Regular Paper

Real-time Container Integrity Monitoring for Large-Scale Kubernetes Cluster

Hirokuni Kitahara[1,a)] Kugamoorthy Gajananan[1,b)] Yuji Watanabe[1,c)]

Received: November 30, 2020, Accepted: June 7, 2021

Abstract: Container integrity monitoring is defined as a key requirement for regulatory compliance such as PCI-DSS, in which any unexpected changes such as file updates or program runs must be logged for later audit. System call monitoring provides comprehensive monitoring of such change events on container since it may suffer from large amount of false alarms unless well-defined allowlist rules are coordinated before deploying a container. Defining such a comprehensive allowlist is not feasible especially when managing various kinds of application workloads in large-scale enterprise cluster. We propose a new approach for identifying real anomalies in system call events effectively without relying on any predefined allowlist configuration in this paper. Our novel filtering algorithm based on the knowledge acquired autonomously from Kubernetes cluster control plane reduces 99.999% of noise effectively and distills only abnormal events in real time. Furthermore, we define concrete criteria for highly-scalable container integrity monitoring and verify the implementation of proposing filtering method that has actual high scalability while maintaining its detection capability. Our experiment with real applications on around 3,800 containers demonstrates its effectiveness even on large-scale clusters, and we clarified how detected events are triggered by user operation.

Keywords: container integrity, Kubernetes, allowlist

1. Introduction

System integrity monitoring refers to a function that detects and notifies the administrator of events such as file changes and deletions that should not be rewritten in service operations and process executions that should not be executed. It is an important part of several compliance and audit requirements including PCI [4], FISMA [18], and HIPAA [18]. All changes need to be evaluated (after accounting for known changes to be expected in the container based on a allowlisted profile) by the system administrator for further investigation.

To deal with the requirement, a mechanism for detecting changes of system state must be put in place in the system. A common approach for host system is to introduce a monitoring agent on the host OS, and a number of known File Integrity Monitoring (FIM) tools such as Tripwire [12] or OSSEC [15] have been known for host integrity monitoring, while container integrity monitoring requires a different approach called "agentless" monitoring since it is much harder to introduce agents to a container with a shorter life-cycle than the host. There are two types of approaches to monitor such changes in container: - 1) detecting differences between periodical scanning, or 2) continuous system call monitoring. For the first approach, the crawler running on the host monitors the state of containers running on the same host periodically, and checks difference in states between crawls. The periodic scanning approach provides comprehensive detection of any changes of static state such as files with very small overhead since it does not add to running cost during idle time, while some information, such as program runs during crawls or process which causes the file changes, cannot be identified directly.

System call monitoring addresses this issue in a different approach. System call events include all OS-level system call information such as file change/update/delete, program run/kill. System call events in containers can be observed on host transparently due to the nature of container. Monitoring system call events on host provides comprehensive detection of any lowlevel integrity events without touching containers, while the following issues rise from its comprehensiveness and become even worse especially in a large-scale cluster [*1].

- Too many and too much detail in low-level events could include vast majority of events from expected behaviors of container such as expected file changes (e.g., temporal files, log files) or internal or regular program runs. This means reported alerts may include a lot of false alarms.
- Too many events cause increase in resource usage such as data bandwidth and storage in central server as audit log.

Such false alarms can be suppressed by applying allowlist which accounts for known changes to be expected in a container, but defining the allowlist configuration properly is not feasible because administrators may not know the full internal behavior of all applications running on cluster especially in large-scale enterprise cluster where thousands of applications from different own-

[1] IBM Research - Tokyo, Chuo, Tokyo 103–8510, Japan
[a)] hirokuni.kitahara1@ibm.com
[b)] gajan@jp.ibm.com
[c)] muew@jp.ibm.com

[*1] This paper is an extended version of a conference paper for IEEE BigData 2020 accepted on Oct. 21, 2020. The 2 major extensions are performance evaluation for scalability and analysis on detected results.

ers are hosted and maintained by a few administrators. Furthermore, the task for defining a comprehensive allowlist is not easy even for application developers and requires cumbersome manual tasks with deep understanding of run-time behaviors.

In this paper, we propose a novel approach for highly scalable container integrity monitoring using system call monitoring [8]. We do not rely on any predefined allowlist configuration, i.e., no prior knowledge about container image for creating container. Instead, our approach leverages knowledge acquired from Kubernetes cluster control plane for constructing effective filtering. We implement a new algorithm for accumulating multiple events with a process tree to maximize the power of reduction by filtering. The major part of aggregation and filtering are completed at each node in a distributed manner, and so it minimizes the resource usage for enabling this monitoring on a large-scale cluster. In addition to this, we provided specific criteria for evalutation about scalability of container integrity monitoring from 3 perspectives of processing delay, distribution and data size. In order to demonstrate the effectiveness of the proposed method and to verify its scalability, we conducted experiments on real application workloads running on Kubernetes cluster consisting of 3,800 containers (on average) on 160 nodes. We use Sysdig [2] Falco [6] for monitoring events such as file update/delete/create, and process exec. We observed the total amount of such events reached the rate of 2.3 million events/hour, which resulted in 6 GB hourly volume usage increase before applying the proposed approach. Then, we deployed our proposed filtering on the cluster without having any prior knowledge about deployed application. We confirmed it effectively distills only 3 events/hour of interest from 2.3 million in real-time with very limited resource usage, thus successfully suppressing the vast majority of events (99.99988%) only with the knowledge acquired autonomously from Kubernetes cluster control plane via standard Kubernetes API. For its correctness, we conducted an interview with a cluster administrator of the target cluster, and verified that all of detected events are real anomalies to be reported. This paper gives the first specific construction method for achieving that level of suppression of events in container integrity monitoring without relying on any predefined allowlist.

2. Definition and Goals

We briefly describe relevant technologies used in this work and define some terminologies to describe our approach.

2.1 Container Orchestration System

In cloud context, containers are an approach to package an application including their libraries, dependencies, configurations and files. Containers share the same operating system kernel when running on a host machine, and isolate the application processes running in the container from the rest of the system [10].

Container orchestration systems are a framework for managing life-cycle of containers at scale. Container orchestration systems schedule containers across a cluster, scale those containers, and manage their health over time. One could build a container orchestration system by clustering a group of hosts (aka nodes), either physical or virtual machines and running containers [9]. For instance, Kubernetes is a popular open source container orchestration system [14].

