2016 IEEE 2nd International Conference on Big Data Security on Cloud, IEEE International Conference on High Performance and Smart Computing, IEEE International Conference on Intelligent Data and Security

A Big Data architecture for security data and its application to phishing characterization

Pedro H. B. Las-Casas, Vinicius Santos Dias, Wagner Meira Jr. and Dorgival Guedes
Computer Science Department
Universidade Federal de Minas Gerais
{pedro.lascasas,viniciusvdias,meira,dorgival}@dcc.ufmg.br

***Abstract*—As the Internet grows, cybersecurity problems also arise. Different types of malicious activities have been explored by attackers. However, the existent defense mechanisms are not able to completely end the malicious threats, perpetuating this continuous arms race. The development of applications to mitigate those threats presents some complicating factors such as the growth in the amount of data, and the variety of data, that can come from different sources. In this paper we present an architecture built on top of Big Data frameworks that aims to mitigate cybersecurity problems such as spam and phishing and we show how it is being used to study spam and phishing collected using a global honeynet.**

***Keywords*-architecture, cybersecurity, spam, phishing, hadoop, spark**

I. Introduction

According to McAfee [1], the likely annual cost of cybercrime to the global economy is more than $400 billion. The cost of cybercrimes includes the damage to company performance and to national economies caused by cyber-attacks, and the effects of information theft. As reported by McAfee, in 2013, this types of incidents include more than 40 million people in the US. One estimate puts the total at more than 800 million individual records in 2013.

To face such attacks, we need to be able to identify and characterize them. A complicating factor in this case is the enormous increase in the volume of available data that has happened in recent years. Almost 90% of data in the world today were created in the last two years alone, and every day, approximately 2.5 quintillion bytes of data are created [2]. Security issues become more critical due to factors such as the large volumes and variety of data that may be vulnerable, the diversity of data sources and formats, and the velocity in which data are generated, typically following a stream nature with a high volume. Enterprises usually collect terabytes of security-relevant data, including network traffic, and software application events, among others [3]. However, well established techniques, most of the time, are not scalable and typically produce many false positives when dealing with large amounts of data, degrading their efficacy. To face these emerging problems, big data analytics has attracted the interest of the security community.

The use of big data frameworks for security solutions presents several benefits, such as the possibility of storing and using large quantities of security data. Although analyzing logs, network flows, and system events has been used for several decades in security solutions, conventional technologies are not adequate to be applied on such long-term, large-scale volumes. In general, the traditional infrastructure keep the data only for a limited period. Besides that, traditional techniques are inefficient when performing analytics and complex queries on large, unstructured datasets, while big data platforms perform these operations efficiently. However, so far solutions have been tailor-made, making their experience hard to adapt to other contexts.

In this paper we present an architecture for cybersecurity applications based on big data frameworks. Our architecture has the capability of collecting data from different sources, storing, combining, and processing them effectively. For example, sources like pcap files and other logs from a honeynet, data streams collected from black list sites and security-related search streams from social networks like Twitter, can all be stored in our system. Different algorithms, possibly implemented in different programming environments, can then be used to process combine data from different sources as needed.

To illustrate its use, we implemented an application to process large volumes of spam traffic collected from all the world. Our application collects data from honeypots located in different countries and continents, and stores the messages in mailboxes on top of HDFS. In that way, all security-related data becomes available to applications using Hadoop and Spark. The main contribution of the application is its capability to identify phishing e-mails in a set of spam messages. Using Natural Language Processing (NLP) and Locality-Sensitive Hashing (LSH), to inspect the text present in the messages we were able to detect different phishing campaigns. Evaluating the performance of the application, our experiments showed that our tool presents a speedup of 8.7 and 13.17 in a cluster with 9 machines using Hadoop and Spark, respectively.

To describe our architecture and results, the remainder of this paper is organized as follows: Section II presents our architecture, and the results are presented in Section III. After that, Section IV describes related works, and Section V presents some conclusions and discusses future works.

