DDHCS: Distributed Denial-of-service Threat to YARN Clusters based on Health Check Service

Wenting Li, Qingni Shen, Chuntao Dong, Yahui Yang and Zhonghai Wu
School of Software and Microelectronics & MoE Key Lab of Network and Software Assurance, Peking University, Beijing, China

Keywords: DDoS, Hadoop, YARN, Attack Broadness, Attack Strength, Security.

Abstract: Distributed denial-of-service (DDoS) attack continues to grow as a threat to organizations worldwide. This attack is used to consume the resources of the target machine and prevent the legitimate users from accessing them. This paper studies the vulnerabilities of Health Check Service in Hadoop/YARN and the threat of denial-of-service to a YARN cluster with multi-tenancy. We use theoretical analysis and numerical simulations to demonstrate the effectiveness of this DDoS attack based on health check service (DDHCS). Our experiments show that DDHCS is capable of causing significant impacts on the performance of a YARN cluster in terms of high attack broadness (averagely 85.6%), high attack strength (more than 80%) and obviously resource utilization degradation. In addition, some novel schemes are proposed to prevent DDHCS attack efficiently by improving the YARN security.

1 INTRODUCTION

Hadoop is open source software based on scalability and reliability. It can be used to process vast amount of data in parallel on large clusters. Since then Apache Hadoop has matured and developed to a data platform for not just processing humongous amount of data in batch but also with the advent of YARN. It now supports many diverse workloads such as interactive queries over large data with Hive on Tez, realtime data processing with Apache Storm, in-memory datastore like Spark and the list goes on.

For Hadoop's initial purpose, it was always assumed that clusters would consist of cooperating, trusted machines used by trusted users in a trusted environment. Initially, there was no security model – Hadoop didn't authenticate users or services, and there was no data privacy (O'Malley et al., 2009); (Kholidy and Baiardi, 2012). When moving Hadoop to a public cloud, there are challenges to original Hadoop security mechanisms. However, the research on MapReduce and Hadoop has mainly focused on the system performance aspect, and the security issues seemly have not received sufficient attention.

A distributed denial-of-service (DDoS) is where the attack source is more than one–and often thousands–of unique IP addresses, it is an attempt to make a machine or network resource unavailable to its intended users, such as to temporarily or indefinitely interrupt or suspend services of a host connected to the Internet. The first DDoS attack incident (Criscuolo, 2000) was reported in 1999 by the Computer Incident Advisory Capability (CIAC). Since then, most of the DDoS attacks continue to grow in frequency, sophistication and bandwidth (Hameed and Ali, 2015); (Kholidy et al., 2015).

Previous work has demonstrated the threat and stealthiness of DDoS attack in cloud environment (Sabahi, 2011); (Durcekova et al., 2012); (Ficco and Rak, 2015). As a solution, (Alarifi and Wolthusen, 2014); (Karthik and Shah, 2014); (Mizukoshi and Munetomo, 2015) have successfully demonstrated how to mitigate DDoS attack with cloud techniques. There also have been numerous suggestions on how to detect DDoS attack. For example, using MapReduce for DDoS Forensics (Khattak et al., 2011), a hybrid statistical model to detect DDoS attack (Girma et al., 2015); (Lee, 2011). Unlike in cloud environment, DDoS attacks in BigData based on Hadoop/YARN environment are more aggressive and destructive, but there seems a lack of research.

One problem with the Hadoop/YARN system is that by assigning the tasks to many nodes, it is possible for malicious users submitting attack program to affect the entire cluster. In this paper, we study the vulnerabilities of Health Check Service in

Li, W., Shen, Q., Dong, C., Yang, Y. and Wu, Z.
DDHCS: Distributed Denial-of-service Threat to YARN Clusters based on Health Check Service.
DOI: 10.5220/0005741801460156
In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 146-156
ISBN: 978-989-758-167-0

YARN. These vulnerabilities encountered in YARN motivate a new type of DDoS attacks, which we call DDoS attack based on health check service (DDHCS). Our work innovatively exposes health check service in YARN as a possible vulnerability to adversarial attacks, hence it opens new avenue to improving the security of YARN.

In summary, this paper makes the following contribution.

- We present three vulnerabilities of Health Check Service in YARN, including i) Resource Manager (RM) is lack of Job Validation; ii) It is easy for a user to make a job failed, which will make the node transform into unhealthy state; iii) RM will add the unhealthy nodes to the exclude list, which means the decrease of service nodes in the cluster.
- We design a DDHCS attack model, we use theoretical analysis and numerical simulations to demonstrate the effectiveness of this attack for different scenarios. Moreover, we empirically show that DDHCS is capable of causing significant impacts on the performance of a YARN cluster in terms of high attack broadness (averagely 85.6%), high attack strength (more than 80%) and obviously resource utilization degradation.
- We propose three improving methods against DDHCS, including User blacklist mechanism, Parameter check and Map-tracing.

The rest of this paper is organized as follows. Section 2 discusses the background. Section 3 describes the vulnerabilities we found in YARN. Section 4 presents DDHCS attack model. Section 5 demonstrates implementation of our attack model and evaluates attack effect by MapReduce job. Section 6 contains our suggestion to strength security of YARN. Section 7 concludes the paper and discusses some future work.

2 BACKGROUND

Health Check Service is a YARN service-level health test that checks the health of the node it is executing on. ResourceManager (RM) using health check service to manage NodeManagers (NM). If any health check fails, the NM marks the node as unhealthy and communicates this to the RM, which then stops assigning containers (resource representation) to the node. Before we introduce health check service, we should know about RM component, NM States and triggering conditions.