Table 1 Types of system calls related to system integrity.

System Call	*Event Type*	*Description*
execve	Process	Process Execution
open, openat, mkdir, mkdirat	File	Create or Change File
link	File	Create Sympolic Link
unlink, unlinkat, rmdir, rmdirat	File	Delete File

Container orchestration systems like Kubernetes provide a mechanism called ‘Namespace’ which are logical partitions for subdividing a cluster so that multiple applications can share a cluster [14]. In a cluster, one could deploy a containerized application in a Namespace. The containers that make up an application are managed in units of in-app components called Pods.

The scope of our work is the integrity monitoring of containers which runs on Linux based hosts managed by a Kubernetes cluster.

2.2 Container Monitoring

The system calls invoked by containers are processed as system calls at the host, since containers and the host they run on share the same operating system kernel. Therefore, by observing system calls on hosts, one could monitor system calls generated by containers, without making any changes to them. For instance, there are tools like Sysdig Falco [2], [6] that exploit this approach to monitor the actual behavior of host systems and containers.

In this work, we use Falco as the underlying technology for system level monitoring of containers.

Falco is used only for getting various attributes like process id and container metadata of a system call event in this paper, and some subsequent filtering and grouping methods are our contribution work.

We introduce the filtering method of our proposal in Section 4.

In typical Linux systems, there are various kinds of system calls. Among them, we focus on the system calls that are closely related to system integrity. For instance, **Table 1** lists the types of system calls related to process execution and file changes.

2.3 Expected System Call Events

A container is created at run-time based on a container image which includes a pre-defined program. This program starts up and operates when the container is created on a host. The program that runs on the container may launch a new process or access a file. For example, let us assume a container that runs a simple HTTP server which accepts incoming HTTP requests, performs some operation and generates some log files. In such case, one could detect system calls related to process execution and file changes. However, these system calls are expected at the time of building and distributing the container image and starting the container. Therefore, these events are not our target for monitoring integrity of containers. These type of events are called “expected system call events” or more simplly “expected events”.

2.4 Mutation Events

Containers should be immutable. Immutable means that these are not expected to be modified after startup during its service life so there are no updates, patches, configuration changes. Therefore, any operations from outside the container are prohibited. All observed events should be expected system call events. Hence, we define all events excepts the expected system call events as mutation events which are the primary target of integrity monitoring for containers. For example, some file change events for an attacker to install their own malicious application in a running container are mutation events.

2.5 Goals of Our Proposed Approach

In large-scale clusters, a number of hosts (nodes) may exceeds a thousand. In such clusters, as the applications are deployed per Namespace, there would be a huge number of system calls might occur. In such an environment, efficiently detecting mutation events from a huge number of events is a challenging task. Therefore, in this work, our goal is to efficiently exclude expected events and detect mutation events from a huge number of system call events observed in a large-scale cluster.

3. Rule-based Allowlist for Container Integrity Monitoring

We describe the challenges of defining rule-based allowlists for filtering expected events, especially in a large-scale cluster.

3.1 System Call Monitoring in Containers

We conducted a preliminary study to analyze what composition of system call events could occur in a three node cluster. From this study, we collected system call event data by monitoring the cluster.

Figure 1 illustrates system call distribution in the data collected over five days. Among 3.9 million events, almost 91% of the events are of 'execve'. In other words, 90% or more of the system calls that occurred in the cluster were process events.

3.1.1 Scale of System Call Events in Kubernetes Cluster

Figure 2 reveals the amount of system call events monitored per node and per namespace over time. A huge number of events occured constantly at every node and the application running on the cluster constantly generated system call events. It also illustrates that a huge amount of system call events constantly occurred regardless of the content of the application. This means that container operation always generated a large number of expected events. Hence, we need to allowlist these container expected behavior events while monitoring integrity.

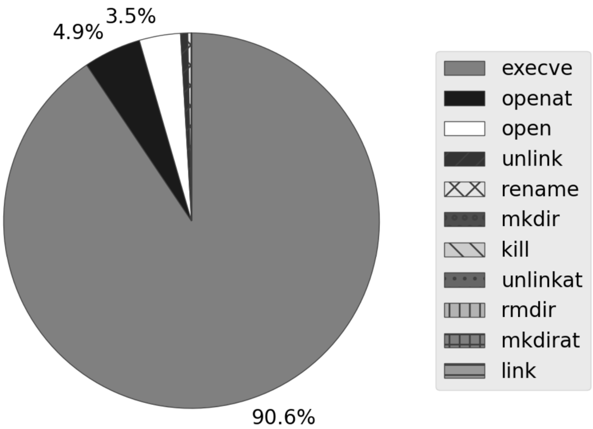


Fig. 1 Distribution of system calls.

3.2 Allowlist of Process Events

From the preliminary study, which was monitoring all file related system calls and process related system calls on a cluster, we found that over 90% of the events that occured in a cluster were process events and file changes were triggered by some process execution. The ratio of process executions and file changes depends on actual application on each cluster, however, at least, all file changes are triggered by some corresponding parent process events.

Therefore, we will focus on process events in the rest of this section.

3.2.1 Process Execution Commands

We studied frequently appear commands among the system calls events observed. We collected the command string that caused the event as the attribute value of the event. **Table 2** provides a list of frequently appeared commands in the study environment and their number of occurrences.

For example, we observed that the python command with the highest number of times occurrence at a frequency of 130 events/min or more for 4 days. Process executions that happened at such a high rate were obviously caused by some programs like container expected programs or Kubernetes system. Those machinery events are candidates for filtering.

3.2.2 Converge of Frequent Commands

To create an allowlist based on frequently appearing commands in events, we examined the top ten most frequently used commands that were executed repeatedly tens of thousands to hundreds of thousands times. It is likely that the high occurrence frequency of a specific command was due to the fact that the process defined as the operation of a container (i.e., the expected event) repeatedly executed the command. If these frequent com-

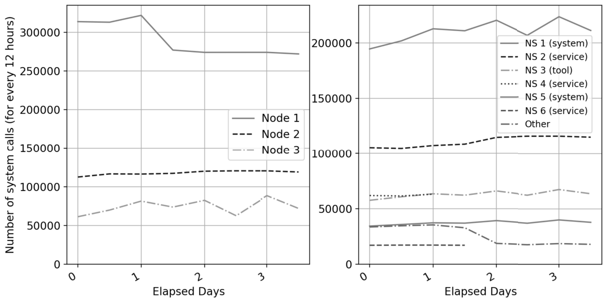


Fig. 2 System call events over time for each node and namespace.