978-1-5090-2403-2/16 $31.00 © 2016 IEEE
DOI 10.1109/BigDataSecurity-HPSC-IDS.2016.44

II. Our architecture

We proposed an architecture for cybersecurity systems and used it to implement a spam analysis application to identify phishing campaigns. The architecture must satisfy important properties when considering a general large-scale application, that are: (i) scalability, the performance must be scalable as input data grows; (ii) efficiency, since it aims to deal with security threats, a major property is the ability to analyze the data efficiently, being able to mitigate the proposed problem as fast as possible; and (iii) uniform programmability, in which the architecture must be able to deal with multiple kinds of data types (e.g. structured text, binary data, etc.) and from different sources. The architecture is depicted in Figure 1. It is composed by five parts: (i) data collection, (ii) storage, (iii) reader, (iv) processing and (v) visualization.

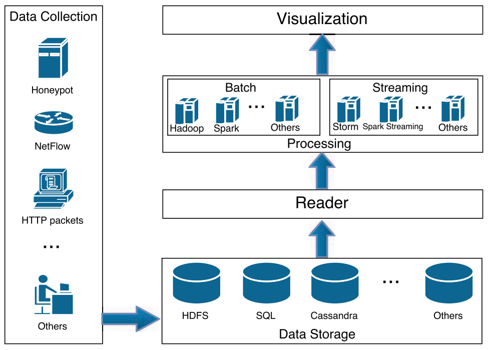


Figure 1. Architecture of our suggested solution.

A. Data collection

The first step of our architecture is the collection of data. The most important aspect of this step is to identify relevant data sources. Due to the constant advances of techniques used by attackers, it is increasingly necessary to use distinct sources of data in order to mitigate cibersecurity threats. For network security proposals, there are different possible data sources such as the network traffic packets, honeypot data, DNS information, NetFlow records, among others. Our architecture considers that different situations may require different data sources. In our example application, the goal was to analyze spam traffic, so we used low interactive honeypots installed around the world as data collectors. One of the main advantages of using honeypots to study spam is the direct interaction with the agent responsible for the abuse, making it possible to capture important information about it.

The honeypots used in this work emulate open proxies and open mail relays, machines on the network that are traditionally abused by spammers[1]. For each connection received by the honeypot, we stored the full content of each message, with original headers, as well as additional network information extracted at the time of the connection, which were stored as extended headers for each message. The collected messages are stored in mailboxes.

[1]http://spammining.speed.dcc.ufmg.br

B. Storage

Considering the heterogeneity of data that can be used by an application, it is important to have a storage strategy that will maximize performance and facilitate the use of the datasets. Every data source has different characteristics and is produced in a different volume and velocity, thus this is important to be considered in order to store them in the best possible way. One of the most used data storage is HDFS (Hadoop Distributed File System). It is a reliable and scalable distributed file system that provides high performance access to data. One important aspect of HDFS is its fault tolerance, which is obtained through replication, keeping multiple replicas of each data block in several different nodes. Another important aspect is that it is directly accessible from most Big Data processing frameworks.

C. Reader

Once the data is properly stored, we must enable the processing tool to process the input data. To guarantee scalability, HDFS is designed to allow parallel access to large files, so that different processing elements can work in parallel, reading from different parts of the file. The first step for that is to partition the data in large (128 MB) blocks, which may then be stored in multiple machines. If that blocking pattern happens to break a logical record across two blocks, the HDFS API must include enough information about the file data type to reconstruct those records.

Partitioning the blocks into logical records is performed by the Input Format associated with a file and the Record Reader for it. The Input Format tasks are (i) validate the input; (ii) split the input blocks and files into logical chunks; and (iii) create a Record Reader implementation to be used to create key/value. A Record Reader uses the data within the boundaries created by the input split to generate key/value pairs, that will be used by the mappers.

