Massive Distributed and Parallel Log Analysis For Organizational Security

Xiaokui Shu, John Smiy, Danfeng (Daphne) Yao, and Heshan Lin
Department of Computer Science Virginia Tech
Blacksburg, Virginia 24060
Emails: {subx, jsmiy, danfeng, hlin2}@cs.vt.edu

***Abstract*—Security log analysis is extremely useful for uncovering intrusions and anomalies. However, the sheer volume of log data demands new frameworks and techniques of computing and security. We present a lightweight distributed and parallel security log analysis framework that allows organizations to analyze a massive number of system, network, and transaction logs efficiently and scalably. Different from the general distributed frameworks, e.g., MapReduce, our framework is specifically designed for security log analysis. It features a minimum set of necessary properties, such as dynamic task scheduling for streaming logs. For prototyping, we implement our framework in Amazon cloud environments (EC2 and S3) with a basic analysis application. Our evaluation demonstrates the effectiveness of our design and shows the potential of our cloud-based distributed framework in large-scale log analysis scenarios.**

I. Introduction

Logs of computing systems, networks, and transactions contain rich information for security analysis. They can be used to diagnose faults and errors, as well as uncover past and ongoing anomalies. Network log analysis enables system administrators to reconstruct attack scenarios, e.g., [1], or identify unexpected network events [2]. File system log analysis allows one to detect unauthorized file creation and execution, a common symptom of infection [3].

However, a large number of logs coming from different security systems becomes a huge burden to corporations and organizations. To effectively, efficiently and economically store and analyze the rapidly growing number of logs is a big challenge to many organizations. Existing security-related log analysis solutions typically assume centralized data storage and computing, which does not scale. This paper addresses this research question of how to analyze massive logs for security in a timely fashion.

To speed up the computation, parallelization of conventional Intrusion Detection Systems (IDS) has been explored on multi-core CPU, FPGA, and GPU, e.g., [4]. Another approach to speeding up is to distribute the workload to multiple nodes, then to harvest and aggregate the results, e.g., using MapReduce framework [5]. Simple log analysis tasks such as pattern matching are easy to be made parallel, since the analyses of events are independent. Complex tasks with dependencies (e.g., event correlation [6]) are usually more difficult to realize.

In this paper, we consider the distributed log analysis model to process a massive number of logs for security. Unlike MapReduce, which assumes a specialized file system for data storage, our model describes a more common cloud storage scenario. That is, the organizational logs are generated and stored in the cloud along with their services outsourced to the cloud [7]. We present a cloud-based framework and protocol for organizations to conveniently store and streamingly examine large-scale event logs. The framework is versatile; it supports pattern matching and counting types of security analysis. It can also be extended to support more complex correlation tasks.

Previous distributed security analysis systems, e.g., [5], [8], [9], are built on general distributed computing frameworks. Most of them emphasize on the effectiveness of making specific security tasks executed distributedly. These existing pieces of work demonstrate the feasibility of developing security applications on general distributed computing frameworks, especially on Hadoop [10]. However, the distributed file system may not be necessary for distributed log analysis tasks. It may reduces system performance, especially when the whole distributed environment is realized on Network File System (NFS) with a single drive pool.

In this paper, we design a practical, lightweight and specialized framework for security log analysis in the cloud, aiming to achieve the following design goals.

- *Usability* includes the ease of use, general log analysis task support and streaming log support.
- *Cloud compatibility* refers to having the advantages of cloud computing, e.g., availability, flexibility, economic efficiency, etc.
- *Scalability* refers to the ability to perform effectively with a large number of cloud instances.
- *Lightweight* requires the low overhead of the framework itself.

The first two authors contributed equally to the paper. This work has been supported in part by Security and Software Engineering Research Center (S^2ERC), a NSF sponsored multi-university Industry/University Cooperative Research Center (I/UCRC), and NSF grant CAREER CNS-0953638.