Table 2 Top 10 commands for process execution.

Command	*Count*	*Percentage*
python3 [filename A]	787,025	20.2%
[filename B] [option1]	307,532	7.9%
[filename B] [option2]	265,378	6.8%
sh [filename C]	206,304	5.3%
sv status [filename D]	176,864	4.5%
sv status [filename E]	176,864	4.5%
[filename F] [option 3]	133,217	3.4%
[filename G] [option 4]	130,495	3.3%
sh [option 5]	117,889	3.0%
sleep [option 6]	88,428	2.3%

mands correspond to expected events, we could use them as an allowlist to exclude those commands as non-mutation events.

From the collected data, we observed that commands with the highest frequency of appearance occupied the majority of all events. In fact, the top 10 commands cover 61.6% and 100 commands covered 95.5% of the total events observed. Therefore, if the hypothesis (frequent command=expected event) holds, one could consider occurrence frequency of commands executed to prove effective in narrowing down candidates for mutation events to some extent.

3.2.3 Limitations of Command Allowlist

Although the occurrence frequency of commands executed could help to form an allowlist. In reality narrowing down mutation events is not practical due to the following reasons.

- Static allowlist cannot filter commands that change at each execution
- Updating the allowlist dynamically would generate false negatives

In fact, including more frequent patterns in the allowlist leaves a large amount of expected behavior events in the remaining events.

For example, when the top 1,000 items are included in the allowlist, a command pattern

httpserver -output 20200726*.log

is included in the allowlist as a frequent pattern, while the same type of command

httpserver -output 20200727*.log

remains without being allowlisted.

The results from the investigation demonstrate the need for an automated allowlisting mechanism as cluster administrators would not know what applications would run on a cluster. In addition, application developers may not comprehensively define all patterns of commands executed in a container. Therefore, for integrity monitoring of a cluster, an automated mechanism that effectively filters expected events (i.e., generation of allowlist), is indispensable, especially in a large-scale cluster where many applications are deployed.

4. Proposed Approach (Auto Allowlisting)

The proposed method consists of the following three stages of event filtering.

(1) Common Pattern Filter - Our approach considers that there is a common pattern that does not depend on each container image, among expected events. It specifies a common pattern that can classify most of the events as the expected event.

(2) Event Grouping - The expected events, which cannot be filtered in (1), differ by application. Our approach considers most of these event patterns also should be allowlisted, but they differ for each application. Therefore, our approach applies event grouping to the rest of events by focusing on the parent-child relationship between events. It specifies the event that is the starting point of each group.

(3) Application Pattern Filter - Our approach applies the correlation with the application configuration to the origin event of the groups identified in (2). Then, it identifies the event pattern originated by each application. The events filtered by this pattern are the expected events too.

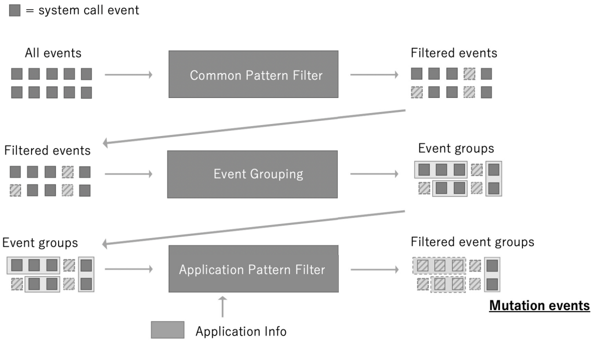


Fig. 3 Steps for effective event filtering.

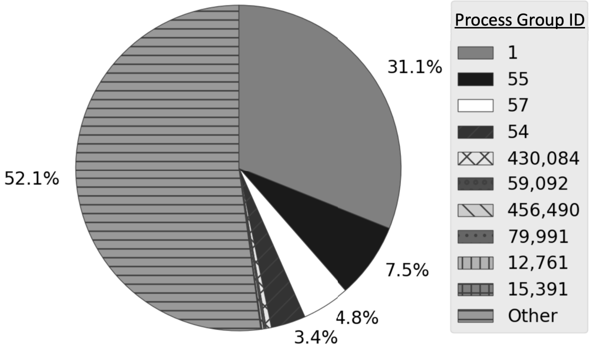


Fig. 4 Distribution of process group IDs.

Figure 3 shows a conceptual diagram of the above three steps. By the above steps, our approach could effectively filter both the application-independent expected behavior and the application-dependent expected behavior and extract only mutation events. We describe the actual configuration method and design guidelines for each step as below.

4.1 Common Pattern Filter

By applying the system call monitoring technique, one could acquire information of the process id that generated the events. Since each container normally has a unique process namespace, the process ID number depends on the process execution status in each container.

The process ID itself has no meaning as a allowlist condition. However, system call monitoring could acquire the group ID of the process executed in the container as well as the process ID. This group ID represents the ID of the starting process of the process tree. When we classify events by group ID, as shown in **Fig. 4**, only the top 10 could cover about 50% of the total events. Therefore, we investigate this process group ID and why some specific process group IDs covers such portion of events in the next section.

4.1.1 Mechanism on Common Pattern Filter

In containerizing an application using a container, each container has only one application executed in the container. In other words, each container would run only one application, and when the application ends, the container would also stop operating. Even though it is not impossible to execute multiple applications like a conventional virtual machine, but it will become a zombie process if the application terminates abnormally. This would greatly reduce the merit of using the container. Such container design is called single process per container.

With this design, we can represent the processes running in a container as a process tree with a common ancestor. The process group ID is common to the events in the tree. The most characteristic example is group ID 1, which is shown as the most common group ID in figrefvpgid. The events that occur from the container expected behavior have a process group ID of 1.

In addition, it is possible to execute multiple processes from one container as described above, there may be different groups generated, which have group ID 1 as an origin. A typical example is a process execution using the sv program of Linux. In this case, the process tree executed from the program called runsv, which is also used as a allowlist. On the other hand, the above "multiple starting process container" could be created without runsv. A small amount of pre-defined allowlists might be needed for these type of cases.

Additionally, if attacker tampers with a container image to inject their attack commands into the starting process, the attack commands might hide from our proposal detection. To avoid these type of attacks, the container image should be managed properly and it should be protected in some additional ways such as signature.