Hadoop environment offers several InputFormats, which go from simple line-based to JSON files. Unfortunately, some data frequently used in security analisys, like pcap files and mailboxes, are not treated the the standard API. Pre-processing such files to put them into a JSON schema, for example, is often not a reasonable solution. As our tool aims to process mailboxes, we implemented a new Input Format and Record Reader to handle a complete e-mail as a record, providing the correct interface for the processing tools. A mailbox is a simple text file that stores e-mail messages. If we used HDFS text file format to read them, however, we would get independent lines, with no notion of their order — messages would be completely scrambled. To avoid that, our architecture includes an HDFS extension for mailboxes. Our new Record Reader uses a regular expression to identify the beginning of an e-mail, and follows standard mailbox formatting rules to identify its header and the message contents, building an object for each message. It even crosses file block boundaries if necessary [4]. A similar procedure may be applied to other important file types, such as pcap and nfcap.

As we split a logical partition into records, our architecture's mail reader extracts information about the email, to be used in the processing phase. That allows, for example, that a Hadoop file be written to receive complete mail messages as objects to be processed by its *map* and *reduce* operators.

D. Processing

Once we have a scalable environment to store all types of data, from different sources, that may be of interest, we need a tool to process and analyze terabytes in a scalable manner, so we can achieve high performance. Recently, many different processing environments have been proposed for such tasks. In our current architecture, we use Hadoop and Spark, with their direct extensions (*e.g.*, like Hadoop's pig and hive, and Spark's GraphX, Streaming and SparkSQL). Both frameworks fit well with the operations required to analyze spam traffic, presenting reasonable performance. Others may be added when needed, since most big data frameworks are designed to integrate with HDFS and YARN, original Hadoop components.

Apache Hadoop is an open-source framework used for large scale processing of datasets on clusters of commodity hardware [5]. It is based on the MapReduce model, a programming paradigm used to simplify the expression of parallelism.With Hadoop, all data is treated as key-value pairs (K, V). The programmer must define a Map function that may transform each pair; the run-time system then groups all pairs with a same key and feed them as a list to a Reduce function, also provided by the programmer, which may process the list as a unit. By design, all mappers and reducers may execute independently, as soon as their inputs become available.

Although Hadoop has been widely adopted by the security community in many specific problems, it is not adequate when the application cannot be directly expressed as map-reduce tasks. To handle cases where that is the case, our architecture also includes Spark, which has a more flexible programming paradigm: instead of just map and reduce functions, users have access to a large set of data operators that may be applied to datasets in parallel [6]. Different from Hadoop, which depends on HDFS as source and destination of all map-reduce computation, Spark uses memory to store intermediate results, so it has better performance when executing iterative algorithms.

E. Visualizing results

Finally, after executing all operations, the application must be able to present it in a human-friendly form. Different visualization platforms may be added to the architecture easily using HDFS as a data repository, or by integrating them directly into the applications. Our analysis uses Python libraries, but other tools can be easily added.

III. Results

To illustrate the use of the architecture, we present tasks that would be performed during the initial steps of an analysis of phishing using our collected spam data. Experiments were conducted on a cluster of 10 machines, with one master and 9 slaves, performing storage and processing functions. Each scenario was executed five times, and averages are shown, with bars indicating the standard deviation in each case. Each machine has 4 CPUs, 8 GB of memory and 300 GB of disk.

A. Tuning: counting spams

Our first activity, to evaluate the scalability of the solution under different configurations, is a simple task: message count. More than the actual count, our goal was to validate our HDFS extension for mailboxes and to understand how elements like data replication in HDFS, the programming environment and the number of compute nodes used affected scalability, so we could find the best configuration for the cluster. We used both Hadoop and Spark versions of the count application, to assess the relative performance of the frameworks. The Hadoop implementation follows the organization of the classic word count application, using mappers to read each message and to issue a tuple with a fixed key and value 1 as output for each message. The reduce function just adds the values in the list for the fixed key it receives. The Spark implementation receives the mailboxes as its input dataset; we just have to apply the *count* operator on the dataset to get the answer. For these tests we used messages collected from February 01 to February 28, 2014. In that period, 93,891,539 messages were collected totalling approximately 268 GB, for an average of 9.57 GB collected per day.