We present our design of the lightweight distributed security log analysis framework. Our contributions are summarized as follows.

- We utilize a two-level master-slave model to efficiently distribute, execute and harvest tasks. This architecture is specifically optimized for security log analysis. It adopts techniques such as log segmentation, task scheduling, etc.
- We design the streaming log analysis feature based on the appending file feature supported by the native file system. It avoids the "appending file" feature in complex distributed file systems, which is expensive to maintain and sometimes error-prone[1].
- We design our framework upon the computation-on-demand cloud environment. Customers pay for the analysis computation they need.
- We realize our framework in the mature and stable Amazon cloud environments (EC2 and S3). We demonstrate the ease of use in the environment.

In the following sections, we first introduce our lightweight distributed security log analysis framework design in Section II. Then we present the realization of our framework with a log analysis application in Section III. We further evaluate our framework in Amazon EC2 and S3 cloud environment in Section IV. In the end we give the conclusions and discuss our future work.

II. Design

Our proposed log analysis framework is for the cloud computing environment. We have two assumptions of the environments.

i) The logs are generated and stored in the cloud.
ii) The deployment cloud environment provides a comprehensive interface to manage its computation resources.

The first assumption is reasonable in terms of the trend to outsource services to the cloud from traditional IT department of companies and organizations. For example, [7] and [13] reported this change of logging perspectives. The second assumption is practical. It is one of the keys to the success of current commercial clouds. It guarantees that the computation power is easy to use and manage. This is a critical assumption in our design, only on which can our framework conveniently create and destroy analysis nodes at any time. This makes it possible to response to dynamically generated logs and to adjust processing speed to log generation speed on the fly.

A. Entities and Components

We define the basic unit of our distributed framework as a *node*, which is a machine, physical or virtual. There are three kinds of nodes in the framework, *master*, *computation slave*, and *storage slave*. There is only one *master node*, which is possessed by the user, or the security administrator. Other nodes are cloud nodes, which are created and owned by the cloud provider. The framework is responsible for configuring the *computation slave nodes* and *storage slave nodes*, and specifying the protocols so that nodes can communicate correctly and efficiently.

Master node (n_m) is the only command and control (C2) node in the whole framework. And it is the only interface to the user. It configures the other two kinds of cloud nodes, and communicates with them to perform the analysis. It retrieves log information (metadata) from *storage slave nodes*, programs the log segmentation and work balance of each *computation slave node*, distributes tasks to different *computation slave nodes* and harvests analysis results from them.

Computation slave nodes ($\{n_{cs}\}$[2]) are responsible for concrete log analysis tasks. There could be a large amount of them to analyze logs in parallel. Because of the advantage of cloud computation, their computation power is flexible to adjust according to the size of logs, the speed of log generations and the complexity of the analysis. They receive tasks from the *master node*. Then they retrieve log segments from *storage slave nodes* according to the tasks, normalize and start to analyze the logs.

Storage slave nodes ($\{n_{ss}\}$) are simple nodes used only to store logs. The distributed nature of the design boosts the power of concurrent log retrieval from the storage to the analysis nodes. The bandwidth of multiple links could be fully utilized to accelerate the transferring and minimize the overhead of I/O. In some cases, especially when the analysis is not complex, one can merge the *computation slave node* and the *storage slave node* into one. The analysis can be done on the node which minimizes data movement. In the streaming log analysis scenario, the *storage slave node* is responsible for reading in the latest log from its native file system and responding to the network request for the appended log.

B. Framework Architecture and Operations

With the three kinds of nodes, the architecture of our lightweight distributed log analysis framework is presented in Figure 1. Each edge indicates an abstract operation/step.

Step 1: RETRIEVEMETA is a communication step where the *master node* retrieves metadata of the logs stored at *storage slave nodes*. This may be a running service in the background when dealing with streaming logs.

