A \to B$" and "$B \to C$" in the rule database, B needs to be stored twice. When B is a long string, it takes a lot of space to store many strings such as B. The second case is that a large number of strings have only a few differences. Most of the different parts of similar strings appear at the end of the path. This is because intruders are likely to access different files in the same folder. In this case, the intrusion process may rely on these files from the same folder. The names of these files have a common prefix. We use dictionary encoding to compress these duplicate strings, which will be encoded as integers.

3.6 Detection process

PIDAS detects intrusion by judging whether a fixed length (i.e., L) of a path is abnormal. However, if L is much smaller than the actual length of a path, the limited length cannot actually reflect the whole intrusion behavior along the complete path. Thus it cannot represent the intrusion behavior that is described as a provenance graph. We can propose PIDAS-Graph, a method that considers the entire provenance graph to improve the detection accuracy. Similar to PIDAS, PIDAS-Graph also filters the provenance data (such as temporary files or pipe) that are not likely to contain intrusion information, and then stores the pruned provenance data into BerkeleyDB databases. The algorithm of PIDAS-Graph is as follows:

1) Computing the anomaly degree (i.e., path decision value) of each complete path using the algorithm in Section 2.2.
2) As the anomaly degree of each provenance path can have different impact on the anomaly degree of the whole provenance graph, we assign each provenance path a weight value W according to the length of the path. Let the anomaly degree of each path in a provenance graph be $P_1, P_2, ..., P_n$, the lengths of these paths be $L_1, L_2, ..., L_n$, the corresponding W_i $(1 \leq i \leq n)$ is calculated as: $W_i = L_i/(L_1 + L_2 + ... + L_n)$, the anomaly degree Q of the whole provenance graph is calculated as follows: $Q = P_1 \times W_1 + P_2 \times W_2 + ... + P_n \times W_n = (P_1 \times L_1 + P_2 \times L_2 + ... + P_n \times L_n)/(L_1 + L_2 + ... + L_n)$.
3) We set the graph threshold as T. If $Q > T$, then the behavior that forms this provenance graph is judged as intrusion.

The algorithm first computes the anomaly degree of each path based on the anomaly degree of each edge on this path, which has been outlined in Section 2.2. Then it multiplies the length of each path by its anomaly degree to get a weight value, and then calculates the anomaly degree of the whole provenance graph by using the sum of all these weight values to divide by the sum of the lengths of all the paths. When the anomaly degree is bigger than a predefined threshold, the system is judged to have been attacked. However, it is time-consuming to calculate the anomaly degree of the whole provenance graph every time. Especially when the provenance graph is big enough to hold hundreds or thousands of paths, detection time can be very long.

Though counting the anomaly degree of only one path can lead to low accuracy, we find that detection accuracy can still be high if the anomaly degree of the path is high enough. For example, if the length of a path is 4, and 3 or 4 edges (or dependencies) on this path are not in the rule database, these edges are likely to be generated by the intrusion behavior. We set the anomaly degree of each edge that does not occur in the rule database as 1, otherwise we set its anomaly degree as 0. The path in which the anomaly degree of most of the edges is 1 can be identified as the invasion path and the whole program behavior can be judged as intrusion behavior. This can be very common in today's intrusion attack. For instance, in an APT attack, the clever intruders can hide their behaviors in a bunch of normal behaviors to evade the detection. If we compute the anomaly degree of the entire provenance graph, as most of the paths have very low anomaly degree, this kind of attack may not be easily identified.

So this work develops Pagoda to detect intrusion by taking into account the anomaly degree of both path and graph. The basic idea behind Pagoda is to identify intrusion in a large bunch of data in real time and accurately. The basic approach is that we first quickly detect the intrusions by analyzing and locating the path with high anomaly degree. We can then calculate the anomaly degree of the whole provenance graph based on the length and anomaly degree of all the paths to further improve detection accuracy if necessary.

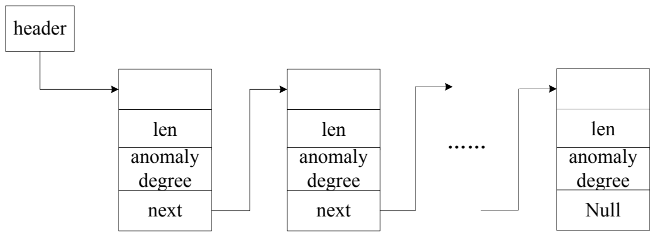


Fig. 2: Data structure of list_path that stores the length and anomaly degree of each path in a provenance graph

The whole workflow of the detection process is shown in Algorithm 1. The algorithm first traverses all the head nodes (i.e., all nodes in the graph that have no parent nodes) which can be acquired by searching for the ParentDB using Algorithm 2 and Algorithm 3. For each head node, it searches for all the paths starting from itself, and then stores the anomaly degree and length of each path to the list_path as shown in Figure 2. If the anomaly degree of any path exceeds a predefined threshold, the algorithm will be terminated and sound an alarm. The administrator can further analyze whether the intrusion has really happened and update the rule database when the false alarm happens. If no alarm is sounded, the anomaly degree of the whole provenance graph will be computed by first calculating the whole length of all the paths and the sum of the anomaly degree of all the paths based on their lengths, and then using the latter to divide by the former. If the anomaly degree of the graph exceeds a predefined threshold, then the alarm will be sounded.

3.7 Forensic analysis

After judging intrusion online, the administrator can further make forensic analysis to identify what has happened to a system. She can use traditional tools such as Tripwire to find a detection point (e.g., a damaged file or a suspicious process), and then use this detection point as keyword to query in the memory database to make forensic analysis. The detection point can be also acquired by analyzing the abnormal edges in a provenance path when the anomaly degree of this path exceeds the predefined threshold. The forensic analysis mainly includes two steps: backward query and forward query. Backward query is used to query the system vulnerability and the source of the intrusion. Forward query is used to query all the intrusion behaviors of the attackers. The integration of backward query with forward query can construct the provenance graph of an intact invasion process.

As provenance data resides in memory, it is efficient to make queries for forensic analysis. After the system vulnerability or intrusion sources are ascertained, the administrator can patch the defect on the system software or improve the system security level.

4 Evaluation

We first describe the testbed and data sets we use in the experiment. Then we evaluate the system by comparing it with other classical intrusion detection systems on a series of critical axes such as detection rate, false alarm rate, detection time, query time and storage overhead.