Figure 2 shows the performance for the operation, using Hadoop and Spark. It presents the execution time, in minutes, when varying the number of compute nodes. For both frameworks, the experiment using fewer slaves (3) resulted in the worst performance, taking $\approx$57 minutes to complete for Hadoop, and $\approx$42 minutes for Spark. Increasing to 5 machines, the runtime falls to $\approx$36 minutes and $\approx$25, 1.57 and 1.68 times faster, for Hadoop and Spark, respectively. Using 9 machines, Hadoop's running time was $\approx$21 minutes, 2.74 faster than the 3-node scenario and Spark's was $\approx$14 minutes (3.05 faster). Another interesting result is obtained when compared to the sequential implementation. An efficient sequential program takes about 180 minutes to process all 268 GB of data, almost 8.7 and 13.17 times slower than Hadoop and Spark with 9 nodes, respectively. In all cases we see that both Hadoop and Spark scale well in our scenarios, although Hadoop is always a little below optimum speedup (which would be 9.0 in this case), and Spark a little above that. In that case, the super linear speedup is due to Spark's better use of caching in multiple levels of the system. This result shows the potential of the solution on scaling the number of machines and their applicability to real data.

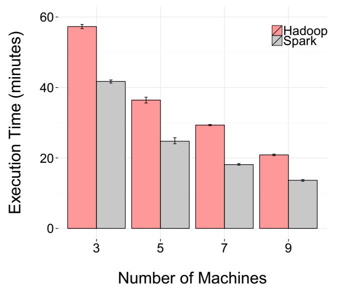


Figure 2. Performance varying the number of compute nodes.

One important element in a large scale storage system like HDFS is how it balances reliability and performance by creating multiple copies of the data. The HDFS replication factor can be configured by the user and this affects both the reliability of storage as the reading speed. A factor of 1 means no replication (and therefore no reliability in the case of a machine failure); a factor of 3 is the default configuration. Figure 3 shows the impact of replication for the scenario with 9 compute nodes. Clearly, a replication factor of 1 shows the worst performance for both environments. This occurs because, as there is only one replica of each block, when a node makes a reading request, there is a greater probability that the same is not available locally, leading to an overhead due to the necessary networks transfer. To show that, Fig. 3 also indicates, at the top of each bar, the number of non-local accesses (NL) incurred in each case. Using replication, HDFS tries to satisfy a read request with a block replica that is closer to the requesting node, and we can see that with 2 copies the number of no-local accesses already drops markedly, to the point that a further increase in replication does not provide visible effects in execution time. Users must consider, however, that the system reliability is further improved with 3 copies of each file block.

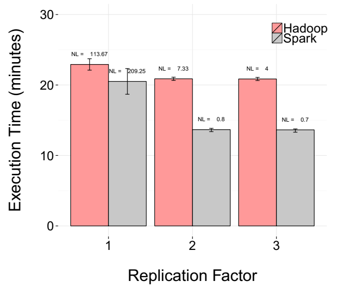


Figure 3. Replication effects. NL = # **N**on-**L**ocal accesses, on average (of a total of 2,151 blocks).

Hadoop divides data in file splits, that are read completely by some mapper. The granularity it uses to define splits and assign them to nodes is limited by the scheduler. On the other hand, Spark sees the file as a direct mapping of a distributed data structure that it can handle just as it is laid across compute nodes. That difference in approach makes its scheduling of computation more efficient in terms of data location, causing very few non-local accesses, one of the reasons for Spark's better performance with more copies.

B. Counting spams per recipient

Counting messages is basically a direct access application, since it just has to read data from storage once and count the objects found. Another operation that makes a more elaborate use of the architecture is to count the number of spams that would be sent to each recipient in our data. In that case, e-mails from the mailboxes must be read by the Record Reader, grouped by the destination addresses in their headers and then counted. That operation maps directly to the Map-Reduce execution model in Hadoop, and can be easily implemented by a reduceByKey operator in Spark.

