EagleEye: A Logging Framework for Accountable Distributed and Networked Systems

Nandhakumar Kathiresshan, Zhifeng Xiao, and **Yang Xiao**
Department of Computer Science,
The University of Alabama
Tuscaloosa, AL 35487-0290 USA
Emails: nkathiresshan@crimson.ua.edu, zxiao1@crimson.ua.edu, yangxiao@ieee.org

***Abstract*— We propose EagleEye, an accountable logging framework as a middleware for distributed and networked systems. EagleEye offloads the logging function from the distributed application program so that applications can focus on the logic handling without worrying about when and how to do logging. By capturing and analyzing network packets, EagleEye is able to reproduce the entire networking event history in the application layer, which is the basis of implementing an accountable system. We provide a case study by replacing the logging component of PeerReview [1] with EagleEye. The evaluation result shows that EagleEye can achieve equivalent accountability without modifying the host application program, which can save numerous workloads of modifying, republishing, and redeploying the host software.**

I. INTRODUCTION

Logging is an essential component in computer/network systems since it provides the first-hand information of system events. The traditional way of logging is to implant code into the software in which interesting events might occur. The benefits of this method is that the system developer can handily control when, where, and how to do logging. However, this way makes logging become a functional module coupling with the system. Once there are new requirements on logging, the entire system will have to be modified, republished, and redeployed. For instance, PeerReview [1] intended to enable accountability for distributed/networked systems by detecting Byzantine faults. PeerReview is designed as a software library which can be used by different distributed applications that need accountability support. To actually use PeerReview, those software has to be modified in programs to add PeerReview specific functionality. Additionally, PeerReview and its attached application form one-to-one relation, meaning that PeerReview has to make equivalent number of copies when there are multiple applications in need of the support of accountability. The above PeerReview case motivates us to re-think the way of logging.

In this paper, we propose EagleEye, which is a logging framework for building accountable distributed and networked systems. EagleEye decouples the process of logging and its management from the host application so that both two can vary independently. By employing packet-capture technique to collect all network packets, EagleEye is able to obtain the raw information of network events. After filtering and analyzing, the entire network event history can be reproduced for a specific network application. On top of EagleEye, multiple distributed applications can co-exist without caring about logging, because EagleEye's responsibility is to deal with log generation, filtering, and processing, etc.

Fig. 1 shows the sketch of EagleEye framework, which is compared with PeerReview. We can see that EagleEye resides in the application layer, acting as a middleware between high level applications and low level network stack. EagleEye captures all incoming and outgoing traffic as raw data of log files. The applications are unaware of the underlying actions performed by EagleEye when their network events are all recorded and categorized. On the basis of EagleEye's log files, accountability can be offered as a third party software. For example, we can build a tracing component to identify the node/hop where a message was lost [21]; we can also design a truth-finding protocol to discover some specific events happened in the system (e.g., how a confidential file was leaked to the network [22]); It is also convenient to implement the Byzantine fault detection strategy [1] with EagleEye. In other words, it forms orthogonal relationship between various host applications and the supporting accountability software. The design approach makes the entire system more extensible and maintainable.

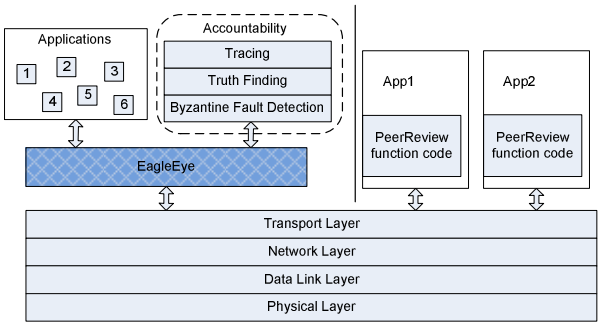

Fig. 1 EagleEye and PeerReview framework

On the other hand, there are also tools for packet capturing such as wireshark [23] and other methods [24]. However, these packet capturing tools cannot incorporate with applications closely to provide accountable functions since they only focus on capturing the low level packets. EagleEye leverages these tools by transforming the raw packet information to high level application messages, which can be selectively logged to provide accountability support.

The rest of this paper is organized as follows: we have reviewed related literatures in Section II. The problem is stated in Section III. The EagleEye system design is presented in Section IV. In section V, we evaluate EagleEye with a case

978-1-4244-9920-5/11/$26.00 ©2011 IEEE

study. We conclude the paper in Section IV.