We describe the above process as pseudo code shown below.

Algorithm 1: Common Pattern Filter

```
Input: Event $e_{all}$
Output: Filtered Event $e_{filtered}$
for $i = 0 \ldots e_{all}$.length do
    $filterOut = False$
    if $ProcessGroupID(e_i) == 1$ then
        $filterOut = True$
    else
        $e_p = GetParentProcess(e_i)$
        for do
            if $GetParentProcess(e_p) == runsv$ then
                $filterOut = True$
                break
    if $filterOut == False$ then
        $e_{filtered}.Add(e_i)$
return $e_{filtered}$
```

4.1.2 Verification in Test Environment

We have validated a common pattern filter using the test data collect in a cluster consisting of three nodes, and about 43% of the whole could be filtered by common pattern filter. This result is almost equivalent to previous section, so we can conclude that a common pattern filter could work as a filter for 40–50% of events out of the call events for the entire system.

4.2 Event Grouping

For application dependent filtering in second stage of event grouping, we perform event grouping by the following process. We organize all the events that occurred in each container into a process tree, and specify the starting event.

Our approach (1) sorts events by container (2) groups process execution events and file change events as a process tree which has a root process execution event above it and also has an offspring process and file events (3) using the parent process ID and group ID to process all process events in process tree format.

Table 3 Top 8 commands after common pattern filter & event grouping.

Command	*Count*	*Configuration*
[filename A] [option 1]	101,970	livenessProbe
sh [filename B]	71,577	livenessProbe
[filename C] [option 2]	66,994	readinessProbe
[filename D] [option 3]	51,067	readinessProbe
sh [option 4]	36,774	readinessProbe
test [filename E]	34,068	readinessProbe
[filename A] [option 5]	34,059	readinessProbe
sh [filename F]	30,656	readinessProbe

4.3 Application Pattern Filter

4.3.1 Events Defined by Application

Table 3 shows the list of frequent commands for events remaining after Common Pattern Filter and Event Grouping. One can see that there are multiple commands that are repeatedly executed in large numbers on the scale of hundreds of thousands of times. Since the Common Pattern Filter already removed the event pattern from a container, these frequent commands are events that are caused by the Kubernetes internal components and not by a container itself. We analyze each system configuration information (Pod definition) for the frequent commands in Table 3. One could see that all of these are commands defined by the item "livenessProbe" or "readinessProbe" in the configuration information.

Although Table 3 lists livenessProbe and readinessProbe, we also find that the command string defined as 'lifecycle.preStop' was also detected in the same way.

We found that the 51.4% of commands are originated from readinessProbe configuration, 48.4% are from livenessProbe and 0.01% are from lifecycle.Prestop. Other events were very small, less than 0.01%.

Although it is different from the event due to the expected operation of the container itself, it is an event that occurs by default in the application settings and therefore we could treat them as allowlist commands. There is a possibility that this probe configuration is used for a malicious command injection if the configuration is not correctly managed. In Kubernetes, this kind of configuration is normally managed by role base access control (RBAC), which defines what configuration a user can change. So, in this paper, we assume probe commands have not been illegally tampered and this is why we can filter out events that are triggered by probe commands.

4.3.2 Mechanism on Application Pattern Filter

It turns out that there are a large number of commands defined for items such as livenessProbe and readinessProbe. What do these items mean in the pod definition? **Table 4** shows the meaning of each item and Pod definition. Note that lifecycle.postStart, which may be detected as in lifecycle.preStop, is also shown in the table.

The commands defined in this pod configuration information are specified when creating a pod. The events generated from them are preset to be executed as an application. Therefore, we

Table 4 Pod configurations related to system call events.

Configuration	*Purpose*	*Frequency*
livenessProbe	Check container liveness	repeat
readinessProbe	Check container readiness	repeat
lifecycle.preStop	Execute finalize method	once
lifecycle.postStart	Execute initialize method	once

Algorithm 2: Application Pattern Filter

```
Input: Event Groups $g_{all}$
Output: Mutation Event Groups $g_{mut}$
for $i = 0 \dots g_{all}$.length do
    $P_i$ = GetApplicationDefinition($g_i$)
    $C$ = GetDefinedCommands($P_i$)
    $filterOut = False$
    for $j = 0 \dots C$.length do
        if $GetCommand(g_i) == C_j$ then
            $filterOut = True$
    if $filterOut == False$ then
        $g_{mut}.Add(g_i)$
    return $g_{mut}$
```

could consider these events as part of expected behavior in the container. Hence, we could add them to the allowlist condition by referencing the configuration information.

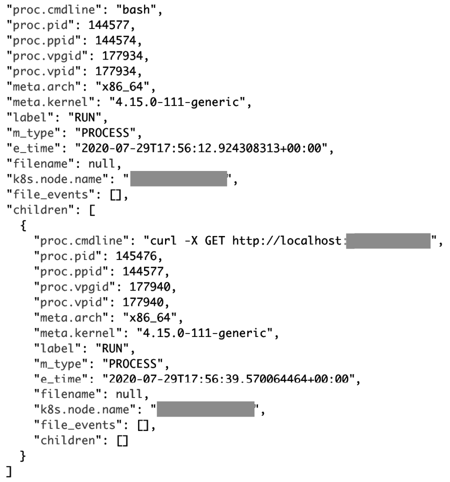
We provide a pseudo-code for the process discussed in this section, as shown in Algorithm 2. Our approach obtains Pod configuration information by using the Kubernetes API corresponding to GetApplicationDefinition() in the pseudo code.

4.3.3 Verification in Test Environment

We verified the effect of Application Pattern Filter with the event data collected in a 3-node cluster different from the above. Similar to the previous section, the application-defined expected behavior accounted for almost 100%, and other events were negligible at 0.05%.

Most of the events are livenessProbe (34.0%) and readinessProbe (65.8%), and number of lifecycle.preStop is very small (0.2%). This is due to the difference in the execution timing of each item. We observe the same result in all environments.