2.1 YARN-ResourceManager

Hadoop has evolved into a new generation—Hadoop 2, in which the classic MapReduce module is upgraded into a new computing platform, called YARN (or MRv2) (Vavilapalli et al., 2013).

YARN uses RM to replace classic JobTracker, and uses ApplactionMaster (AM) to replace classic TaskTracker (Lee, 2011). The RM runs as a daemon on a dedicated machine, and acts as the central authority arbitrating resources among various competing applications in the cluster. The AM is "head" of a job, managing all life-cycle aspects including dynamically increasing and decreasing resources consumption, managing the flow of execution, handling faults and computation skew, and performing other local optimizations.

The NM is YARN's per-node agent, and takes care of the individual compute nodes in a Hadoop cluster. This includes keeping up-to date with the RM, overseeing containers' life-cycle management (Huang et al., 2014) monitoring resource usage of individual containers, tracking node-health, log's management and auxiliary services which may be exploited by different YARN applications.

There are three components connecting RM to NM, which co-manage the life-cycle of NM, as shown in Figure 1. They are NMLivelinessMonitor, NodesListManager and ResourceTrackerService. We discuss the three services as follows.

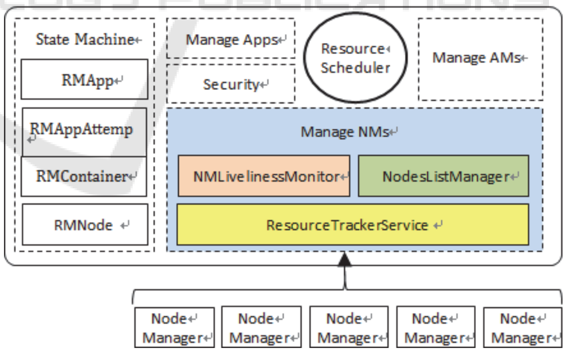


Figure 1: ResourceManager architecture.

1) **NMLivelinessMonitor:** This component keeps track of each NM's last one heartbeat time. Any DataNode that doesn't have any heartbeat within a configured interval of time, by default 10 minutes, is deemed dead and expired by the RM. All the containers currently running on an expired DataNode are marked as dead and no new containers are scheduling on it.
2) **NodesListManager:** This component manages a collection of included and excluded DataNodes.

It is responsible for reading the host configuration files to seed the initial list of DataNodes. The files are specified as "*yarn.resourcemanager.nodes.include-path*" and "*yarn.resourcemanager.nodes.exclude-path*". It also keeps track of DataNodes that are decommissioned as time progresses.

3) **ResourceTrackerService:** This component responds to RPCs from all the DataNodes. It is responsible for registration of new DataNode, rejecting requests from any invalid/decommissioned DataNodes, obtain node-heartbeats and forward them over to the Yarn Scheduler.

2.2 Node States

In YARN, an object is abstracted as a state machine when it is composed of several states and events triggering transfer of these states. There are four types of state machines inside RM—RMApp, RMAppAttempt, RMContainer and RMNode. We focus on RMNode state machine.

RMNode state machine is the data structure used to maintain a node lifecycle in the RM, and its implementation is RMNodeImpl class. The class maintains a node state machine, and records the possible node states and events that may lead to the state transform (Huseyin et al., 2015).

As shown in Figure2 and Table1, each node has six basic states (NodeState) and eight kinds of events that lead to the transfer of the six states (RMNodeEventType), the role of RMNodeImpl is waiting to receive events of RMNodeEventType type from the other objects, and transfer the current state to another state, and trigger another behavior at the same time. In subsequent articles, we focus on the *unhealthy* state and *decommission* state:

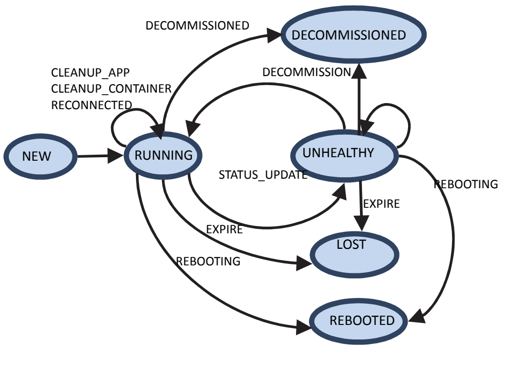


Figure 2: Node state machine.

UNHEALTHY: The administrator configures on each NM a health monitoring scripts, NM has a dedicated thread to execute the script periodically, to determine whether the NM is under *healthy* state. The NM communicates this "*unhealthy*" state to the RM via heartbeats. After that, RM won't assign a new task to the node until it turns to be *healthy* state.

DECOMMSSIONED: If a node is added to exclude list, the corresponding NM would be set for *decommission* state, thus the NM would not be able to communicate with the RM.

2.3 Health Check Service

The NM runs health check service to determine the health of the node it is executing on, in intervals of 10 minutes. If any health check fails, the NM marks the node as *unhealthy* and communicates this to the RM, which then stops assigning containers to the node. Communication of the node status is done as part of the heartbeat between the NM and the RM.

This service determines the health status of the nodes through two strategies, one is Health Script, Administrators may specify their own health check script that will be invoked by the health check service. If the script exits with a non-zero exit code, times out or results in an exception being thrown, the node is marked as *unhealthy*. Another one is Disk Checker. The disk checker checks the state of the disks that the NM is configured to use. The checks include permissions and free disk space. It also checks that the file system isn't in a read-only state. If a disk fails the check, the NM stops using that particular disk but still reports the node status as *healthy*. However, if a number of disks fail the check (25% by default), then the node is reported as unhealthy to the RM and new containers will not be assigned to the node.