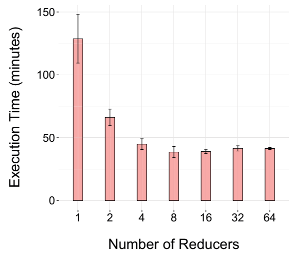
One important element in both Hadoop and Spark is the configuration of the system to make sure it makes the best use of the parallelism in each execution. In Hadoop, the user must set the number of Reducers it will use, which may be chosen basically in terms of the number of machines that will run them, considering each node may run one reducer (or at most a few of them) per core. Spark, on the other hand, may be configured with the number of partitions it must divide the data into before processing, and that is related to the total number of file blocks involved (although multiple blocks can be grouped in a single partition, or a single block may be split into multiple partitions). This experiment intended to check how sensitive each framework was to their configuration parameters.

Figure 4(a) shows the impact when varying the number of reducers from 1 to 64, using Hadoop, and varying the number of partitions from 541 to 10009 in Spark (prime numbers are recommended). Hadoop clearly is very sensitive to its configuration and execution times drop approximately four times from one reducer to eight (close to the number of machines) and then stay more or less stable. Spark, on the other hand, is much more resistant to configuration variations in this case and always outperforms Hadoop.

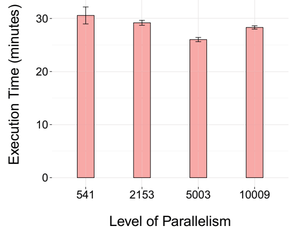
Comparing Hadoop and Spark, we observe that Spark obtained better results in both operations. Although we find Spark more efficient than Hadoop in these experiments, we highlight the flexibility of our architecture in dealing transparently with different processing environments. That is useful if the user decides for example to use an existing application already implemented with Hadoop.

C. Phishing identification

For our phishing application, the first step to identify phishing campaigns is to differentiate phishing messages



(a) Variation of the number of reducers using Hadoop.



(b) Spark's level of parallelism.

Figure 4. Sensitivity of each framework to configuration parameters.

from other types of spam, since our dataset contains all kinds of spam messages (but only spam messages, so there is no concern here about false positives, a legitimate user message being classified as spam). In general, phishing attacks attempt to steal sensitive information of the victim, such as passwords, credit card number, among others, trying to convince the user that the message is legit. Therefore, we implemented a method from the literature [7] to identify messages attempting to steal user's information. Our method uses natural language processing techniques to identify key features of each message to generates a score, enabling the identification of a phishing message by its differentiated scores. Those features are discussed as follows.

Direct message: The first feature of our method is to verify if the message is a message addressed directly to one person or if it is a generic message sent to more people. Usually, direct messages, referring to the user by his/her name, are used in phishing to try to make the message sound more legitimate. Identifying this feature helps in the phishing identification. Our approach to detect if the spam is a direct message is to check if the message's number of recipients is equal to one.

Mention of money: One way often used by attackers to convince users to reply their emails is to promise easy money. Once the victim believes that the email is legit and that exists the chance to get "free money", they usually answer the message, sending their information. Considering this characteristic, we analyze the whole content of the message in order to identify any mention of money, trying to find words such as "money", "cash", "dollar", among others, and any money related symbol like "$", "R$", "CAD$", etc.

Sense of urgency: the phishing attacker tries to induce the victim to answer the message as soon as possible. One of the reasons is that, when a person is under pressure or an urgent situation, it usually lacks the logical reasoning, tending to make hasty decisions. Another reason is because the sooner the user answers the message, the smaller will be the probability of the message or his URL's to be blacklisted. This feature is identified in our tests if the content of the spam presents any word with a sense of urgency. The set of words to consider was created using some key words, such as "urgent", "immediate", "desperate", etc., and their hyponymy, identified using WordNet[2].