Mutation events are the group of events that finally remain after the processing of filtering stages described in Sections 4.1, 4.2, and 4.3. In addition, we call the origin event of each process tree among mutation events, **the Infiltrate Event** in this paper. This is because the collection of origin events represent the command when invading a container. Organizing them in a process tree makes it easier to determine what happened during one intrusion. We show an example of the detection result by the proposed method as below. In this example, after invading the container with the bash command, an intruder invoked the communication with the specific API via a curl command. The bash command becomes the Infiltrate Event, and the process execution of the curl command is detected as the Mutation Event of its child. In this way, we demonstrate that it is possible to analyze the mutation events executed in a container more effectively by organizing them in a tree format using the parent-child relationship instead of the system call event.



```
"proc.cmdline": "bash",
"proc.pid": 144577,
"proc.ppid": 144574,
"proc.vpgid": 177934,
"proc.vpid": 177934,
"meta.arch": "x86_64",
"meta.kernel": "4.15.0-111-generic",
"label": "RUN",
"m_type": "PROCESS",
"e_time": "2020-07-29T17:56:12.924308313+00:00",
"filename": null,
"k8s.node.name": "[redacted]",
"file_events": [],
"children": [
  {
    "proc.cmdline": "curl -X GET http://localhost[redacted]",
    "proc.pid": 145476,
    "proc.ppid": 144577,
    "proc.vpgid": 177940,
    "proc.vpid": 177940,
    "meta.arch": "x86_64",
    "meta.kernel": "4.15.0-111-generic",
    "label": "RUN",
    "m_type": "PROCESS",
    "e_time": "2020-07-29T17:56:39.570064464+00:00",
    "filename": null,
    "k8s.node.name": "[redacted]",
    "file_events": [],
    "children": []
  }
]
```

Fig. 5 Example of mutation events.

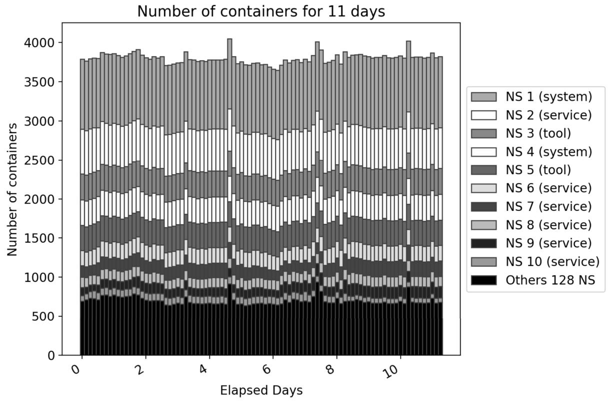


Fig. 6 Number of containers in a large-scale cluster over time.

5. Large-scale Cluster Implementation and Evaluation

5.1 Large-scale Cluster Implementation

5.1.1 Experimental Environment

In this work, we conducted a container integrity monitoring experiment on a large-scale Kubernetes cluster. Since a huge volume of system calls are expected to occur in a large-scale cluster, our implementation needs to consider processing speed and transfer method for data. In this section, we discuss how we implement our proposed approach in such an environment and discuss the evaluation.

Our experimental environment includes a large-scale cluster which is hosted by 160 nodes and 3,800 containers (on average) running in 138 Namespaces as seen in **Fig. 6**. This large-scale cluster serves as a development environment where DevOps cycles take place in an IT organization.

Our approach does not use any prior-knowledge on what kind of applications are running in the cluster or if there is any changes in certain currently running applications.

5.1.2 Evaluation Index and Placement

In a large-scale cluster, detecting mutation events by monitoring system calls in all nodes involves handling a huge amount of system call events. We implemented our proposed method in this type of large-scale environment. To make sure our proposed approach does not impact normal operations of the cluster we

Table 5 Scale of the experiment.

Item	*Number*
container (average)	3,800
pod (average)	3,000
node (average)	160
namespace	138
experiment period	11 days
Total monitored system calls	596,025,036

evaluated the impact on clusters with the following indicators.

5.1.3 Processing Delay

As seen in **Table 5**, in a cluster with 160 number of nodes, system calls occur at the rate of 2.3 million events/hour. To process such huge volume of events, first we consider the placement of processing pipelines and components. Let us assume t is the time taken for processing a single event filtering and T_{sys} is the time taken for system calls to occur in the entire cluster. To process all system call events in the cluster, t needs to be less than T_{sys}.

5.1.4 Distributed Processing

Kubernetes clusters require a connection to the Kubernetes API server to access the cluster configuration. To determine if system calls events are generated by the cluster system, our approach needs to use the command character string defined in the configuration information for a Pod. For this, our approach needs to request Pod information from Kubernetes API server. However, one API is generally used in the cluster. This API is provided at a single point even in a large-scale cluster. Therefore, if our approach makes a request for a huge volume of events, the operation of the API server will be greatly delayed, and in some cases it may cause the entire cluster to stop working.

In addition, detected mutation events are eventually stored in one central server, so the data storage component is also a single point. If there is too much access to it, data storing would stop.

To avoid such issues, we need to identify the component that is a single point and measure access to single point component in the entire cluster.

5.1.5 Data Volume

Due to the large number of system call events, data size is one of the issues to handle. The raw event output by the system call monitoring by Falco used in this paper is about 3 KB in size per event. As stated in Section 5.1.3, 2.3 million events/hour occur in the 160 node cluster.

Depending on the transfer method, 2,300,000 [/hour] × 3 [KB] = 144 [GB/day] of traffic will be generated in the worst case, which may also affect the operation of the entire cluster, as discussed in the previous section. Therefore, we use the following two point evaluation metrics for data transfer.

- The event filtering component is properly configured and can process data.
- The amount of data transferred to a single point that can affect the entire cluster is low enough

From the above three perspectives, we implemented our proposed approach with the architecture shown in **Fig. 7**. All event filtering components are placed in each node, so they can process all system call events inside their nodes without access to the other nodes. After the event filtering steps, the number of output (a.k.a mutation events) is very small, so it can effectively avoid the overloads at a single point.

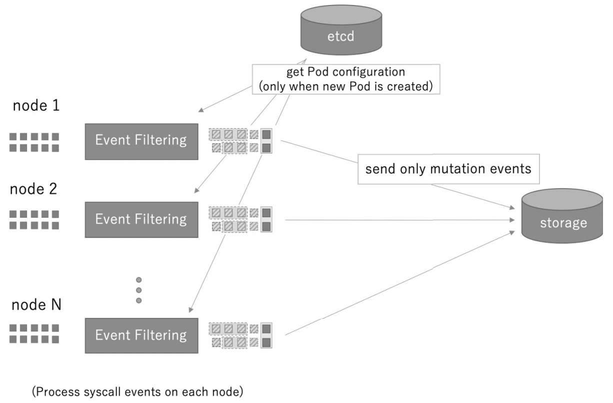


Fig. 7 Architecture to implement our proposed approach in the large-scale cluster.