Table 1: Basic states and basic events of node.

States	Describe	Trigger Events
NEW	The initial state of state machine	-
RUNNING	NM register to RM	STARTED
DECOMMISSION	A DataNode is added to exclude list	DECOMMISSIO N
UNHEALTHY	Health Check Service determines whether NM is unhealthy	STATUS_ UPDATE
LOST	NM doesn't heartbeat within 10 minutes, is deemed dead	EXPIRE
REBOOTING	RM finds NM's heartbeat ID doesn't agree with its preservation, RM require it to restart.	REBOOTING

We focus on the Health Script, we note that if the script cannot be executed due to permissions or an incorrect path, etc. then it counts as a failure and the node will be reported as *unhealthy*. The NM communicates this "*unhealthy*" state to the RM, which then adds it into exclude list. The NM will run this Health Script continuously, once the state is transformed into "*healthy*", RM will remove it from the exclude list, and reassign containers to the node. The administrator can modify the configuration parameter in yarn-site.xml.

3 VULNERABILITY ANALYSIS

3.1 Lack of Job Validation

The fundamental idea of MRv2 is to split up the two major functionalities of the JobTracker into separate daemons. The idea is to have a global RM and per-application AM. An application is a single job in the classical sense of Map-Reduce jobs.

Jobs are submitted to the RM via a public submission protocol and go through an admission control phase during which security credentials are validated and various operational and administrative checks are performed.

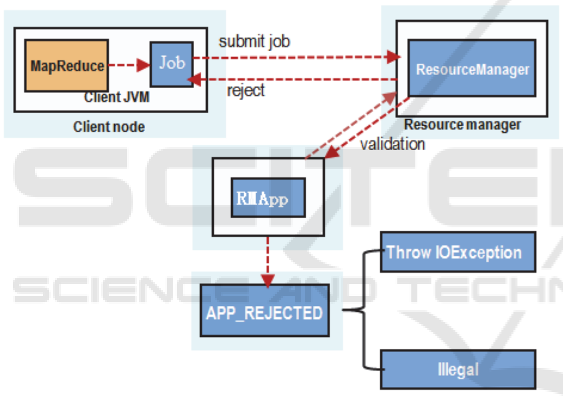


Figure 3: YARN rejects a MapReduce job.

RMApp is the data structure used to maintain a job life-cycle in RM, and its implementation is RMAppImpl class. RMAppImpl holds the basic information about the job (i.e. Job ID, job name, queue name, start time) and the instance attempts.

We found that only the following situations will lead to *APP_REJECTED* (an event of RMApp state machine) event, as shown in Fiture3:

1) The client submit a job to RM via RPC function *ApplicationClientProtocl#submitApplication* ma y throw an exception, it happens when Resource-Request over the minimum or maximum of the re sources;
2) Once the scheduler discovers that the job is illegal, (i.e. users submit to the inexistent queue or the queue reaches the upper limit of job numbers), it refuses to accept the job.

RM validates resource access permission, but lack of job validation about whether or not the job can finish. The only event that causes the job to enter the FINISHED state is the normal exit from the AM container. We can submit a job to the cluster which is bound to fail, RM allocates resources for it and it's running on corresponding NM. However, RM doesn't check whether the job can be successfully completed.

3.2 Easy to Make a Job Failed

The MapReduce enforces a strict structure: the computation task splits into map and reduce operations. Each instance of a map or reduce, called a computation unit, takes a list of key-value tuples. A MapReduce task consists of sequential phases of map and reduce operations. Once the map step is finished, the intermediate tuples are grouped by their key-components. This process of grouping is known as shuffling. All tuples belong to one group are processed by a reduce instance which expects to receive tuples sorted by their key-component (Wu et al., 2013). Outputs of the reduce step can be used as inputs for the map step in the next phase, creating a chained MapReduce task.

Each Map/Reduce Task is just a concrete description of computing tasks, the real mission is done by TaskAttempt. The MRAppMaster executes the Mapper/Reducer task as a child process in a separate JVM, it can start multiple instances in order. If the first running instance failed, it starts another one instance to recalculate, until this data processing is completed or the number of attempts reaches the upper limit. By default, the maximum attempts are 4 times. The users can configurate parameter in the job via *mapreduce.map.maxattempts* and *mapreduce.reduce.maxattempts*. MRAppMaster may also start multiple instances simultaneously, so they will complete data processing. In MRAppMaster, the life-cycle of the TaskAttempt, Task and Job are described by a finite state machine, as shown in Figure 4, where TaskAttempt is the actual task for the calculation, the other two components are only responsible for monitoring and management.
To our best knowledge, in some cases, the task never completes successfully even after multiple attempts. And it is easy to make the failed job, for instance, hardware failure, software bugs, process crashes and OOM (Out Of Memory). If there is no response from a NM in a certain amount of time, the MRAppMaster makes the task as failed. We

summarize the five conditions result in task failed as follows:

1) Map Task or Reduce Task fails. It means the problems of the MapReduce program itself which makes the task failed. There may be some errors in the user code.
2) Time out. It may be due to network delay to read data out of time, or the task itself takes longer time than expected. In this case, the long-running tasks take up system resources and will reduce the performance of the cluster over time.
3) The bottleneck of reading files. If the number of tasks performed by a job is very great, the common input file may become a bottleneck.
4) Shuffle error. If the map task completes quickly, and all the data is ready to copy for shuffle, it will lead to overload of threads and memory usage of buffer in the shuffle process, which will cause a shortage of memory.
5) The child process JVM quit suddenly. It may be caused by the bug of JVM, which makes the MapReduce code running failed.