Reply request: The last feature we identify in the message is the presence of a reply request. If the phishing attacker intends to obtain sensitive information from the user, it will only happen if the user answers the message. Thus, the attacker tries to convince the victim to reply to the e-mail. Here, we also used a set of words created using WordNet, based on key words like "reply", "response", "answer", "contact", etc.

If the message presents three of the features described above, we consider it as being a phishing message. Once we analyze the whole set of spams, the next step is to detect the phishing campaigns in the selected messages. This operation is performed using a Locality-Sensitive Hashing (LSH) algorithm [8]. It represents similarities between objects using probability distributions over hash functions, where the hash collisions capture the objects similarity. Similar objects are considered to be in the same phishing campaign.

We implemented the operations described above using Spark and performed the executions on top of the same cluster as used in the first two experiments. In this case, we used only spam with English content. The data was collected using honeypots located in Australia, Brazil, Chile, Hong Kong, Netherlands, Norway, United States and Uruguay, from April 01 to June 30, 2015. The dataset is composed by ≈19 million e-mails, totalling around 60 GB.

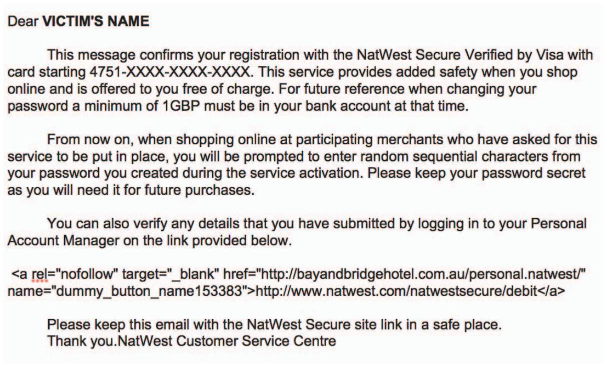
Our results highlighted 212 phishing campaigns, with a total of 775,115 messages, representing ≈4% of the spam. On average, each of those campaigns have 3,656 e-mails, sent by 1.71 different IP address, present in 1.44 and 1.26 Autonomous System and Country Code, respectively. The biggest campaign found presents more than 57 thousand messages sent by only 2 different IP addresses from 2 distinct ASes. Figure 5 presents part of an example extracted from that campaign. All other e-mails sampled from that group are similar to this one. As can be seen, the attacker tries to convince the victim that it is a legitimate message from a bank, and tries to to induce the victim to access his bank account to steal his information. Despite the message appearing legitimate, we can see that the URL is not from the bank in question, but a page created to deceive the victim.

In order to evaluate if our method was able to correctly classify the messages into phishing campaigns, we evaluated manually all campaigns generated. From the total of 212, only 4 were incorrectly classified, an accuracy of 98.1%. We are currently researching other rules that can be used to widen the set of possible phishing campaigns that can be identified by our system.

IV. Related Works

Several studies have proposed parallel and distributed solutions for network security problems. Using Hadoop,

[2]http://wordnet.princeton.edu


Dear **VICTIM'S NAME**

This message confirms your registration with the NatWest Secure Verified by Visa with card starting 4751-XXXX-XXXX-XXXX. This service provides added safety when you shop online and is offered to you free of charge. For future reference when changing your password a minimum of 1GBP must be in your bank account at that time.

From now on, when shopping online at participating merchants who have asked for this service to be put in place, you will be prompted to enter random sequential characters from your password you created during the service activation. Please keep your password secret as you will need it for future purchases.

You can also verify any details that you have submitted by logging in to your Personal Account Manager on the link provided below.