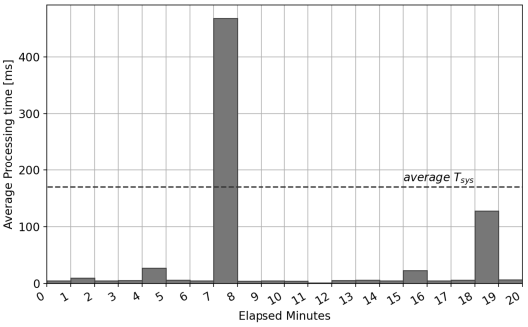


Fig. 8 Example of processing delay (at 1 node).

5.2 Experimental Results

5.2.1 Processing Delay

The event filtering component performs the flow of processing as shown in the pseudo code of Section 4, and each system call event contains the information necessary for that processing as an attribute in the following form. In addition, as described in Section 5.1.3, access to Kubernetes API occurs only once for a new pod event. Thus, the processing time order of 1 event is $O(1)$, or namely a constant time. Since it does not depend on the number of events, only the time to process single event matters.

Regarding the actual processing speed measured during the experiment, the average system call event generation period was 170 [ms], but our proposed approach can process a single event within 9 [ms] on average for the filtering step. Most of system call events are filtered out by our proposing filtering method (see discussion in Section 5.2.4), so the speed 9 [ms] for filtering is enough for processing events generated every 170 [ms].

5.2.2 Distributed Processing

First, regarding the requests to Kubernetes API server, very few requests are made to the K8s API server in the steady state (as seen in **Fig. 9**). This is because the request timing of the pod configuration information is set only when a new pod is detected, as stated in Section 5.1.4.

Second, concerning the access to data storage, a huge volume of events are filtered within a single node. Among 2.3 million events/hour, there are very few mutation events as shown in **Fig. 10**. Output frequency for data storage is also well distributed.

Although it is rare, some mutations events are sent to storage.

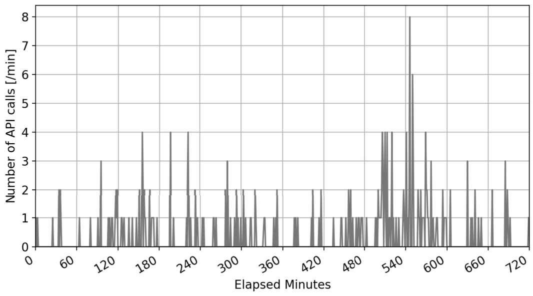


Fig. 9 Kubernetes API access over time.

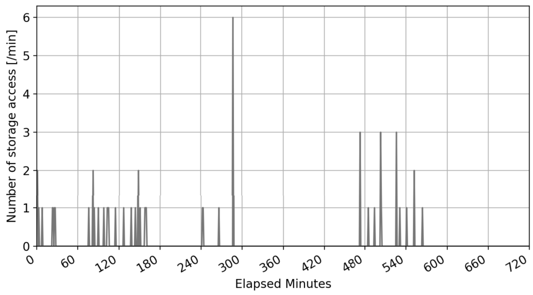


Fig. 10 Number of storage access over time.

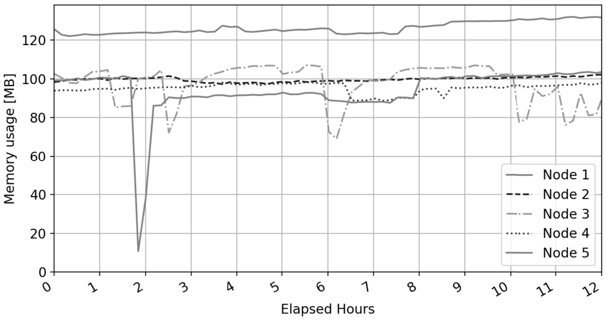


Fig. 11 Memory consumption for event filtering component over time.

In Section 6, we discuss what those detected mutation events are.

5.2.3 Data Volume

We evaluate data volume from two perspectives: a) amount of data that corresponds to system calls that occurred per node (host) b) data transfer rate for a single point.

First, per node, system call events occurred in amounts of 14 thousand [events/hour]. If the processing speed per event slows down as shown in Section 5.2.1, a large number of events can occur instantaneously. Therefore, we need to establish a buffer to handle such cases and need some other constant amount of memory for filtering itself. During the experiment, the overall memory size used by the filtering component was 132.7 [MB] at maximum. This memory consumption is small and will not impact the cluster operation. On the other hand, CPU usage was around 40 [milli core] on average with 2.0 GHz CPUs, and this is also small enough to allow other applications on a cluster to maintain their workload.

Next, regarding data transfer rate for a single point, from a single node, we check the transfer volume of mutation event data as seen in **Fig. 12**. For all nodes in the cluster, average data transfer size is 72.8 [KB/hour], which is small if one considers the data storage component. Even at a highest transfer size of 3.59 [KB/min], one could see that the transfer amount is sufficiently small.

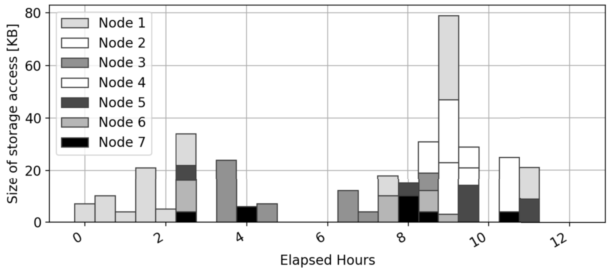


Fig. 12 Size of mutation events sent to storage.

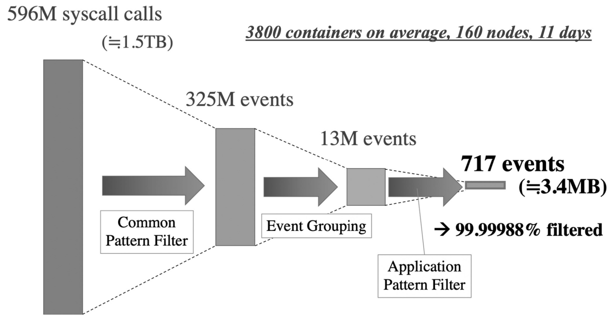


Fig. 13 Result of experiment - effect of event filtering.