Algorithm 1 Pagoda: Provenance-based intrusion detection that considers the anomaly degree of both path and graph

```
Input: list_head_node /// A list that stores all head nodes
Output: alarm
1:  for each head node p in list_head_node do
2:      for each path that starts from p do
3:          set the anomaly degree of each edge in the path;
4:          compute the anomaly degree of the path;
5:          if anomaly_degree_path > path_threshold then
6:              call alarm;
7:              return;
8:          endif
9:          store the length and the anomaly degree of the path into list_path;
10:     endfor
11: endfor
12: for each $\langle len, anomaly\_degree \rangle$ in list_path do
13:     total_len=total_len+len;
14:     anomaly_degree_graph = anomaly_degree_graph + len*anomaly_degree;
15: endfor
16: anomaly_degree_graph = anomaly_degree_graph/total_len;
17: if anomaly_degree_graph > graph_threshold then
18:     call alarm;
19:     return;
20: endif
```

Algorithm 2 Get_head_list

```
Input: ParentDB
Output: The list of head records
1: for each record $\langle cname, pname \rangle$ in ParentDB do
2:     Get_head_node(pname);
3: endfor
```

Algorithm 3 Get_head_node(pname)

```
Input: pname
Output: The list of head nodes for pname
1: Using pname as keyword to search for parent in ParentDB;
2: if parent does not exist then
3:     add pname into list_head_node;
4:     return;
5: else
6:     Get_head_node(the parent of pname);
7: endif
```

4.1 Experimental setup

The experimental host machine runs Windows 8.1 Pro 64-bit operating system, with four Intel(R) Core(TM) i7-6700 CPU @3.40GHz and 32 GB memory. The virtual machine on it runs Pagoda intrusion detection framework. It installs Ubuntu 16.04.1 operating system, with four processors, 16 GB memory, and 60 GB hard disk.

TABLE 2
Basic Descriptions for a Variety of Normal and Vulnerable Applications Used in Intrusion Detections.

Application	Description	# of traces	# of nodes	# of relationships	Training set	Test set	
					# of normal traces	# of normal traces	# of intrusion traces
blast-lite	A simple instance of the Blast biological workload	1	350	628	1	1	0
postmark	The PostMark file system benchmark	1	7818	4777	1	1	0
elaine-oct25	A researcher developed a python application and wrote a conference paper	47	117575	292432	40	7	0
linux-apr13	Build of the Linux kernel	15	138285	1355651	7	5	0
am-utils	Compilation of am-utils	1	83524	195579	1	1	0
patch-apr17	Patching the Linux kernel	2	131981	73387	1	1	0
NetBSD	Build of several components of NetBSD	616	10193304	17062049	49	50	0
firefox	A web browser	200	183053	269658	100	100	0
vsftp (CVE-2011-2523)	A secure FTP server	451	35094	41967	350	48	47
samba (CVE-2007-2447)	A program that implements the SMB (Server Messages Block) protocol and provides file sharing services	180	25661	36283	90	50	40
distcc (CVE-2004-2687)	A distributed, C++ compiler tool	130	465887	1029087	30	50	50
flash (CVE-2008-5499)	Adobe flash player, a runtime that executes and displays content from a provided SWF file	300	311874	581594	200	50	50
proftp (N/A)	Highly configurable GPL-licensed FTP server software	204	22844	23648	78	24	102
ptrace (CAN-2003-0127)	A tool that enables a process to control another	250	18779	17373	100	50	100
sendmail (CVE-2002-1337)	An email transfer agent program	238	26339	32331	100	38	100
Web attack	Web application stress tool [38] for benchmarking Apache HTTP server	577	36005184	18038787	_	_	_
phishing email	A user clicks a web page link in an email to open the firefox browser that downloads a malicious backdoor program. The user wrongly executes it that automatically sends sensitive data outside.	400	340518	558958	100	100	200

In order to evaluate the performance and efficiency of the system, a total of seventeen different kinds of applications (see Table 2) were tested. The first eight applications are normal applications, and most of the traces on them are generated by Harvard PASS research group [39] with the exception of the traces of firefox that are collected based on the previous work [40]. The other nine applications are vulnerable to local or remote attacks and the traces on them simulate the normal use of programs and the intrusion against them respectively. Some (e.g., vsftp and samba) are already used in the previous work [17]. Others (e.g., web attack and phishing email [11]) simulate the frequently occurring network attack in today's real world. All these traces represent a wide variety of workloads and vary widely in size and complexity. For instance, blast-lite records the behavior of the biological workload, postmark simulates the small file read/write workload and linux-apr13 and NetBSD both compile the system files or components in different directories. For the vulnerable applications, except the applications that have been identified with CVE numbers, proftp has a backdoor command execution vulnerability in its 1.3.3c version that allows the remote unauthenticated user to access the system. Phishing email simulates a typical APT attack by cheating users to download a malicious trojan via clicking a browser link in the email and then send the sensitive data to a remote host. Web attack simulates the existing web server/application attack in different attack frequencies using the web application stress tool [38]. We mainly use it to measure the provenance growth overhead.

The third column in Table 2 shows the number of traces in each application. The fourth and fifth columns show the total number of nodes and edges (or relationships) in these traces. Note that each node in the graphs is uniquely identified by the combination of a node number and a version number. The sixth, seventh and eighth columns show the number of traces in training sets and test sets respectively. As some applications have only one trace, we use this trace as both training and test set. For the traces in vulnerable applications, both sets cover some common operations: file creation, deletion, download/upload and modification.

4.2 Detection rate and false alarm rate

We evaluate the detection rate and false alarm rate by comparing four intrusion detection methods: the classical system call based method [5], PIDAS [17], PIDAS-Graph, and Pagoda. The parameters of the different methods are as follows: for system call method, the length of the system call sequence is 6, and the threshold is used to compare with the minimal ratio of system calls in a system call sequence that do not occur in each sequence of the rule database; for PIDAS, the path length to be judged is 3 for best result [17]. The detection rate shows the percentage of intrusions that have been correctly classified. The false alarm rate indicates the fraction of normal behaviors that have been judged as anomaly.