We can easily make job failed using one of these items, for instance, we write program with an infinite loop, or we specify the timeout as 10 seconds, but submit a long-running job, which need at least 2 minutes.

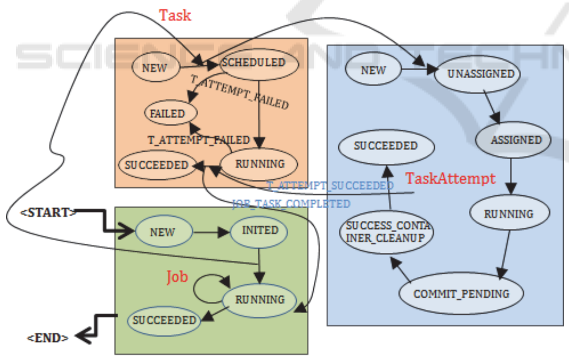


Figure 4: The job/task state transition.

3.3 Weak Exclude List Mechanism

As discussed in 2.3, NM runs health check service to determine the health of the node it is executing on. If the task failed more than 3 times in a node, the node is regarded under the *unhealthy* state. When a DataNode under *unhealthy* state, all the containers currently running on this DataNode are marked as dead and no new containers are scheduled on it. Explicitly point out the default failure times in the RMContainerRequestor class as follows:

```
   maxTaskFailuresPerNode =
conf.getInt(MRJobConfig.
   MAX_TASK_FAILURES_PER_TRACKER, 3);
```

NodesListManager maintains an exclude list - a file that resides on the RM and contains IP address of the DataNodes to be excluded. When NM reports its *unhealthy* state to RM via heartbeat, RM doesn't check why and how it becomes *unhealthy*, but adds it into exclude list directly.

Before that, RM calculates the proportion of the nodes in exclude list, which gets parameter information from MRJobConfig interface. When the node number of exclude list is less than a certain percentage (default is 33%), RM will add the node into exclude list, otherwise the *unhealthy* node won't be added to exclude list.

Finally, the failure handling of the containers themselves is completely left to the framework. The RM collects all container exit events from the NMs and propagates those to the corresponding AMs in a heartbeat response. AM already listens to these notifications and retries map or reduce tasks by requesting new containers from the RM.

4 DDHCS THREAT MODELS

The adversary is the malicious insider in the cloud, aiming to subvert availability of the cluster. As discussed in Section 3, we discovered three vulnerabilities of YARN platform, we can use the health check service to submit easy failed jobs to add DataNodes to exclude list, which will cause service degradation and the reduction of active DataNodes.

Considering the scenario in Figure 5, the normal users and malicious users can submit jobs to the YARN cluster. The jobs that normal users submitted can finish completely, while the jobs that malicious users submitted are the failed jobs, which will never complete. We use the running process of an applicat-ion to analyze the attack process. The steps are detailed as follows:

1) Distributed attackers and normal users submit applications to the RM via a public submission protocol and go through an admission control phase during which security credentials are validated and various operational and administrative checks are performed.
2) Accepted applications are passed to the scheduler to run. Once the scheduler has enough resources, the application is moved from accepted to *running* state. Aside from internal bookkeeping,

this involves allocating a container for the AM and spawning it on a node in the cluster.

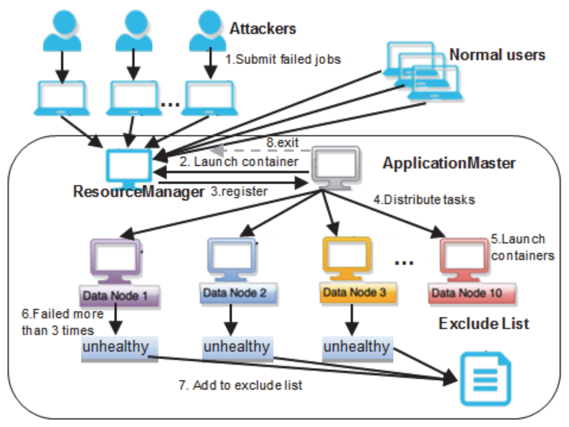


Figure 5: DDHCS: DDoS attack based on health check service.

3) When RM starts the AM, it should register with the RM and periodically advertise its liveness and requirements via heartbeat. To obtain containers, AM issues resource requests to the RM.
4) Once the RM allocates a container, AM can construct a container launch context (CLC) to launch the container on the corresponding NM. Monitoring the progress of work done inside the container is strictly the AM's responsibility.
5) To launch the container, the NM copies all the necessary dependencies to local storage. Map tasks process each block of input (typically 128MB) and produce intermediate results, which are key-value pairs. These are saved to disk. Reduce tasks fetch the list of intermediate results associated with each key and run it through the user's reduce function, which produces output.
6) If the task fails to complete, the task will be tried for a number of times, saying 3 times; if all tries fail, this task will be treated as a failure, and AM will contact RM to set up another container (possibly in another node) for this task, until this task is completed or the MapReduce job is terminated.
7) For each DataNode, which executes the failed task, its health check service will add one to its total number of failures. And if the DataNode has failed more than 3 times, the node will be marked as *unhealthy*. The NM reports this *unhealthy* state to the RM, which then adds it into exclude node lists.
8) Once the AM is done with its work, it should unregister from the RM and exit cleanly.

Attackers repeat the procedure until the exclude list has 33% nodes of the total number, aiming at reducing the service availability and performance by exhausting the resources of the cluster (including memory, processing resources, and network bandwidth).