5.2.4 Integrity Event Detection and Filtering Results

Figure 13 illustrates the event count and the effect of event filtering observed in this experiment. Even in large-scale cluster, our approach realized the same event filtering effect as it did with the test data described in Section 3. Covering the entire cluster, from 596 million system call events, our approach filtered almost 99.99988% of events that are expected events, and detected 717 mutation events that relate to container integrity.

In terms of data size, raw system call events amount to 1.5 TB, which were collected over a period of 11 days. However, the detected mutation events data correspond to only 3.4 MB.

Therefore, the proposed method and this implementation are suitable for long-term integrity monitoring of containers running a large-scale cluster.

6. Analysis of Detected Infiltrate Events

From 3,800 containers running on 138 Namespaces, only 18 containers on 12 different Namespaces were responsible for all of 717 infiltrate events. We analyzed the character string and the execution timing of the commands found in each infiltrate event occurring in the 18 containers. We found that they were roughly divided into the following two cases.

- Case A: Almost the same (or completely the same) commands are repeated
- Case B: Commands such as bash and sh are executed independently

Here, in Kubernetes, there are two types of commands, kubectl exec and kubectl cp, that can be performed externally on a running container. ‘kubectl exec’ executes the specified command in the container, whereas kubectl cp is a command for copying a file from the container to the host or from the host to the container. Based on these results, we categorize infiltrate events into two types.

- Type A: kubectl exec/cp is repeatedly executed by some

Table 6 Type A: Repeated commands in a container.

Event Time	*Command*
2020-XX-XX 21:00:36	tar cf - [dirctory]/jenkins-XXXXX-202-XXXXX
2020-XX-XX 15:35:13	tar cf - [dirctory]/jenkins-XXXXX-203-XXXXX
2020-XX-XX 21:02:02	tar cf - [dirctory]/jenkins-XXXXX-204-XXXXX
2020-XX-XX 18:46:46	tar cf - [dirctory]/jenkins-XXXXX-205-XXXXX

Table 7 Type B: Specific command execution with shell.

Event Time	*Command*
2020-XX-XX 19:18:35	bash
2020-XX-XX 19:18:44	ls
2020-XX-XX 19:18:48	ls -l
2020-XX-XX 19:20:21	cat [filename].sh
2020-XX-XX 19:21:03	vim [filename].sh

script etc.

- Type B: user executes kubectl exec/cp for some purpose.

Among the 18 containers in which infiltrate events occurred, 6 containers fall into type A, the rest fall into type B.

Since type A refers to repeated events, we could consider them to be allowlisted. However, the commands were executed on the container from outside the system unlike the commands defined in the application configuration information such as livenessProbe.

In addition, it is likely that the cluster administrator might not be aware what command is being executed. Considering that it may be an attack that executes a large number of same commands so even if the command is repeatedly executed, it is not defined in the configuration information of the running container. Therefore, this type of event is the primary target of our proposed approach for integrity monitoring.

An example of type A (repeated events) is shown in **Table 6**. One can see that a part of the command character string is a serial number, and similar commands are repeatedly executed. An infiltrate event with command "tar cf -" is a typical pattern of "kubectl cp", so this can be considered as a repetition of kubectl cp. Further, the string 'jenkins' appears as part of the command. We could infer that the events in this example were caused by Jenkins which is a tool that monitors and manages applications in the Kubernetes cluster.

Mutation events that are type B are distinctly different from the expected events. This type of event is the primary target of our proposed approach.

Table 7 shows an example of an event occurred due to a set of bash commands executed on a container by a user. User entered a container via kubectl exec and executed ls, cat, etc., and opened a script file with vim command. It is obvious that the user tried to change configuration of this container.

In this case, a user might view and edit a script file. Since we performed this experiment in a development cluster for business purposes, the example event above is thought to be caused by a developer and the malignancy is low. However, on a production cluster for services, if a malicious person infiltrates a container and accesses an important file (for example, the private key file of the Kubernetes cluster placed in the container), our approach could detect such actions as mutation events. This type of mutation event is the primary target for integrity monitoring and need to be detected.

Furthermore, we conducted an interview with a cluster administrator of the cluster. The administrator commented that the jenkins is actually used for this cluster, so detected infiltrate events with jenkins commands would be the actual external operation to containers, and the rest of the detected events indicate unknown intrusions to container. This is a clear evidence of the capability of the proposed approach.

7. Related Work

Prior work on monitoring system changes were primarily based on rule-based approaches [11], [15], [26]. System experts designed a list of targets (files and processes) and a set of rules for change identification. Then rule-based approaches monitored the existence of target files and their properties such as size of files, last modification time, and hash value of files etc. For instance, Refs. [5], [12], [22] used the files' last modification data and time and hard value to determine a file change event. The approaches in OSSEC [15] and Tripwire [12] detected the change events by a) storing the hash value, latest update date and time, and permission information of the monitored file and b) checking periodically whether there is a difference. Rule-based approaches have several drawbacks: a) maintaining rules for monitoring system changes is labor and resource intensive, b) good understanding of systems is required for experts, c) rules are hard to reuse in heterogeneous environment like cloud [25].

As an alternative, Refs. [13], [17], [19] proposed approaches based on monitoring system call events at operating system (OS) level. Jin et al. [11] proposed a similar approach that detects file change events using system calls in virtual environments (VMs). Here, the approach obtained the information about which user changed the file and when from the system calls.

Reference [1] proposed an approach that uses time and space efficient point-in-time copy and performs file system integrity checks to detect intrusions in storage systems.

For monitoring process executions events, there are methods that periodically acquire a list of processes and system calls. Since there existed a predefined list of processes to monitor, these methods were less resource sensitive than the approaches described in Refs. [7], [16].

Reference [3] proposed a "discovery by example" approach that autonomously searched and identified system changes in cloud. This approach automatically learned characteristic features of system changes without requiring any rule definitions or expert knowledge of underlying systems. This approach treated each system event independently and captured the changes associated to it. However, system changes may be represented by multiple system events in a cloud environment. Therefore, Ref. [25] introduced a practical system change discovery framework to capture system states on-demand, detect multiple system changes between them and form relationships among events.

There existed a wide range of commercial products. For instance, Solarwinds's Security Event Manager [23], Qualys's File Integrity Monitoring [21], Trustwave's Endpoint Protection [24], OSSEC [15] and Tripwire [12] for file integrity monitoring in cloud environment. We refer readers to Ref. [20] for detail comparison of currently available tools for file integrity monitoring and their underlying approaches.