We first roughly look at the detection performance of normal traces. Compared to PIDAS, Pagoda has a comparable or better performance (see Table 3). For the applications (e.g., blast-lite, postmark and am-utils) that have only one single trace, both Pagoda and PIDAS have no false alarms. This is because all the test data have occurred in the training data. For patch-apr17, NetBSD, and firefox, Pagoda performs better than PIDAS. The main reason is that Pagoda takes into account both long paths and the whole graph but not a fixed length of a short path as in PIDAS, reducing the potential false alarms. For instance, for patch-apr17, all traces record the patching process of different kernel files. The dependency relationship between a kernel file and the patch process (e.g., `/usr/bin/patch`) in the test trace does not appear in the training trace. Thus false alarm can easily happen for PIDAS when this dependency relationship is in a path of length 3 in case that the threshold is 0.3. As there is only one test trace, so the false alarm rate is 100%. However, Pagoda has a more strict judgement condition, especially it sounds an alarm only when most of the edges in a complete path do not occur in the rule database or the anomaly degree of the whole provenance graph exceeds a predefined threshold. Pagoda has a comparable performance with PIDAS on linux-apr13 and elaine-oct25 traces. The reason that false alarm happens in both of these two cases is that the anomaly degree of a path exceeds the threshold. For instance, the linux-apr13 trace compiles all the files in different directories in a system. As there are no common edges or short paths between the training set and test set, neither PIDAS nor Pagoda correctly classifies the traces in the test set.

For intrusion data sets, we first present the overall results for all intrusion methods and then discuss the performance numbers on individual traces in more detail. To get a picture of how well Pagoda performs on the data sets, we average the performance numbers across all the data sets as shown in Figure 3(a). Each point in the figure denotes the detection performance at a threshold. The points in the top-left corner achieve the best performance on both detection rate and false alarm rate. Pagoda performs best in all the methods. PIDAS and Syscall have big false alarm rate as they judge intrusion by considering only one abnormal provenance path or one system call sequence that does not occur in the rule database. PIDAS-Graph has a low detection rate especially when the graph threshold is big. As the average numbers are heavily influenced by the worst results especially when the false alarms span several orders of magnitude, we also show the median results as shown in Figure 3(b). The median false alarm rates for all the methods have significantly decreased to below 0.04. Pagoda achieves the best performance for most of the cases. When the path threshold is small, the false alarm rates of both Pagoda and PIDAS are slightly big as the provenance paths in this case are easily to be judged as anomaly. This indicates that we can choose a proper threshold to work around it.

The performance for individual intrusion data sets is shown in Figure 4. For each application, we show the average value of each method. Each point in the upper left corner of the figure has both a high detection rate and a low false alarm rate, and thus shows the best performance. Pagoda significantly outperforms other methods in most of the applications. The average detection rates of Pagoda for both distcc and ptrace are not high. This is because intruders have performed a lot of normal behaviors to interfere detection in an invasion. In this case, anomaly degree of a path or a whole graph will both be very low. When the path threshold and graph threshold are both big, intrusion cannot be identified. We can choose proper thresholds to achieve better performance for these applications. The average detection rate of syscall method is even better than that of Pagoda for distcc. The possible reason is that, though there are many normal behaviors in an invasion, the alarm can still be sounded for syscall method when there is an anomalous syscall sequence generated by the abnormal behavior in this attack. The average false alarm rate of Pagoda for sendmail is a little high. This is because its normal behavior that is a little different from the rule database can be easily wrongly judged as anomalous when the graph threshold is small (< 0.5). We can choose a proper

TABLE 3
Comparison Between Pagoda and PIDAS on False alarm rate for all the normal traces.

Application	False alarm rate	
	Pagoda	PIDAS
blast-lite	0%	0%
postmark	0%	0%
am-utils	0%	0%
patch-apr17	0%	100%
elaine-oct25	33.3%	33.3%
linux-apr13	100%	100%
NetBSD	98%	100%
firefox	0%	1%

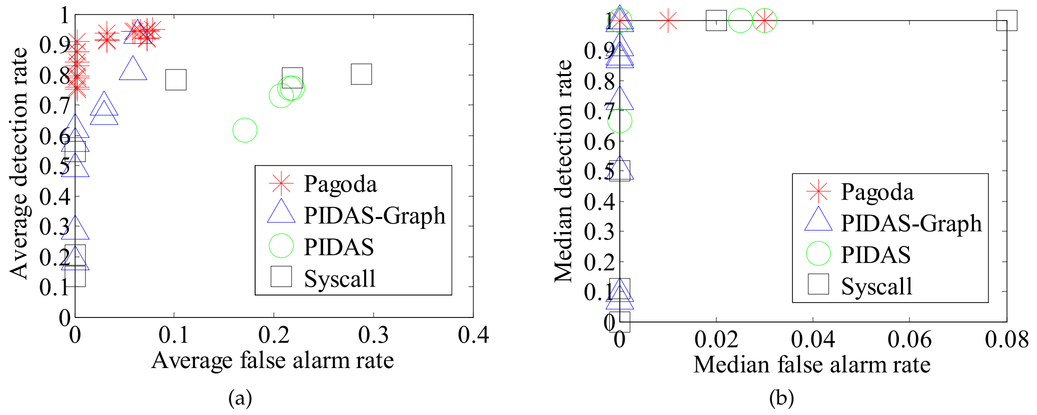


Fig. 3: Composite result for all the intrusion methods on all provenance intrusion data sets. Each point in the upper left corner of the figure has both a high detection rate and a low false alarm rate, and thus shows the best performance.