Step 2: SCHEDULE is a local step run on the *master node*. The *master node* divides the task into task units and computes a scheme to assign them to each *computation slave node*. The scheduler

[1]Hadoop suffers from file appending bugs since several years ago [11], [12]

[2]$\{n\}$ denotes the whole collection of node type n.

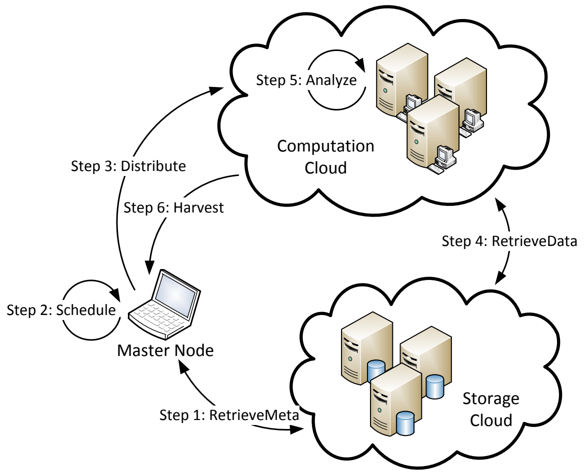

Fig. 1. The workflow of our distributed log analysis framework

considers the computation power of each *computation slave node* and its network condition to compute a good task splitting scheme. It is aware of the boundaries of the entries in logs through the metadata. The scheduler should also be able to deal with log generating speed vibration in streaming log analysis tasks. In another word, it should schedule proper amount of *computation slave nodes* to match the change of the speed of log generation.

Step 3: DISTRIBUTE is a management and communication step where the *master node* initializes each *computation slave node* and sends scheduled task units to them.

Step 4: RETRIEVEDATA is a communication step executed by every *computation slave node* to request the segments of logs stored at *storage slave nodes* with respect to their task units. One *computation slave node* may be assigned several task units, which correspond to several log segments. The step retrieves all log segments for all task units which are executed on the specific *computation slave node*.

Step 5: ANALYZE is a local step run on each *computation slave node* to analyze the segments of logs. It consists of two substeps in our design, i.e. log normalization and log analysis. This design guarantees the analysis of logs in various formats. There is no restriction on writing a log analysis payload (in each task unit) in terms of the analysis method and programming preference. The user should be aware of the cloud instance architecture and optimize the analysis payload for it.

Step 6: HARVEST is a step mixed with communication between *computation slave nodes* and the *master node*. It is also the step to merge the results harvested from *computation slave node* and to present the final result to the user.

To instantiate the framework with a concrete log analysis application, the user needs to prepare the analysis algorithm. She also needs to conceive how to perform the concrete analyzing steps in parallel. We do not intend to propose an artificial intelligent method to automatically parallelize an arbitrary log analysis algorithm in this paper. We minimize the complexity of SCHEDULE on the *master node* to make the framework feasible and friendly to users. The method of parallelizing an arbitrary log analysis algorithm and defining the task unit is left to the user in each specific deployment.

We show the six steps in our design as a linear procedure. However, they can also be performed in a cycle depending on the needs, e.g., in the situation of streaming log analysis. The *master node* performs Steps 1 and 2 in the background to compute how much computation power is needed for the streaming log analysis. In the meanwhile, the system executes Step 3 to Step 6 to obtain, analyze and report the latest logs analyzed, harvested and reported to the user.

With the lightweight design, we develop our distributed security log analysis framework in the cloud environment. Many usability issues, e.g., streaming log, are taken into account in the design process. The scalability goal is achieved by the lightweight design minimizing the management overhead, and we demonstrate the cloud compatibility of our framework by implementing and evaluating its prototype in the Amazon cloud environment.

III. IMPLEMENTATION WITH AN ANALYSIS APPLICATION

We realize our distributed log analysis framework in the commercial Amazon cloud environments, i.e., Elastic Compute Cloud (EC2) and Simple Storage Service (S3) clouds for computation and storage services, respectively. In our implementation, we develop an analysis application to calculate security event occurrences, and test our framework using it in Section IV.