II. Related Work

In this paper, we regard logging as an essential technique in designing an accountable system. Our literature review will then focus on 1) accountability in distributed and networked systems, 2) general logging techniques, and 3) how logging helps to build accountable computer/network systems.

A. Accountability

Accountability implies that an entity should be held responsible for its own actions or behaviors [14]. It has been a longstanding concern of trustworthy computer systems [5], and researchers have recently elevated accountability to be a first class design principle for dependable networked systems [7, 8]. Accountability has been applied to various distributed and network contexts. Yumerefendi *et al*. present CATS [9], a network storage service with strong accountability properties. AudIt [10] is described as an accountability interface that determines which unit is responsible for dropping or delaying the traffic. Mirkovic *et al.* propose a perfect accountability scheme in [11] as an alternative to the operational accountability in AIP. Re-ECN [12, 13] allows accurate congestion monitoring throughout the network, thus enabling the upstream party at any trust boundary in the internetwork to be held responsible for the congestion that it causes or allows to be caused. Apart from the accountable systems designed for different contexts, Xiao *et al.* [15] proposed P-Accountability, which is a general approach to quantify and evaluate the degree of system accountability. In the above systems, logging is widely used to support building system accountability.

B. Logging

Log files are very important for the management of computer systems for the reason that they are the original source of event history in the systems [16]. Typically, operating systems such as UNIX platforms provide logging service to applications by running general-purpose logging software, e.g., syslog, which is configurable. Applications send information to the syslog process which will store the messages in a text file (log file). In a distributed computing environment, multiple approaches are developed for generating log files. In a typical cluster, log files are stored in a dedicated server which receives messages from other hosts within the cluster [17]. A log analysis tool called Picviz is proposed in [19] for the purposes of spotting network attacks. SALSA [20] is a centralized approach of log analysis; it collects log files from all nodes in a distributed system and output the state-machine views of the history of systems' executions along with related statistics, from the log files. NetLogger [18] is a lightweight tool kit for visualizing the information in log files. In this paper, we only focus on network events since they can be reproduced by inspecting the network traffic. One limitation of EagleEye is that it is not applicable to systems in which network-irrelevant events are required.

C. Logging and Accountability

Creditable log files are the raw evidence when determining responsibility in accountable systems. PeerReview [1] detects Byzantine faults by replaying a tamper-evident log file kept in each machine. Flow-Net [6] aims at providing better accountability for the entire computer/network system with a comprehensive logging. Both of them use the traditional method to do logging, which is lack of decent system maintainability and extensibility. EagleEye intends to make up this weakness by decoupling logging and the host applications.

There are other related work in security [25-35].

III. Problem Statement

To demonstrate the applicability of EagleEye, throughout the rest of this paper, we will provide a case study of detecting general Byzantine faults with PeerReview [1].

A. Byzantine Fault detection problem

There are a number of ways in which distributed systems can be attacked. The general faults are termed as Byzantine Faults [2]. By definition, a Byzantine fault is an arbitrary fault that arises during the execution of an algorithm by a distributed system. Specifically, they are the faults in which a defective node might damage its logical condition and send random messages intended to destabilize the system. Detecting these faults and then removing them if they exist instead of masking them have proven to be cheaper and comparatively efficient. This is done by simply attaching a detector to each node in the system. If the detector finds that a given node is defective, then it alerts the corresponding application software which subsequently takes suitable measures. This has been the traditional fault detection method so far.

Due to the demand caused by these threats, we need a detector which supports accountability. In such cases, these detectors should not only detect the fault but should also know which node has performed the faulty action and also be able to inform the whole system about that particular node/information. The system should be complete and accurate. Completeness means that, whenever any node is found to be faulty by a correct node, the system generates evidence against at least one faulty node; by accuracy, we mean that it never produces legitimate evidence against a correct node.

B. Assumptions

We have made a few assumptions for EagleEye:

A1: All assumptions that have been made in PeerReview [1]. Since our case study is following the design of PeerReview. We have to follow its assumptions. They are:

- The application-relevant protocols are deterministic.
- A message sent from one correct node to another will be eventually received.
- The hash function used to build tamper-evident log is pre-image resistant, second pre-image resistant, and collision resistant.
- Each machine has public/private keypair bound to a unique node identifier.
- Each node has access to a reference implementation, which can be used to replay logs and detect faults.