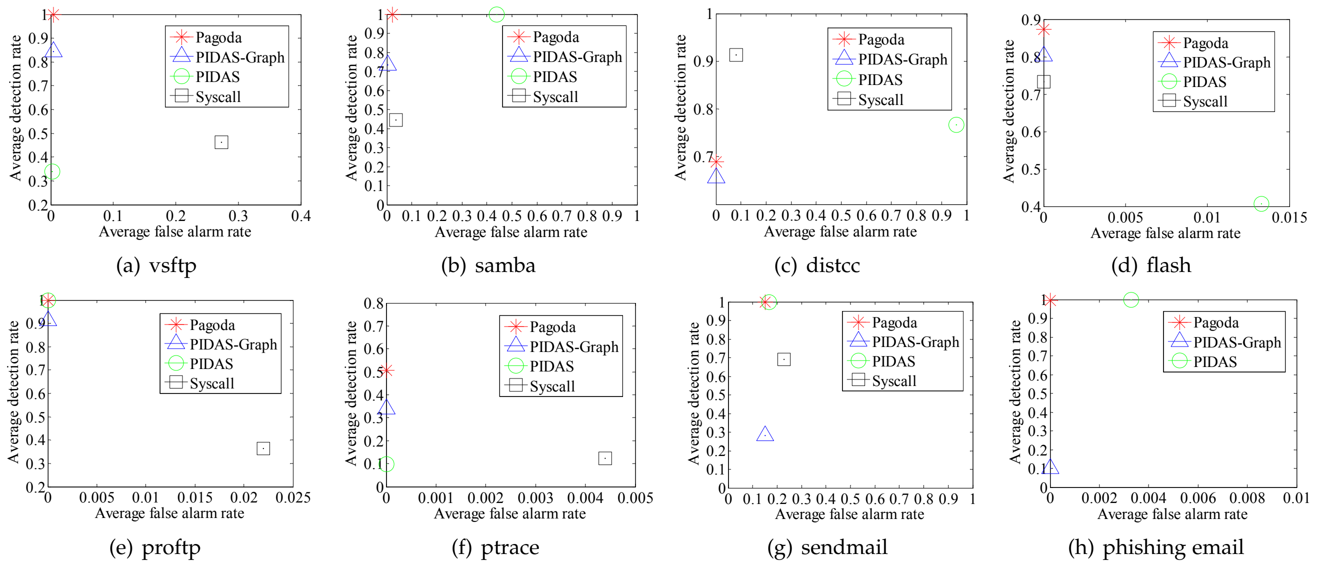


Fig. 4: ROC curves of different applications for different intrusion detection methods. For each application, we show the average value for each method. Each point in the upper left corner of the figure has both a high detection rate and a low false alarm rate, and thus shows the best performance. For Pagoda, the path threshold is 0.7 for the best detection accuracy (see Section 4.3)

graph threshold to achieve a better performance.

Note that we do not provide results on syscall for phishing email. While the *click* action on the malicious link does not invoke explicit system calls, the web page content can generate a lot of syscalls during web browsing. However, some of these syscalls do not accurately reflect the browser-intrinsic behavior (e.g., navigating to a hyperlink and then opening a tab) [41].

4.3 Threshold chosen

As the detection accuracy of Pagoda relies on both the path threshold and graph threshold, we explore how different path and graph thresholds impact detection accuracy for our method by averaging the results across all the data sets as shown in Figure 5. The false alarm rate can be very high when path threshold does not exceed 0.3. This is because a provenance path can be easier to be judged as abnormal in this case when only a few edges in this path do not occur in the rule database. Yet, the detection rate decreases with the increase of the graph threshold when the path threshold exceeds 0.3. This is due to the generation of a more strict condition with the increase of the graph threshold. However, the graph threshold must be non-trivial as a small value (< 0.5) in the graph threshold can make a large number of activities that behave only a little different from the rule database wrongly judged as abnormal. Comprehensively, Pagoda gets the best result when the path threshold exceeds 0.3 and graph threshold is around 0.5.

4.4 Detection time

Figure 6 shows the detection time of different intrusion detection methods for a wide variety of intrusion applications. The detection time is obtained by computing the means of multiple tests. Pagoda outperforms PIDAS-Graph in all the cases, reducing detection time from 21.18% to 74.48%. This is because the former first computes the anomaly degree of

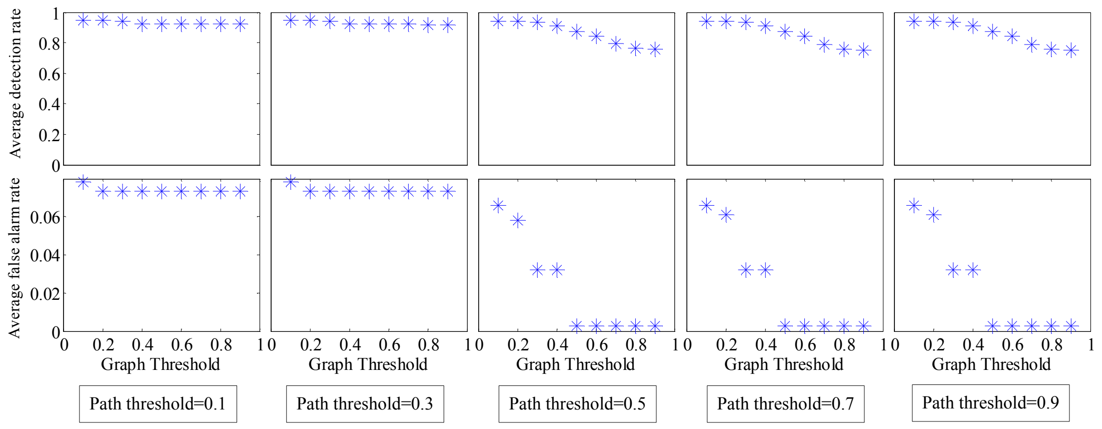


Fig. 5: Average detection rate and average false alarm rate for Pagoda for all the data sets in different path thresholds and graph thresholds.

path and judges whether an intrusion has happened. So it sometimes does not need to traverse the whole graph. It also outperforms PIDAS and PIDAS-Redis (i.e., provenance are stored in Redis when using PIDAS for detection) in all the data sets for two reasons. First, Pagoda filters more data than PIDAS, so the data to be detected is less than the latter. Second, Pagoda aggregates multiple provenance records with the same key and different values into one provenance record, so the provenance graph traverse in the detection process is faster.

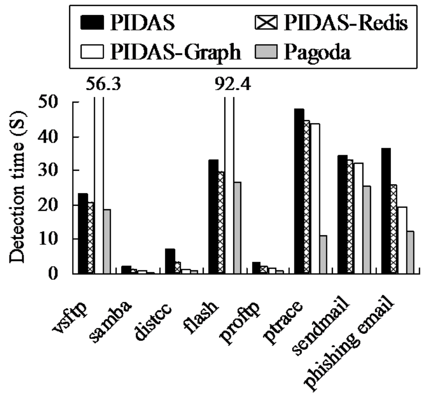


Fig. 6: Detection time of different intrusion detection methods for a wide variety of intrusion applications. PIDAS and PIDAS-Graph both get data from BerkeleyDB databases, while PIDAS-Redis and Pagoda get data from Redis when judging intrusions.