5 EVALUATION

5.1 Experiment Setup

We set up our Hadoop cluster with 20 nodes. Each node runs a DataNode and a NodeManager with an Intel Core i7 processor running at 3.4 GHz, 4096 MB of RAM, and run Hadoop 2.6.0, which is a distributed, scalable, and portable system. All experiments use the default configuration in Hadoop for HDFS and MapReduce except otherwise noted (e.g., the HDFS block size is 128MB, max java heap size is 2GB).

A. Attack Programs

Attack Setting. We consider a setting in which attackers and normal users are concurrent using the same YARN platform. It is well known that YARN in public clouds makes extensive use of multi-tenancy. We design three attack programs as follows:

WordCount_A: We use WordCount benchmark in Hadoop as our main intrusion program because it is widely used and represents many kinds of data-intensive jobs. We specify the timeout parameter as 10 milliseconds (named as WordCount_A).

Since the input file we used is the full English Wikipedia archive with the total data size of 31GB, the program can't finish within the time limit.

BeerAndDiaper: We write an infinite loop in this program and specify the timeout parameter as 10 milliseconds, which will fail to complete within the time limit.

WordCount_N: We use an executable program, but as a normal user, we can modify the configuration file–*map-site.xml* in client. We change the value of *mapreduce.task.timeout* from 1000 (ms) to 10 (ms). We use the "*hadoop dfsadmin -refreshNodes*" command to reload the configuration file. We submit executable WordCount program (named as WordCount_N) with large input file, since it can't finish in 10 milliseconds, it will be marked as failed.

B. Evaluation Index

First, we introduce the variable to be used as follows. N denotes the total number of living nodes that a Hadoop cluster currently has; m denotes the number of *unhealthy* nodes after DDHCS attack. Here for simplicity, we assume that all of the nodes in a cluster are identical. T_{start} denotes the start time of the job, T_{finish} denotes the end time, then we calculate the total completion times under normal circumstances as $T=T_{finish}-T_{start}$, we repeat the jobs for 20 times, recording the start time and finish time, so we can obtain the average time under normal circumstances as :

$$\overline{T}=\sum_{i=1}^{n} T_i / n \qquad (1)$$

Similarly, we calculate the average time under DDoS attack as:

$$\overline{T}' =\sum_{i=1}^{n} T_i' / n \qquad (2)$$

Wherein, T' denotes the total completion times under DDoS attack, calculated by

$$T' = T'_{finish} - T'_{start} \qquad (3)$$

We can characterize the scale of the addressed DDHCS attacks in three dimensions: (i) attack broadness, which is defined as $b = m/N$; (ii) attack strength, denoted as s, which in the portion of resource occupied by the DDHCS attack in an infected node. For example, given attack broadness b=83.2%, and attack strength s=80%, a task will cost as $1/(1-s)$ (here 5) times long as usual to complete, with the probability of b. As shown in the follow, we can go through a mathematical derivation that attack strength is as follows:

$$s = (\overline{T}' - \overline{T}) / \overline{T}' \qquad (4)$$

(iii) resource degradation, we compare the CPU, memory occupancy rate and network bandwidth usage with and without DDHCS attacks, which can read from the job logs.

5.2 Evaluations

To verify the attack effectiveness of our approach, we test three programs mentioned above for evaluating attack broadness, attack strength and resource degradation. In the following section, we describe the details of the experimental records.

A. Attack Broadness

As we discussed in 5.1, N denotes the total number of living nodes that a Hadoop cluster currently has; m denotes the number of *unhealthy* nodes after DDHCS attack. We use $b = m/N$ to describe the attack broadness. We investigate a range of DDHCS intensities with three programs: WordCount_A, BeerAndDiaper and WordCount_N, running 100 times, 80 times, 60 times respectively. We can check the *unhealthy* nodes and decommission nodes in the cluster using the website http://master:8088/cluster/apps. We record the *unhealthy* nodes and *decommission* nodes after each DDHCS attack, as shown in Table.2.

As we can see in Table.2, the experimental results are the same as our research results. The decommission nodes represent the nodes which are added to exclude list, it accounts for less than 33% of total nodes. The average attack broadness of these three programs are 86.7%, 83.3%, 86.7% respectively, we can see that the cluster becomes unable to provide the services to its legitimate users.

B. Attack Strength

In this experiment, we run 4 benchmark applications to cover a wide range of data-intensive tasks: compute intensive (Grep), shuffle intensive (Index), database queries (Join), iterative (Randomwriter). We first run the 4 benchmark applications 20 times before the DDHCS attack to calculate the average running time, then we run three attack programs 100 times separately as three attack scenarios. After each attack scenarios we run each benchmark 20 times again to calculate the average running time after DDHCS attack.

Grep. Grep is a popular application for large scale data processing. It searches some regular expressions through input text files and outputs the lines which contain the matched expressions.

Inverted Index. Inverted index is widely used in search area. We implement a job in Hadoop that builds an inverted index from given documents and generates a compressed bit vector posting list for each word.

Join. Join is one of the most common applications that experience the data skew problem.

Randomwriter. Randomwriter writes 10GB data to each node randomly, it is memory intensive, CPU intensive and have high I/O consumption.

Firstly, we run each benchmark 20 times with no DDHCS attack to summarize the average running time $\overline{T}$. Then we run 100 times of the three attack programs WordCount_A, BeerAndDiaper, WordCount_N separately and record the running time of each legal benchmark application after each attack program. We summarize the average running time $\overline{T}''_{grep}$, $\overline{T}''_{inverted\ index}$, $\overline{T}''_{join}$, $\overline{T}''_{randomwrite}$ in Table.3, and analyze the attack strength. The result shows that under each type of DDHCS attack, the attack strength is more than 80 percent, and the cluster performance is more degraded.