A. Distributed Analysis Framework Realization

Using the Amazon AWS SDK for Java, we implement two components, one for the *master node* and the other for *computation slave nodes*. The former is run directly on n_m by the user, while the latter is distributed and executed on $\{n_{cs}\}$ which are in charge of n_m. The user is responsible for giving n_m the data (log) location in $\{n_{ss}\}$ and the number of *computation slave nodes*. n_m takes care of scheduling, creating $\{n_{cs}\}$, distributing task units, and harvesting results.

We adopt the log segmentation technique on $\{n_{ss}\}$, and we employ HTTP requests to transfer log segments between $\{n_{ss}\}$ and $\{n_{cs}\}$. This method directly stands on the native file system and avoids using *appending files* in a distributed file system, which makes our lightweight system stable and robust.

We describe the concrete operations implemented in our framework.

- n_m.RETRIEVEMETA($\{n_{ss}\}, l_m$)
  n_m requests the metadata, l_m, of all logs stored on the storage cloud, or the *storage slave nodes*, $\{n_{ss}\}$. The communication is based on HTTP protocol. l_m consists of
  - the URL of each log stored,
  - the number of entries in each log,
  - the average size of each entry for each log,
  - the range of every entry in each log.
- n_m.SCHEDULE($\{t_u\}, l_m, |\{n_{cs}\}|$)
  Given task units $\{t_u\}^3$, the log metadata l_m, and the number of *computation slave nodes* $|\{n_{cs}\}|$, the step computes a scheme assigning $\{t_u\}$ to $\{n_{cs}\}$. We assume all *computation slave nodes* are equal in terms of their computation power. The step yields a scheduling table $\mathbb{T}_s$, which maps every t_u to an n_{cs}. Each t_u consists of
  - t_{u_p}, an executable task procedure,
  - l_{u_l}, the URL of the corresponding log segment.
- n_m.DISTRIBUTE($\{t_u\}$)
  n_m creates all *computation slave nodes* $\{n_{cs}\}$ (initializes and launches the cloud instances) and distributes $\{t_u\}$ to them according to $\mathbb{T}_s$.
- n_{cs}.RETRIEVEDATA$_d$(t_u)
  Given the task units t_u assigned to it, each n_{cs} retrieves the log segment l stored at *storage slave nodes* according to l_{u_l} in t_u. When an n_{cs} is assigned to process multiple task units, multiple log segments should be retrieved.
- n_{cs}.ANALYZE(l, t_u)
  Every n_{cs} analyzes the log segment l using the analysis executable t_{u_p}. For general security log analysis, we implement a format unification procedure before the run of t_{u_p}, so that t_{u_p} can be written in a general way without considering the input log format. Using the free tier Amazon EC2 instance, we make use of the multi-processor property and implement a multithreading t_{u_p} in our realization. The result of each thread is merged to the node result.
- n_m.HARVEST($\{n_{cs}\}$)
  n_m harvests results from $\{n_{cs}\}$. n_m waits each n_{cs} for sending back the finish signal and the local result. For streaming log analysis, n_m does not wait for all $\{n_{cs}\}$ submitting their results before assigning new t_u to new n_{cs}. n_m assembles the local results retrieved from $\{n_{cs}\}$ to obtain the global result of the current overall analysis. The global result is then presented to the user.

B. Analysis Application: Security Event Occurrence Counter

Counting security events is the basic step in understanding the security status of a running system. More sophisticated statistical analyses can be performed further based on frequency and other collected information.

[3]It should be planned by the user paralleling the overall analysis task into task units, as it is discussed in Section II-B.