4.5 Query time

Query performance reflects the efficiency of forensic analysis. We issue the following queries on the vsftp application to compare the query performance between different intrusion detection methods.

(Q.1) Backward query: Given a detection point (e.g., a damaged file), retrieve all the objects on which it directly depends.

(Q.2) Backward query: Find all the ancestry of a given detection point.

(Q.3) Forward query: Given an intrusion process (e.g., a socket IP), retrieve all the files it has accessed.

We choose these queries because they represent the most common queries when system administrators make forensic analysis after the intrusion has happened. In addition, these queries represent different query complexity. The first two queries aim to locate the system vulnerability or intrusion sources using a detection point (e.g., a damaged file) as a starting point. The third query involves a forward query to find all the files that may have been accessed. For intrusion detection methods that use BerkeleyDB, we run experiments on both warm cache and cold cache (i.e., reboot machine before each test).

TABLE 4
Query time (us) of different intrusion detection methods.

Category	PIDAS/PIDAS-Graph		PIDAS-Redis	Pagoda
	Cold-cache	Warm-cache		
Backward-single level	9182	113	75	72
Backward-all ancestor	105795	907	891	853
Forward	96791	5121	4105	3986

Table 4 shows the query performance of different intrusion detection methods. Pagoda performs slightly better than PIDAS-Redis as Pagoda reduces more noisy data from both the rule database and the intrusion data set. They both perform better than PIDAS/PIDAS-Graph on warm or cold cache cases. First, the query process in Pagoda or PIDAS-Redis receives the data from memory, but not from the BerkeleyDB database files in the disk as in PIDAS/PIDAS-Graph. Thus their performance is better in the cold cache case. Second, Pagoda or PIDAS-Redis aggregate the provenance records that have the same key and different values into one provenance record that uses a member set to include all the values. This can significantly reduce query time especially when an object has a large number of ancestries. Hence, the query performance of Pagoda also outperforms PIDAS/PIDAS-Graph even on warm cache case.

4.6 Overhead analysis

(1) Size of rule database and intrusion data sets

TABLE 5
Size of rule database and intrusion data sets for PIDAS and Pagoda.

Application	Size of rule database (MB)			Size of intrusion data sets (MB)				
	PIDAS	Pagoda	Pagoda/PIDAS	Original	PIDAS	Pagoda	Pagoda/Original	Pagoda/PIDAS
vsftp	40	8	20.00%	2.2	1.5	0.59	26.82%	39.33%
samba	16	6	37.50%	0.855	0.537	0.31	36.26%	57.73%
distcc	1536	236	15.36%	0.465	0.284	0.15	32.26%	52.82%
flash	56	20	35.71%	18.5	12.6	10.8	58.38%	85.71%
proftp	168	124	73.81%	1.36	0.878	0.525	38.60%	59.79%
ptrace	0.016	0.008	50.00%	1.97	1.33	0.553	28.07%	41.58%
sendmail	0.016	0.008	50.00%	2.86	1.95	0.98	34.27%	50.26%
phishing email	0.048	0.036	75.00%	17.18	10.52	8.68	50.52%	82.51%

Table 5 shows the size of rule database and intrusion data sets for PIDAS/PIDAS-Graph and Pagoda respectively. As PIDAS and PIDAS-Graph employ the same filtering strategy, they have the same size of rule database and intrusion data sets. The size of the compressed rule database for Pagoda contains both the database that stores child-parent relationships and also the dictionary database that stores the mapping from the integers to the duplicate strings. The size of rule database for Pagoda is only 15.36%–75% of the ones for PIDAS/PIDAS-Graph. Pagoda has also reduced the intrusion provenance store by 41.62%–73.18% and 14.29%–60.67% when compared to the original case and PIDAS/PIDAS-Graph respectively. This is because PIDAS/PIDAS-Graph omits the temporary files or pipe files but Pagoda further filters the attributes information (e.g., arguments or environments) that is not related with the intrusion. As shown in Table 6, attributes have taken up a sizeable space in the provenance intrusion data sets. Pruning them without affecting intrusion detection can be an important way to reduce the storage overhead.

TABLE 6
Ratio of Attributes in Provenance Intrusion Data Sets.

Application	Size of attributes %	Size of others %
vsftp	78.84%	21.16%
samba	51.27%	48.73%
distcc	55.57%	44.43%
flash	24.95%	75.05%
proftp	49.45%	50.55%
ptrace	77.42%	22.58%
sendmail	64.39%	35.61%
phishing email	28.02%	71.98%

To further investigate whether Pagoda has dropped intrusion information during the filtering, we make a breakdown analysis of the intrusion data set in the cases of original, PIDAS and Pagoda as shown in Table 7. The number of processes in the attack path, files affected, intrusion sockets and the dependency relationships are the same for PIDAS and Pagoda. The decrease in the number of nodes lies in the discarded provenance records that contain the attributes (e.g., argument input of a process or environment variable) that are not used by Pagoda. This implies that Pagoda has generated a more efficient and condensed data set used for intrusion detection and analysis.

(2) Memory overhead

While the main purpose of Pagoda is to detect an intrusion with high accuracy and in real time, it is important to keep the runtime memory overhead low. Figure 7 shows the memory overhead for PIDAS and Pagoda on the 50 vsftp intrusion events. The detection algorithm processes one event every time. The memory overhead for PIDAS mainly consists of two parts: the memory overhead of detection process and the DB cache for BerkeleyDB to speed up the detection. The memory overhead for Pagoda lies in the memory overhead of both the detection process and the Redis server process to support the provenance store and process consistently in memory. Detection processes for both Pagoda and PIDAS have comparable memory overhead. This is because they both load the whole provenance data for each intrusion event into the memory. Alternatively, we set the DB cache size 16 MB for BerkeleyDB, while the memory overhead for Redis server process is only 8 MB. Thus PIDAS consumes more memory than Pagoda. The DB cache size for BerkeleyDB can be decreased, however this can have a significant impact on the detection and query performance especially when a large number of provenance data is frequently accessed in the cache.