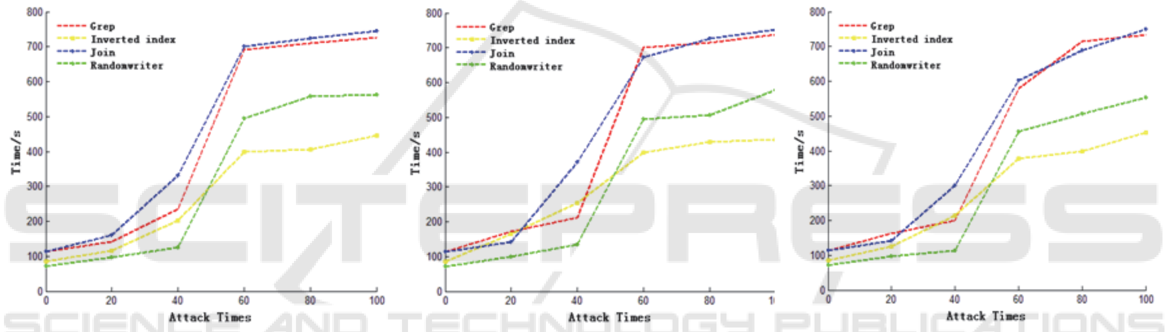
Figure 6 demonstrate the average running time of the 4 benchmark applications with the increase of

Table 2: Summary of DDHCS Attack broadness.

Job type	Times	Total nodes	Unhealthy nodes	Decommission nodes	Exclude list nodes rate	Attack broadness
WordCount_A	100	20	18	6	30%	90%
	80	20	18	6	30%	90%
	60	20	16	5	25%	80%
BeerAndDiaper	100	20	17	6	30%	85%
	80	20	17	6	30%	85%
	60	20	16	5	25%	80%
WordCount_N	100	20	18	6	30%	90%
	80	20	17	5	25%	85%
	60	20	17	5	25%	85%

Table 3: Summary of the attack strength of 4 benchmark applications.

		Average running time				Attack strength			
		Grep	Inverted Index	Join	Random writer	Grep	Inverted Index	Join	Random writer
Normal		112.4s	86.4s	113.6s	71.3s	0%			
DDoS	WordCount_A	726.7s	444.6s	745.2s	563.7s	84.5%	80.6%	84.8%	87.4%
	BeerAndDiaper	737.8s	435.1s	751.3s	579.2s	86.1%	80.1%	84.9%	87.8%
	WordCount_N	733.1s	453.3s	749.5s	553.8	84.7%	80.9%	84.8%	87.1%



(a)under WordCount_A DDHCS attack (b)under BeerAndDiaper DDHCS attack (c)under WordCount_N DDHCS attack

Figure 6: Job running time under 3 attack scenarios.

DDHCS attacks. We can see that as the increase of the attack program running times, the average running time of each benchmark applications prolonged significantly, which means the cluster is unable to provide service and the average time to access user request is higher than normal.

C. Resource Degradation

In order to demonstrate these results, we run additional experiments trying to compare the resources degradation. We simulated a scenario with BeerAndDiaper DDHCS attack. We run a range of attack program intensities: 20 times, 40 times, 60 times, 80 times and 100 times. The CPU, memory usage and network bandwidth usage before and after BeerAndDiaper DDHSC attack are illustrated in Figure 7, Figure 8.

In this scenario, most of the nodes are infected, and resource consumption has a significant rise and hence the YARN cluster performance is greatly deteriorated, which makes YARN become unable to provide the services to its legitimate users.

6 SUGGESTION AGAINST DDHCS

Recent work has proposed many methods to detect or prevent traditional DDoS attack, but these techniques are not suitable for Big Data platform (Gu et al., 2014); (Kiciman and Fox, 2005); (Specht and Lee, 2004). According to the vulnerabilities of our study, it is mainly because of legal users submitting malicious programs to launch attacks against YARN, we can't make defense by predicting user behavior. An important method to prevent DDoS attacks against YARN is to enhance the cluster. This requires a heightened awareness of security issues and prevention techniques from all YARN users.

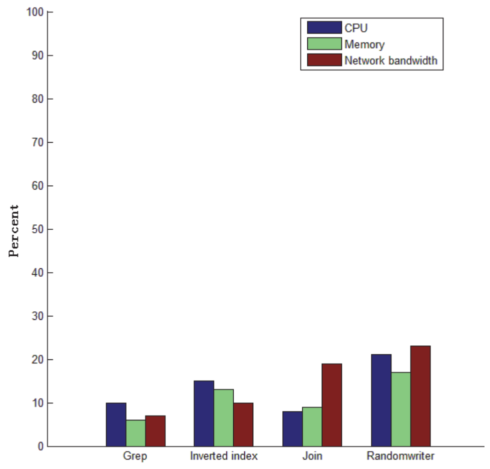


Figure 7: Summary of the CPU, memory occupancy rate and network bandwidth usage before DDHCS attack.

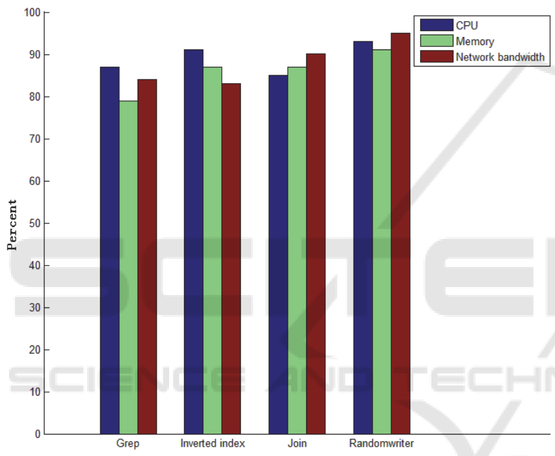


Figure 8: Summary of the CPU, memory occupancy rate and network bandwidth usage after BeerAndDiaper DDHCS attack.