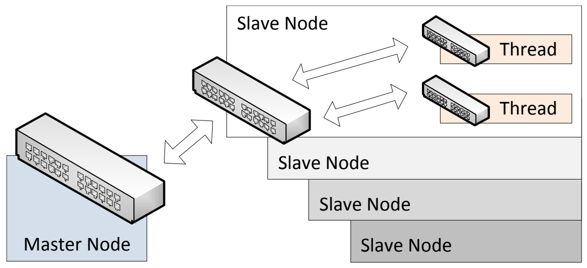

Fig. 2. Three-level hashmap used in our analysis application

In our case, we analyze HTTP connection logs and specify a bunch of connections with pre-defined destination IP addresses as the target. The analysis application counts these special entries in the logs. We parallelize the algorithm by splitting the counting onto different *computation slave nodes* for different log segments. Then the local results are sent back and assembled on n_m to give the overall number of occurrence of each special category of HTTP connections.

To efficiently count the event occurrence with large amount of task units, we use a three-level hashmap in the system to store and merge the analysis results. The lower two levels are built within each *computation slave node*, and they are defined and maintained by the task unit. The highest level is placed in the *master node*. The parallel nature of hashmap merging operation guarantees the ability and efficiency to distribute the analysis onto multiple *computation slave nodes*.

The design of our three-level hashmap is shown in Figure 2. The three levels are *i*) master hashmap, *ii*) slave hashmap and *iii*) thread hashmap. The structures of all hashmaps are the same. Each maps a *Key* field to a *Record* that contains all information needed to summarize all entries that share the same *Key*. In our case, the *Key* is defined as a destination IP address, and the *Record* contains the counter.

When a thread finishes its job, the result (thread hashmap) is merged into slave hashmap. When a n_{cs} terminates and reports back to n_m, its result (slave hashmap) is merged into master hashmap. After all *computation slave nodes* terminate and report, n_m composes and presents the overall security event occurrence report to the user.

IV. EVALUATION

We evaluate the correctness and performance of our framework in Amazon cloud environment. We use Amazon cloud EC2 and S3 to hold *computation slave nodes* and *storage slave nodes*.

We utilize the security event occurrence counter (explained in Section III) as a concrete security log analysis application in our evaluation. The logs to be analyzed

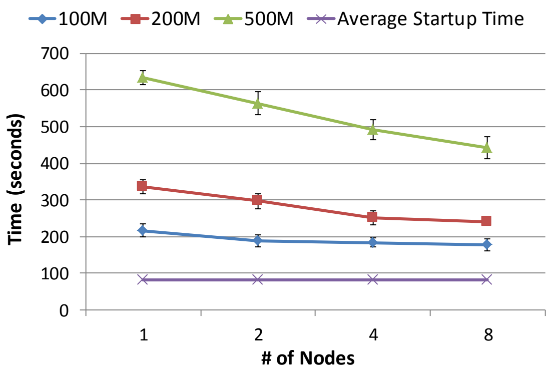

Fig. 3. Log analysis performance vs. number of nodes

consist of pure HTTP request entries. The target security event is defined as a category of HTTP sessions with four specific destination IP addresses. Both the framework and the application are written in Java. The framework uses Amazon AWS SDK for instance creation, management, and destruction.

Our prototype is easy to use. To setup the framework, the user prepares the *master node* by creating an access key pair and a security group in Amazon EC2 cloud. Every *computation slave node* is automatically created by the *master node*, initialized with the access key and placed in the appointed security group.

According to the computation power demand, the analysis throughput requirement, and the economic budget allowance, the user chooses both the type and number of Amazon EC2 instances to fulfill the analysis task. In our experiments, we test 1 to 8 *computation slave nodes* in Amazon EC2 cloud, and we store all logs on a single *storage slave node* in Amazon S3 cloud. This configuration minimizes the SCHEDULE operation and does not cause unfair comparison due to a complex and bad scheduler. All these nodes are the least powerful free tier micro instances (613 MB memory, up to 2 EC2 Compute Unit[4]).