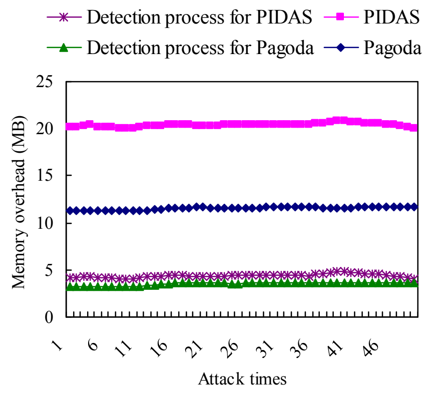


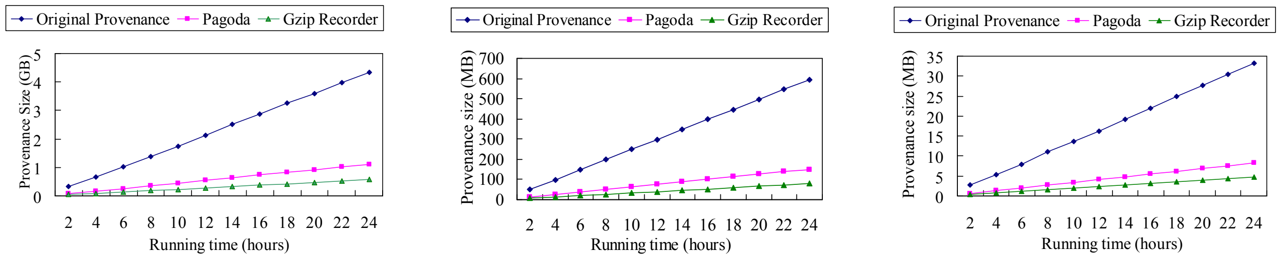
Fig. 7: Memory overhead for PIDAS and Pagoda.

(3) Provenance growth overhead

Provenance growth overhead refers to the space cost of provenance information with the increase of system run-

TABLE 7
Breakdown of Data Sets (Original/PIDAS/Pagoda)

Application	#file	#process	#socket	#relations	#nodes
vsftp	4587/4587/4587	279/279/279	981/981/981	11744/11744/11744	15895/15890/13474
samba	213/213/213	572/572/572	125/125/125	9946/9946/9946	7451/7362/7187
distcc	158/158/158	423/423/423	150/150/150	4923/4923/4923	3735/3635/3475
flash	3540/3540/3540	2251/1957/1957	706/706/706	327991/327242/327242	188221/179286/162919
proftp	250/250/250	1278/1278/1278	415/415/415	16648/16648/16648	11208/11004/10437
ptrace	4717/4717/4717	290/290/290	100/100/100	11289/11289/11289	14124/14070/12988
sendmail	4815/4815/4815	602/602/602	238/238/238	25346/25346/25346	21395/21290/20122
phishing email	601/601/601	7247/4558/4558	6380/6380/6380	289300/283290/283290	157465/149093/144966



(a) no interval (b) 3-5 seconds intervals (c) 60-90 seconds intervals

Fig. 8: Variation of the provenance size

ning time. We utilize the Web Application Stress Tools to simulate the attack under different loads. The experiment simulates 100 clients to request three data files in the server simultaneously. The file sizes are 5 KB, 20 KB, and 50 KB. The request intervals are set to no interval, 3–5 seconds and 60–90 seconds respectively. Figure 8 shows the size growth of provenance for original provenance case, provenance after filtering (i.e., Pagoda), and provenance compression using Gzip under different load conditions. The size of provenance in Pagoda has been significantly reduced. Even under the heaviest workload, the provenance information is generated about 1.1 GB in one day. This can be accepted as the price of hard disk becomes cheaper. Though provenance size can become smaller using Gzip compressor, the resulted provenance cannot be queried efficiently.

(4) Performance overhead

The provenance collection not only incurs storage space overhead, but can also have an impact on the application performance. Typically, we measure the data send rate under the above three workloads of web server attack. The impact is insignificant (0.69%–6.30%). This is because the provenance collection and storage are in different IO path from the data acquirement and send. To run a practical IO bound heavy workload, we also run the postmark benchmark on the honeypot machine. We run 1500 transactions to access 1500 files in 10 different subdirectories, with file sizes ranging from 4 KB to 1 MB. However, the provenance collection only brings 5.43% time overhead (the execution time increases from 35.89 s to 37.84 s). This shows that even under heavy load, the provenance collection scheme still has a low overhead.

To investigate how hardware performance has an impact on the overhead, we also measure the postmark performance with a low-end CPU (Intel(R) Celeron(R) CPU 1007U @1.50G Hz). The execution time has increased to 185.49 s and 199.61 s respectively. This indicates that a more powerful CPU probably exploits the parallelism and improves the performance. It also reduces the performance overhead (from 7.6% to 5.43%) incurred by provenance collection.

5 Related Work

There are some existing works on provenance-based detection method. Lemay et al. [18] proposed to make an automated analysis of directed acyclic graph by using a series of rule grammars, so as to mine and judge the anomaly in the provenance graph of application behavior. Han et al. analyzed the opportunities and challenges of using provenance for detecting intrusions [42] and used K-means clustering to judge whether the application behavior represented by a provenance graph is abnormal in the PaaS cloud [19]. In previous work, we proposed PIDAS [17], a provenance path based intrusion detection system which can clearly capture the dependencies across system activities, and easily confirm why and how the intrusions happened. The key difference of this work is that it judges intrusion not only by whether a fixed length of a single path is abnormal, but also by considering the weight factor of different lengths of paths in a whole provenance graph. Thus the detection result can be more accurate. In addition, this work filters unrelated data, compresses rule database and employs memory database to further reduce the detection time.

Provenance has been also widely used for forensic analysis [14, 21, 29, 33, 34, 43–46]. These systems collect whole-system provenance and are thus not applicable for online detection requirement in big data environments due to the dependence explosion problem [47, 48]. To alleviate the problem, a large number of techniques (e.g., execution partitioning [12, 40], logging and tainting [11]) have been proposed to reduce provenance graph size during the forensic analysis. Pagoda may complement all these OS-level provenance systems to further improve the detection accuracy and query efficiency.

One of the mainstream methods in existing host-based intrusion detection is to use a fixed length of system call sequences as the data source for intrusion detection [6]. The typical methods include system call patterns classification and mining [7], statistical analysis of intrusion behavior [8], finite-state automata [9], probabilistic approaches [49] and exploration of the system call arguments [10]. The key difference of our work is that we identify intrusions by explicitly exploiting the information flow using dependency relationships between files and processes.