Since the root of this problem is that there lack job inspecting mechanism by Hadoop/YARN, the most straightforward recipe is to verify whether the job succeeds within the time limitation. We proposed three methods to strength YARN security as follows:

User Blacklist Mechanism. Just like the node exclude list mechanism, we could construct user blacklist. As shown in Figure 9. When a user submitted jobs fail more than 3 times, the user is added into User blacklist. Every entry in the User blacklist includes the User ID, IP address of a blacklisted user, and a list of submitted jobs associated with this user. A user that matches an entry in the blacklists is placed on the isolated nodes running text and cannot distribute his jobs on the other nodes until he proves to be clean.

A user should not be blacklisted forever. A blacklisted user should be allowed to gain his/her rights back if it can be verified that the user's jobs are no longer failed. This is realized as follows. Each user in the blacklist is associated with a time-to-live value. Periodically each job submitted by the user runs test on the isolated nodes: if it still fails then the user's time-to-live value adds one, otherwise, it can finish successfully, the value is reduced by one. The user is removed from the User blacklist when its time-to-value is down to 0.

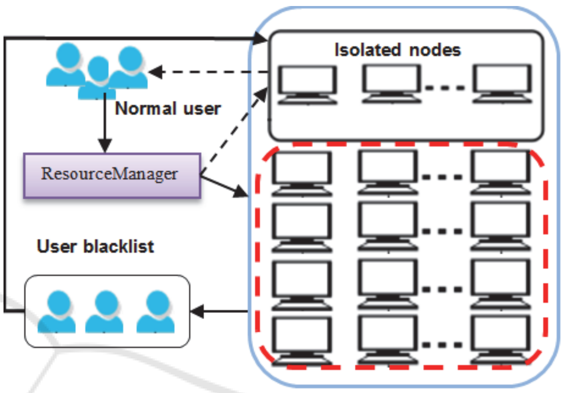


Figure 9: User blacklist mechanism.

Parameter Check. We all know that MapReduce program has a fixed structure. Consider the problem of WordCount in a large collection of documents, the user would write code similar to the following pseudocode.

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
EmitIntermediate(w, "1");
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

In addition, the user writes code to fill in a mapreduce specification object with the names of the input and output files and optional tuning parameters. So we can check these parameters before the program execution. If we find that some of the parameters are too high or too low compared with the normal value, the MapReduce program is not allowed to execute. For instance, the default execution time are 10 minutes, if the user specifies it as 10 milliseconds, this job will be rejected.

Map-tracing. Novel visualizations and statisti-cal views of the behavior of MapReduce programs enable users to trace the MapReduce program behavior through the program's stages. Also, most previous techniques for tracing have extracted distributed execution traces at the programming language level (e.g. using instru-mented middleware or libraries to track requests (Chen et al., 2002) (Barham et al., 2004); (Koskinen and Jannotti, 2008), we can learn from them and generate views at the higher-level MapReduce abstraction. Figure10 shows the overall flow of a MapReduce operation, we mainly focus on the Map phase.

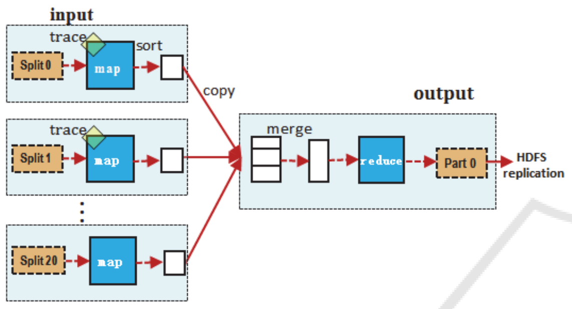


Figure 10: MapReduce execution overview.

We observed that, for each phase, the logs faithfully repeat the observed distributions of task completion times, data read by each task, size and location of inputs, probability of failures and recomputations, and fairness based evictions. So we can trace the first 10% maps for each job, and if the maps have some problems, such as, can't finish successfully, the cluster won't assign resources for the remaining tasks.

Finally, the DDoS attacks exist in multi-tenancy environment, so it is important for a user to learn the security and the resource usage patterns of other users sharing the cluster. It is necessary for rational planning the number of nodes that each user can use.

7 CONCLUSIONS

In this paper, we studied the vulnerability of YARN and proposed a DDoS attack based on health check service (DDHCS). We summarize three vulnera-bilities and design three attack programs to demonstrate how many nodes in a YARN cluster can be invaded by malicious users. We evaluate the attack effectiveness in a YARN cluster under DDHCS attacks. Our study shows that these vulnerabilities may be easily used by malicious users to launch DDHCS attacks and can cause significant impact on the performance of a YARN cluster. The highest 90% of the nodes deny of service and attack strength is more than 80%. Given this, we proposed three methods to enhance YARN. Regarding future research, we will move forward to strengthening the security of YARN, realizing our three suggestions, making good filter and defense. We will extend our trust calculus for estimating and optimizing the trustworthiness of cloud workflow for handing big data.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support of the National High Technology Research and Development Program ("863" Program) of China under Grant No. 2015AA016009, the National Natural Science Foundation of China under Grant No. 61232005, and the Science and Technology Program of Shen Zhen, China under Grant No. JSGG20140516162852628. Specially thanks to Ziyao Zhu and Wenjun Qian for the support of experiments.