In the evaluations, we use HTTP logs in three sizes, 100M, 200M and 500M. We verify the correctness of the analysis application. And we measure the running time of the whole process as well as the *computation slave nodes* initialization overhead.

The results of the framework performance are shown in Figure 3. Each marker on a line represents a group of 5 runs (mean and standard deviation). The y-axis shows the overall average execution time between *i)* the start of the analysis on the *master node*, and *ii)* the presentation of the result on the *master node*. The execution time is expected to decrease with respect to the increasing number of nodes along the x-axis. In the experiments, we find the computation power of free tier instances is not static. When massive instances are run by one user, the computation power of each decreases. We also find that the computation power and the network are not stable for EC2 free tier instances. Outliers are removed from the reported data.

In summary, our evaluation demonstrates the effectiveness of our lightweight distributed framework in analyzing security logs. The advantage of speedup is more apparent on larger dataset, which is expected. The system is designed to analyze large volume of logs.

Besides the demonstration of the analysis speedup achieved by our distributed framework, we also measure and analyze the overhead it introduces. The overhead is most caused by *instance initialization* and *data downloading*. We measure the former one on the *master node* and find it is nearly constant despite of the number of instances launched. We use the line *Average Startup Time* to illustrate this overhead in Figure 3. On the other hand, in our experiment, every *computation slave node* retrieves the log from the only *storage slave node*, which causes the downloading overhead. It can be mitigated by storing logs on multiple *storage slave nodes* and connecting them with *computation slave nodes* with higher overall network bandwidth.

V. RELATED WORK

Different from our lightweight distributed framework specialized for security log analysis, general distributed computing frameworks, e.g., MapReduce [14], are extensively investigated and developed in various big data analysis scenarios [15]. Some of its abstract applications, e.g., distributed data mining and distributed similarity joins, can be further developed for security log analysis.

There are several reports on running specific security analysis tasks on general distributed computing frameworks. MapReduce is tested with security applications such as port-breakdown [8], intrusion detection [5], botnet detection [9], spam classification [16] and spam detection [17]. One work is in-situ MapReduce [18], which proposes a MapReduce variant to start processing logs at the place where they are generated. It reduces the amount of information for further analysis. Most of the works conclude a significant speedup (e.g., 72% in [8], 89% in [5]) due to the use of multiple analysis nodes.

With these above evidences that general distributed frameworks are feasible for security analysis tasks, our lightweight framework aims at providing a practical and easy to deploy security framework with minimized distributing overhead. We realize our entire framework in the commercial Amazon cloud environment. We also show the setup procedure of our framework to demonstrate its ease of use.

In virus detection, cloud computing is adopted by both industry and academia to construct efficient and robust AV engines [19]. Commercial antivirus companies release cloud-based antivirus software e.g., [20] and [21].

[4]One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

[22] proposes a straightforward in-cloud analysis to reinforce the forensic power of antivirus systems. [23] goes further to execute suspicious programs in the cloud to help detect malware.

Our work differs from those cloud antivirus systems in terms of the analyzing objects and methods. Executables are meaningful to antivirus systems. They have properties (e.g., containing instructions, forming behaviors) that can be utilized by cloud antivirus systems. Log is a pure data type. Many sophisticated and specific techniques proposed in antivirus scan cannot be directly applied to distributed log analysis.

Data streaming is challenging in distributed computing framework design. [24] discovers the gap between the conventional cloud service and the streaming property of special applications. It proposes an idea of stream-oriented cloud. Based on Hadoop, [25] presents an architecture to handle multi-source streaming data analysis in the cloud. We also take into account the streaming log issue in our design. We utilize network connections to segment and pull streaming data from sources to analysis nodes.

VI. Conclusions and Future Work

We design and implement a lightweight distributed framework. Consisting a minimized set of components, the framework is different from general ones and is specifically designed for security log analysis. Features such as streaming log analysis are efficiently provided in the design. We realize the framework in Amazon cloud environment demonstrating its ease of use and the efficiency of the prototype. Our future work will be focused on extending the framework and supporting more complex log analysis applications for organizational security.