The application of machine learning and artificial intelligence technology [50–52] has long been proposed to improve the detection rate of IDS. For instance, Lin et al. proposed an anomaly detection approach which combines support vector machine (SVM), decision tree (DT) and simulated annealing (SA) to extract the best selected feature and generate optimal decision rules [50]. Ghosh et al. applied Artificial Neural Networks to generate the rules through autonomous learning, classify the simple data into training data, and make an automatic response to system behavior by constructing finite automata [51]. As a convolutional neural network has been proposed to process arbitrary graphs [53], it can be used to analyze the provenance graph representations to accurately identify the unseen intrusions.

For the past few years, the increase in network speed from Mbps to Gbps has posed a new challenge to existing IDSs and being a main factor to restrict the real-time of IDS. The main solutions include distributing the workload among multiple devices [54] and improving single detection engine by exploiting the potential hardware performance [20]. Our approach improves the processing speed by providing a comprehensive solution that includes optimization of the detection algorithm, filtering the intrusion data set, compressing rule database and employing memory database to avoid disk I/O.

Recently, Gu et al. proposed to first eliminate useless training data by analyzing the control flow graph generated from the intrusion behavior and then applied the statistical learning method to improve the detection accuracy [55]. This is similar to the filter method we use to reduce noisy items in the intrusion data sets. However, our method can be applied to both training data and the test data. There are also other works to improve detection efficiency, such as building intrusion detection model [56], investigating how to evade the detection online [57], and estimating the detection accuracy by sampling [58].

6 Conclusions

Efficient intrusion detection and analysis in big data environments have become a major challenge for today's personal and enterprise users. This paper proposes Pagoda, a simple and hybrid approach to enable accurate and fast detection by taking into account the anomaly degree of both single provenance path and the whole provenance graph. It employs non-volatile memory database to store and process data during the detection process, and aggregates multiple database items into one value to further improve provenance query and detection speed. In addition, it filters out irrelevant information in the provenance collecting process, and removes the duplicates in the rule database to save memory space and speed up the detection. Experimental results demonstrated the performance and efficiency of this system.

Acknowledgments

We thank the anonymous reviewers for their constructive and detailed comments. This work was supported in part by the National Science Foundation of China under Grant No. U1705261 and 61821003, CCF-NSFOCUS KunPeng research fund, Wuhan Application Basic Research Program under Grant No. 2017010201010104, and Hubei Natural Science and Technology Foundation under Grant No. 2017CFB304. Corresponding author: Yulai Xie, ylxie@hust.edu.cn

References

[1] "Security statistics for 2016," https://securelist.com/kaspersky-security-bulletin-2016-executive-summary/76858/.

[2] "APT attack analysis by DARPA transparnt computing program," http://www.darpa.mil/program/transparent-computing.

[3] "Ransomware reported by the guardian," https://www.theguardian.com/technology/2017/may/12/global-cyber-attack-ransomware-nsa-uk-nhs.

[4] "Economic loss reported by security daily," http://securitydaily.org/british-insurance-organizations-global-cyber-attacks-spur-53-billion-losses/.

[5] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[6] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Proceedings of the IEEE Symposium on Security and Privacy*, 1999, pp. 133–145.

[7] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *Proceedings of the 7th USENIX Security Symposium*, 1998, pp. 79–93.

[8] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2001, pp. 156–168.

[9] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2001, pp. 144–155.

[10] G. Tandon and P. K. Chan, "On the learning of system call attributes for host-based anomaly detection," *International Journal on Artifical Intelligence Tools*, vol. 15, no. 6, pp. 875–892, 2006.

[11] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[12] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[13] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[14] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[15] K. H. Lee, X. Zhang, and D. Xu, "LogGC: garbage collecting audit log," in *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, 2013.

[16] "Trustwave global security report, 2015," https://www2.trustwave.com/rs/815-RFM-693/images/2015_TrustwaveGlobalSecurityReport.pdf.

[17] Y. Xie, D. Feng, Z. Tan, and J. Zhou, "Unifying intrusion detection and forensic analysis via provenance awareness," *Future Generation Computer Systems*, vol. 61, pp. 26–36, 2016.

[18] M. Lemay, W. U. Hassan, and T. Moyer, "Automated provenance analytics: A regular grammar based approach with applications in security," in *Proceedings of the USENIX Workshop on the Theory and Practice of Provenance*, 2017.

[19] X. Han, T. Pasquier, T. Ranjan, M. Goldstein, and M. Seltzer, "Frappuccino: Fault-detection through runtime analysis of provenance," in *Workshop on Hot Topics in Cloud Computing*, 2017.

[20] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a highly-scalable software-based intrusion detection system," in *Proceedings of the 2012 ACM conference on Computer and Communications Security*, 2012, pp. 317–328.

[21] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[22] P. A. Boncz, "Monet: A next generation DBMS kernel for query-intensive application," Ph.D. dissertation, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[23] J. Lyle and A. Martin, "Trusted computing and provenance: Better together," in *Proceedings of the 2nd Workshop on the Theory and Practice of Provenance*, 2010.

[24] R. Hasan, R. Sion, and M. Winslett, "The case of the fake picasso: Preventing history forgery with secure provenance," in *USENIX Conference on File and Storage Technologies*, June 2009.

[25] K. K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems," in *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006, pp. 43–56.

[26] P. Buneman, S. Khanna, and W. C. Tan, "Why and where: A characterization of data provenance," in *International conference on database theory*, vol. 1, 2001, pp. 316–330.

[27] T. Gibson, K. Schuchardt, and E. G. Stephan, "Application of named graphs towards custom provenance views," in *Workshop on the Theory and Practice of Provenance*, 2009.

[28] S. Shah, C. A. N. Soules, G. R. Ganger, and B. D. Noble, "Using provenance to aid in personal file search," in *Proceedings of the USENIX Annual Technical Conference*, June 2007.

[29] A. Gehani and D. Tariq, "SPADE: Support for provenance auditing in distributed environments," in *Proceedings of the 13th International Middleware Conference*, Dec 2012, pp. 101–120.

[30] R. P. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok, "Story Book: An efficient extensible provenance framework," in *Workshop on the Theory and Practice of Provenance*, 2009.