Compared to rule based approaches, “discovery by example”, and on-demand approaches, our methodology applies a filtering approach to distill system call events associated to the expected behavior of containerized applications, thereby detecting mutation events that are anomalies.

8. Conclusion

In this work, we proposed a novel mechanism filter out default behavioral events in containers and detect mutations events that are anomalous. For this, based on our empirical studies on a small scale cluster, we built an approach for automatically generating allowlists that could efficiently filter out container expected events. In a large-scale cluster, we then demonstrated our proposed approach could not only process a huge volume of system calls from containers but also efficiently distill expected events to detect mutation events. From our investigations and the experimental results, on the large-scale cluster, we conclude that our approach could detect mutation events due to that fact it could process huge volume of system calls that occur at 144 [GB/day] efficiently from 160 nodes in real time and distill expected events in containers. In addition, we verified that our proposed approach could filter container execution (expected) events and efficiently detect mutation events with less impact on the cluster’s operations from processing speed, distribution and data size perspectives. The empirical studies conducted in a large-scale cluster demonstrated the importance role of filtering the expected events in container integrity monitoring. This type of filtering approach is indispensable for container integrity monitoring, especially for a large-scale Kubernetes cluster.

References

[1] Banikazemi, M., Poff, D. and Abali, B.: Storage-based file system integrity checker, *Proc. 2005 ACM Workshop on Storage Security and Survivability*, *StorageSS ’05*, pp.57–63, Association for Computing Machinery (2005).

[2] Borello, G.: System and application monitoring and troubleshooting with sysdig, USENIX Association (2015).

[3] Chen, H., Duri, S.S., Bala, V., Bila, N.T., Isci, C. and Coskun, A.K.: Detecting and identifying system changes in the cloud via discovery by example, *2014 IEEE International Conference on Big Data* (*Big Data*), pp.90–99 (2014).

[4] PCI Security Standards Council, LLC: PCI DSS quick reference guide (2016).

[5] Ding, B., He, Y., Zhou, Q., Wu, Y. and Wu, J.: hGuard: A framework to measure hypervisor critical files, *2013 IEEE 7th International Conference on Software Security and Reliability Companion*, pp.176–181 (2013).

[6] Dubois-Ferriere, H.: Introducing falco: Open source, behavioral security from sysdig (2016).

[7] Gupta, S., Sardana, A. and Kumar, P.: A light weight centralized file monitoring approach for securing files in cloud environment, *2012 International Conference for Internet Technology and Secured Transactions*, pp.382–387 (2012).

[8] Kitahara, H., Gajananan, K. and Watanabe, Y.: Highly-scalable container integrity monitoring for large-scale Kubernetes cluster, *2020 IEEE International Conference on Big Data* (*Big Data 2020*) (2020).

[9] Redhat Inc.: What is container orchestration? (2020).

[10] Redhat Inc.: What’s a linux container? (2020).

[11] Jin, H., Xiang, G., Zou, D., Zhao, F., Li, M. and Yu, C.: A guest-transparent file integrity monitoring method in virtualization environment, *Computers & Mathematics with Applications*, Vol.60, No.2, pp.256–266 (2010).

[12] Kim, G.H. and Spafford, E.H.: The design and implementation of tripwire: A file system integrity checker, *Proc. 2nd ACM Conference on Computer and Communications Security*, *CCS ’94*, pp.18–29, Association for Computing Machinery (1994).

[13] Ko, R.K.L., Jagadpramana, P. and Lee, B.S.: Flogger: A file-centric logger for monitoring file access and transfers within cloud computing environments, *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pp.765–771 (2011).

[14] Kubernetes: Kubernetes documentation (2020).

[15] Lazarevic, A., Kumar, V. and Srivastava, J.: *Intrusion Detection: A Survey*, pp.19–78, Springer US, Boston, MA (2005).

[16] Maddox, I.: Kubernetes simple file integrity monitoring (fim) container (2019).

[17] Quynh, N.A. and Takefuji, Y.: A real-time integrity monitor for xen virtual machine, *International conference on Networking and Services* (*ICNS ’06*), p.90 (2006).

[18] NIST: Fisma implementation project (2014).

[19] Patil, S., Kashyap, A., Sivathanu, G. and Zadok, E.: FS: An in-kernel integrity checker and intrusion detection file system, *Proc. 18th USENIX Conference on System Administration*, *LISA ’04*, pp.67–78, USENIX Association (2004).

[20] Peddoju, S.K., Upadhyay, H. and Lagos, L.: File integrity monitoring tools: Issues, challenges, and solutions, *Concurrency and Computation: Practice and Experience*, Vol.32, No.22, e5825 (2020).

[21] Qualys: File integrity monitoring (2020).

[22] Shi, B., Li, B., Cui, L. and Ouyang, L.: Vanguard: A cache-level sensitive file integrity monitoring system in virtual machine environment, *IEEE Access*, Vol.6, pp.38567–38577 (2018).

[23] Solarwinds: Security event manager (2020).

[24] Trustwave: Trustwave endpoint protection (2020).

[25] Turk, A., Chen, H., Byrne, A., Knollmeyer, J., Duri, S.S., Isci, C. and Coskun, A.K.: Deltasherlock: Identifying changes in the cloud, *2016 IEEE International Conference on Big Data* (*Big Data*), pp.763–772 (2016).

[26] Zlatkovski, D., Mileva, A., Bogatinova, K. and Ampov, I.: A new real-time file integrity monitoring system for windows-based environments, *Proc. ICT Innovations 2018*, ISSN 1857-7288, pp.243–258 (2018).

Hirokuni Kitahara was born in 1992. He received his M.E. degree from Waseda University in 2018. He joined the Information Processing Society of Japan in 2019. He is currently a research scientist at IBM Research - Tokyo. His research interest is cloud security technology.

Kugamoorthy Gajananan was born in 1981. He acquired his Ph.D. in Informatics at the National Institute of Informatics, also affiliated to the Graduate University for Advanced Studies. He has been working a research at IBM Research - Tokyo since 2014. His current research interests include security, compliance, and cloud.

Yuji Watanabe was born in 1973. He received Ph.D. in the Department of Engineering, University of Tokyo in 2001. He joined IBM Research in 2001. His research interests include security, compliance, and cloud platforms.