References

[1] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality." in *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS)*, 2005.

[2] H. Zhang, W. Banick, D. Yao, and N. Ramakrishnan, "User intention-based traffic dependence analysis for anomaly detection," in *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 104–112.

[3] K. Xu, D. Yao, Q. Ma, and A. Crowell, "Detecting infection onset with behavior-based policies," in *Network and System Security (NSS), 2011 5th International Conference on*. IEEE, 2011, pp. 57–64.

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

[5] S.-F. Yang, W.-Y. Chen, and Y.-T. Wang, "ICAS: An inter-VM IDS log cloud analysis system," in *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*. IEEE, 2011, pp. 285–289.

[6] H. Reguieg, F. Toumani, H. R. Motahari-Nezhad, and B. Benatallah, "Using MapReduce to scale events correlation discovery for business processes mining," in *Business Process Management*. Springer, 2012, pp. 279–284.

[7] R. Marty, "Cloud application logging for forensics," in *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 2011, pp. 178–184.

[8] Y. Lee, W. Kang, and H. Son, "An Internet traffic analysis method with MapReduce," in *IEEE Workshop on Cloud Management, Osaka*, 2010.

[9] J. Francois, S. Wang, W. Bronzi, T. Engel *et al.*, "BotCloud: Detecting botnets using MapReduce," in *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on*. IEEE, 2011, pp. 1–6.

[10] Apache, Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

[11] 2007, Append to files in HDFS. [Online]. Available: https://issues.apache.org/jira/browse/HADOOP-1700

[12] 2013, Enable sync by default and disable append. [Online]. Available: https://issues.apache.org/jira/browse/HADOOP-8230

[13] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter, "The netlogger methodology for high performance distributed systems performance analysis," in *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*. IEEE, 1998, pp. 260–267.

[14] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[15] K. Shim, "MapReduce algorithms for big data analysis," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2016–2017, 2012.

[16] G. Caruana, M. Li, and M. Qi, "A MapReduce based parallel SVM for large scale spam filtering," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, vol. 4. IEEE, 2011, pp. 2659–2662.

[17] W. Indyk, T. Kajdanowicz, P. Kazienko, and S. Plamowski, "Web spam detection using MapReduce approach to collective classification," in *International Joint Conference CISIS12-ICEUTE´ 12-SOCO´ 12 Special Sessions*. Springer, 2013, pp. 197–206.

[18] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ MapReduce for log processing," in *2011 USENIX Annual Technical Conference (USENIX ATC11)*, 2011, p. 115.

[19] I. A. AL-Taharwa, A. B. Jeng, H.-M. Lee, and S.-M. Chen, "Cloud-based anti-malware solution," 2011.

[20] Symantec, symantec.cloud. [Online]. Available: http://www.symantec.com/products-solutions/families/?fid=symantec-cloud

[21] T. Micro, trend Micro Smart Protection Network. [Online]. Available: http://cloudsecurity.trendmicro.com/us/technology-innovation/our-technology/smart-protection-network/index.html

[22] J. Oberheide, E. Cooke, and F. Jahanian, "CloudAV: N-version antivirus in the network cloud." in *USENIX Security Symposium*, 2008, pp. 91–106.

[23] L. Martignoni, R. Paleari, and D. Bruschi, "A framework for behavior-based malware analysis in the cloud," in *Information Systems Security*. Springer, 2009, pp. 178–192.

[24] J. Feng, P. Wen, J. Liu, and H. Li, "Elastic stream cloud (ESC): A stream-oriented cloud computing platform for rich Internet application," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. IEEE, 2010, pp. 203–208.

[25] M. Andreolini, M. Colajanni, and S. Tosi, "A software architecture for the analysis of large sets of data streams in cloud infrastructures," in *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*. IEEE, 2011, pp. 389–394.