[31] A. Vahdat and T. E. Anderson, "Transparent result caching," in *Proceedings of the USENIX Annual Technical Conference*, 1998.

[32] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[33] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: collecting high-fidelity whole-system provenance," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 259–268.

[34] A. Bates, D. Tian, K. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *Proceedings of the USENIX Security Symposium*, 2015.

[35] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in provenance systems," in *Proceedings of the USENIX Annual Technical Conference*, June 2009.

[36] K.-K. Muniswamy-Reddy and D. A. Holland, "Provenance for the cloud," in *Proceedings of the USENIX Conference on File and Storage Technologies*, Feb 2010.

[37] "Redis," https://redis.io/.

[38] "Web application stress tool," https://west-wind.com/presentations/webstress/webstress.htm.

[39] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, Y. Li, and D. D. E. Long, "Evaluation of a hybrid approach for efficient provenance storage," *ACM Trans. on Storage*, vol. 9, no. 4, pp. 1–29, 2013.

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

[41] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie, A. Gehani, and V. Yegneswaran, "MCI: Modeling-based causality inference in audit logging for attack investigation," in *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*, 2018.

[42] X. Han, T. Pasquier, and M. Seltzer, "Provenance-based intrusion detection: Opportunities and challenges," in *Workshop on Theory and Practice of Provenance*, 2018.

[43] A. Bates, W. U. Hassan, K. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear, "Transparent web service auditing via network provenance functions," in *WWW*, 2017.

[44] G. Jenkinson, L. Carata, N. Balakrishnan, T. Bytheway, R. Sohan, R. Watson, J. Anderson, B. Kidney, A. Strnad, and A. Thomas, "Applying provenance in APT monitoring and analysis," in *USENIX Workshop on the Theory and Practice of Provenance*, 2017.

[45] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, "Secure network provenance," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[46] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *NDSS*, 2018.

[47] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," in *Proceedings of the SOSP*, 2005.

[48] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," in *Proceedings of the OSDI*, 2010.

[49] E. Eskin, "Anomaly detection over noisy data using learned probability distributions," in *Proceedings of the International Conference on Machine Learning*, 2000.

[50] S. Lin, K. Ying, C. Lee, and Z. Lee, "An intelligent algorithm with feature selection and decision rules applied to anomaly intrusion detection," *Applied Soft Computing*, vol. 12, no. 10, pp. 3285–3290, 2012.

[51] A. Ghosh, C. Michael, and M. Schatz, "A real-time intrusion detection system based on learning program behavior," in *International Workshop on Recent Advances in Intrusion Detection*, 2000, pp. 93–109.

[52] M. Salama, H. Eid, R. Ramadan, A. Darwish, and A. Hassanien, "Hybrid intelligent intrusion detection scheme," *Soft computing in industrial applications*, pp. 293–303, 2011.

[53] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *Proceedings of the International Conference on Machine Learning*, June 2016.

[54] W. Jiang, H. Song, and Y. Dai, "Real-time intrusion detection for high-speed networks," *Computers & security*, vol. 24, no. 4, pp. 287–294, 2005.

[55] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu, "Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis," in *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015, pp. 57–68.

[56] X. Shu, D. Yao, and B. G. Ryder, "A formal framework for program anomaly detection," in *International Workshop on Recent Advances in Intrusion Detection*, 2015, pp. 270–292.

[57] N. Srndic and P. Laskov, "Pratical evasion of a learning-based classifier: A case study," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.

[58] B. Juba, C. Musco, F. Long, S. Sidiroglou-Douskos, and M. C. Rinard, "Principled sampling for anomaly detection," in *Proceedings of the Network and Distributed System Security Symposium*, 2015.

Yulai Xie received the B.E. and Ph.D. degrees in computer science from Huazhong University of Science and Technology (HUST), China, in 2007 and 2013, respectively. He was a visiting scholar at the University of California, Santa Cruz in 2010 and a visiting scholar at the Chinese University of Hong Kong in 2015. He is now an associate professor in School of Computer Science and Technology in HUST, China. His research interests mainly include digital provenance, intrusion detection, network storage and computer architecture.

Dan Feng received her B.E, M.E. and Ph.D. degrees in Computer Science and Technology from Huazhong University of Science and Technology (HUST), China, in 1991, 1994 and 1997 respectively. She is a professor and director of Data Storage System Division, Wuhan National Lab for Optoelectronics. She is also dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, parallel file systems, disk array and solid state disk. She has over 100 publications in journals and international conferences, including FAST, USENIX ATC, ICDCS, HPDC, SC, ICS and IPDPS. Dr. Feng is a member of IEEE and a member of ACM.

Yuchong Hu received the B.S. degree in Computer Science and Technology from the School of the Gifted Young, University of Science and Technology of China, Anhui, China, in 2005, and the Ph.D. degree in Computer Science and Technology from the School of Computer Science, University of Science and Technology of China, in 2010. He is currently an Associate Professor with the School of Computer Science and Technology, Huazhong University of Science and Technology. His research interests focus on improving the fault tolerance, repair and read/write performance of storage systems, which include cloud storage systems, distributed storage systems and NVM-based systems.

Yan Li received his Ph.D. degree in Computer Science from Jack Baskin School of Engineering, University of California, Santa Cruz, in 2017. His Ph.D. advisor was Professor Darrell Long. He is currently a systems and deep learning researcher, inventor and founder. Yan focuses on computer performance tuning using machine learning and artificial intelligence.

Staunton Sample is an M.S. student in the SSRC who started in Fall 2017. His research interests lie in storage and systems security. He is currently working under Prof. Darrell Long. S-taunton has a B.A. in Linguistics from Louisiana State University.

Darrell Long received his B.S. degree in Computer Science from San Diego State University, and his M.S. and Ph.D. from the University of California, San Diego. Dr. Darrell D.E. Long is Distinguished Professor of Computer Engineering at the University of California, Santa Cruz. He holds the Kumar Malavalli Endowed Chair of Storage Systems Research and is Director of the Storage Systems Research Center. His current research interests in the storage systems area include high performance storage systems, archival storage systems and energy-efficient storage systems. His research also includes computer system reliability, video-on-demand, applied machine learning, mobile computing and cyber security. Dr. Long is Fellow of IEEE and Fellow of the American Association for the Advancement of Science (AAAS).