REFERENCES

Alarifi, S., & Wolthusen, S. D. (2014, April). Mitigation of Cloud-Internal Denial of Service Attacks. *In Service Oriented System Engineering* (SOSE), 2014 IEEE 8th International Symposium on (pp. 478-483). IEEE.

Barham, P., Donnelly, A., Isaacs, R., & Mortier, R. (2004, December). Using Magpie for Request Extraction and Workload Modelling. In OSDI (Vol. 4, pp. 18-18).

Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., & Brewer, E. (2002). Pinpoint: Problem determination in large, dynamic internet services. *InDependable Systems and Networks*, 2002. DSN 2002. Proceedings. International Conference on (pp. 595-604). IEEE.

Criscuolo, P. J. (2000). Distributed Denial of Service: Trin00, Tribe Flood Network, *Tribe Flood Network 2000, and Stacheldraht* CIAC-2319 (No. CIAC-2319). CALIFORNIA UNIV LIVERMORE RADIATION LAB.

Durcekova, V., Schwartz, L., & Shahmehri, N. (2012, May). Sophisticated denial of service attacks aimed at application layer. *In ELEKTRO, 2012* (pp. 55-60). IEEE.

Ficco, M., & Rak, M. (2015). Stealthy denial of service strategy in cloud computing. *Cloud Computing, IEEE Transactions on*, 3(1), 80-94.

Girma, A., Garuba, M., Li, J., & Liu, C. (2015, April). Analysis of DDoS Attacks and an Introduction of a Hybrid Statistical Model to Detect DDoS Attacks on Cloud Computing Environment. *In Information*

Technology-New Generations (ITNG), 2015 12th International Conference on (pp. 212-217). IEEE.

Gu, Z., Pei, K., Wang, Q., Si, L., Zhang, X., & Xu, D. LEAPS: Detecting Camouflaged Attacks with Statistical Learning Guided by Program Analysis.

Hameed, S., & Ali, U. (2015). On the Efficacy of Live DDoS Detection with Hadoop. arXiv preprint arXiv:1506.08953.

Huang, J., Nicol, D. M., & Campbell, R. H. (2014, June). Denial-of-Service Threat to Hadoop/YARN Clusters with Multi-Tenancy. *In Big Data (BigData Congress), 2014 IEEE International Congress on* (pp. 48-55). IEEE.

Huseyin Ulusoy, Pietro Colombo, Elena Ferrari, Murat Kantarcioglu, Erman Pattuk. (2015, April). GuardMR: Fine-grained Security Policy Enforcement for MapReduce System. ASIA CCS'15.

Karthik, S., & Shah, J. J. (2014, February). Analysis of simulation of DDOS attack in cloud. *In Information Communication and Embedded Systems (ICICES)*, 2014 International Conference on (pp. 1-5). IEEE.

Khattak, R., Bano, S., Hussain, S., & Anwar, Z. (2011, December). DOFUR: DDoS Forensics Using MapReduce. In Frontiers of Information Technology (FIT), 2011 (pp. 117-120). IEEE.

Kholidy, H., & Baiardi, F. (2012, April). CIDS: a framework for intrusion detection in cloud systems. In Information Technology: New Generations (ITNG), *2012 Ninth International Conference on* (pp. 379-385). IEEE.

Kholidy, H., Baiardi, F., & Hariri, S. (2015). DDSGA: A Data-Driven Semi-Global Alignment Approach for Detecting Masquerade Attacks. *Dependable and Secure Computing*, IEEE Transactions on, 12(2), 164-178.

Kiciman, E., & Fox, A. (2005). Detecting application-level failures in component-based internet services. *Neural Networks*, IEEE Transactions on, 16(5), 1027-1041.

Koskinen, E., & Jannotti, J. (2008, April). Borderpatrol: isolating events for black-box tracing. *In ACM SIGOPS Operating Systems Review* (Vol. 42, No. 4, pp. 191-203). ACM.

Lee, Y., Kang, W., & Lee, Y. (2011). A hadoop-based packet trace processing tool (pp. 51-63). Springer Berlin Heidelberg.

Lee, Y., & Lee, Y. (2011, December). Detecting ddos attacks with hadoop. *InProceedings of The ACM CoNEXT Student Workshop* (p. 7). ACM.

Mizukoshi, M., & Munetomo, M. (2015, May). Distributed denial of services attack protection system with genetic algorithms on Hadoop cluster computing framework. *In Evolutionary Computation (CEC)*, 2015 IEEE Congress on (pp. 1575-1580). IEEE.

O'Malley, O., Zhang, K., Radia, S., Marti, R., & Harrell, C. (2009). Hadoop security design. *Yahoo, Inc., Tech.* Rep.

Sabahi, F. (2011, May). Cloud computing security threats and responses. *InCommunication Software and Networks (ICCSN)*, 2011 IEEE 3rd International Conference on (pp. 245-249). IEEE.

Specht, S. M., & Lee, R. B. (2004, September). Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures. *In ISCA PDCS* (pp. 543-550).

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., & Baldeschwieler, E. (2013, October). Apache hadoop yarn: Yet another resource negotiator. *In Proceedings of the 4th annual Symposium on Cloud Computing* (p. 5). ACM.

Wu, H., Tantawi, A. N., & Yu, T. (2013, June). A self-optimizing workload management solution for cloud applications. *In Web Services (ICWS)*, 2013 IEEE 20th International Conference on (pp. 483-490). IEEE.