<a rel="nofollow" target="_blank" href="http://bayandbridgehotel.com.au/personal.natwest/" name="dummy_button_name153383">http://www.natwest.com/natwestsecure/debit</a>

Please keep this email with the NatWest Secure site link in a safe place.
Thank you.NatWest Customer Service Centre

Figure 5. Sample message from the largest phishing campaign identified.

for example, François *et al.*, proposed a scalable solution called BotCloud [9]. Regarding the spam detection problem, Caruana *et al.* evaluates the use of Hadoop to spam filtering [10], showing the their approach is scalable and efficient. Considering phishing, Marchal *et al.* [11] proposed PhishStorm, an automated phishing detection system using Storm. The system uses URL features in order to identify the ones that can be considered as phishing. LARX is another parallel and efficient anti-phishing solution [12]. It extracts the URLs contained in the network trace and checks them for phishing using Google Safe Browsing API.

Matatabi [13] presents a big data platform for cyber-threat analysis that is aimed to create a more complete system that allows the detection of complex security threats involving multiple data sources and location of attackers. Even though we implemented an application related to phishing and spam problems, the proposed architecture can be used to build applications to fight different cybersecurity problems.

V. Conclusion and Future Work

The proliferation of data sources and data collecting structures has lead to a large increase in the data available for cyber-security experts. To process such large volumes of data, scalable massive data processing solutions are needed.

In this paper we introduced an architecture that enables the implementation of Big Data applications to be used in the context of cybersecurity. As a case study, we developed an application that aims to process spam traffic using HDFS, Hadoop, Spark, and data collected from honeypots spread in different locations of the world. We were able to demonstrate the power of our application, from the implementation of simple operations to be used in the analysis of spam traffic, to more complex operations such as phishing detection.

Our experiments showed that our method can correctly detect phishing campaigns, presenting an accuracy of 98.1%. Regarding the performance of the application, we obtained a speedup of 8.7 and 13.17, for Hadoop and Spark, respectively, using 9 machines. We also note the importance of replication in data locality and its consequent gain in processing latency. So our architecture is effective to apply massive processing strategies in real applications that are sensitive to scalability. As future work, we intend to develop new applications using different Big Data frameworks, such as Storm.

Acknowledgments

The dataset used was collected and made available to us by CERT.br, part of NIC.br. This work was partially supported by NIC.br, Fapemig, Capes, CNPq, and InWeb.

References

[1] Center for Strategic and International Studies - McAfee, "Net Losses: Estimating the Global Cost of Cybercrime Economic impact of cybercrime II," *White paper*, June 2014.

[2] Y. Yu, Y. Mu, and G. Ateniese, "Recent advances in security and privacy in big data," *j-jucs*, Mar 2015.

[3] A. A. Cardenas, P. K. Manadhata, and S. P. Rajan, "Big data analytics for security," *IEEE Security & Privacy*, 2013.

[4] P. H. B. Las-Casas, V. Santos Dias, R. Ferreira, W. Meira, and D. Guedes, "A hadoop extension to process mail folders and its application to a spam dataset," in *International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, Oct 2014, pp. 108–113.

[5] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud'10*, Berkeley, CA, USA, 2010.

[7] S. Aggarwal, V. Kumar, and S. D. Sudarsan, "Identification and detection of phishing emails using natural language processing techniques," in *Proc. of the 7th Int'l Conference on Security of Information and Networks*. New York, USA: ACM, 2014.

[8] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, vol. 99, 1999, pp. 518–529.

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

[10] G. Caruana, M. Li, and H. Qi, "Spamcloud: A mapreduce based anti-spam architecture," in *Int'l Conference on Fuzzy Systems and Knowledge Discovery*, 2010.

[11] S. Marchal, J. Francois, R. State, and T. Engel, "Phishstorm: Detecting phishing with streaming analytics," in *Network and Service Management, IEEE Transactions on*, Dec 2014.

[12] T. Li, F. Han, S. Ding, and Z. Chen, "Larx: Large-scale anti-phishing by retrospective data-exploring based on a cloud computing platform," in *ICCCN 2011*, July 2011.

[13] H. Tazaki, K. Okada, Y. Sekiya, and Y. Kadobayashi, "Matatabi: Multi-layer threat analysis platform with hadoop," in *BADGERS*, 2014.