- A function w will be used to map each node i to its witnesses. We assume that there is at least one correct node in $\{i\} \cup w(i)$.

A2: EagleEye has access to the various encryption keys managed by the applications when there are needs to view the contents of encrypted messages.

A3: we assume that each application has a unique identifier to be separated from each other.

IV. Design and Prototype Implementation

A. Design

The generalized working flow of Eagle Eye is as follows:

- Each machine in the distributed system has EagleEye installed. Each machine has either a single or multiple applications that will be monitored to be accountable.
- EagleEye logs all incoming and outgoing traffic generated by these applications on each node. The raw traffic data will then be processed to become human-readable log files.
- Each application has one set of witness/witnesses which periodically checks its correctness.
- Witnesses check the correctness of a node by replaying the log segment of that node.

EagleEye takes advantage of the fact that all network events will go through the network interface as packets, meaning that the packets are the only source through which the information is sent/received. Packet capturing therefore plays a vital role in our framework. For referential purposes, all captured packets were saved in tcpdump's capture format (*.pcap). Pcap files give the information sufficient for experimentation. But not all systems can have a network protocol analyzer installed in them to view the Pcap files and not all users can handle them efficiently. In our implementation, all the captured packets were stored in K12 text file format (*.txt) (shown in Fig. 2). All captured packets stored as text files were in complete hexadecimal form with timestamps and a few special characters in between the packets and frames. Each packet starts with a timestamp followed by the interface, indicating the method through which the packets were captured. In this case it was Ethernet and hence was denoted by "ETHER". The pipe symbol ("|") is used to separate hex values. Note that every packet has a single "0" at the beginning; this represents the beginning of the packet.

There might be one or more applications, which are not related to one another. As mentioned above, a network protocol analyzer is used to capture the packets and store them locally.

Eagle Eye has a set of witnesses (protocols) which check the correctness of the nodes. Having recorded the packets, Eagle Eye first converts the raw Pcap files into formatted text. The text file is formatted in such a way that each value can be individually read. Then Eagle Eye separates information from each protocol using the identifier.

Once the packets have been separated according to their protocols, they are stored in separate files to make the log files more manageable. In this way, if the user is aware of which protocol is malfunctioning, they can ask the witnesses to check only for that particular protocol which in turn saves time and memory.

Fig. 2 A sample Pcap file in K12 text file format

The witnesses have full access to the logs stored in the nodes and know the identifier of each node. When the user requests for the witnesses to check a node's credibility, the witnesses run a replay machine which takes the log from the source and replays the entire action performed by the node.

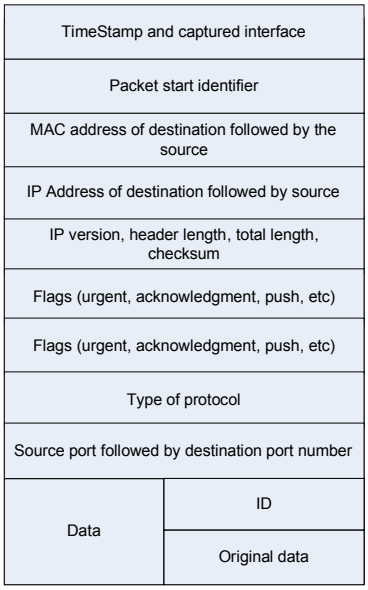

Fig. 3 General structure of a packet being captured

The output from the witnesses is then compared to the one which was saved in the logs. If they are the same, then there is no malfunction; if not the same, there is a chance that the node is not trustworthy. If the output from the logs and replay machine are continuously mismatched, then the node is malfunctioning, and we determine that the node is suffering Byzantine fault and should be then exposed.

B. Prototype Implementation

The experimentation process is as follows. The complete process was first captured using a packet sniffer (Wireshark or Tethereal, for instance). The captured file was saved in Pcap file format and the converted to text format. Then the file was altered (using EagleEye) to the desired format. Once the format is achieved, EagleEye can now start its process. In any network

there are a number of "noise" packets which do not carry any information. They are also used in cases where a continuous connection has to be maintained. These "noise" packets must be eliminated. Note that these noisy packets may have been generated using any protocol (HTTP, UDP, etc.), and thus cannot be eliminated considering the hex value corresponding to the protocol. Another reason why the packets cannot be eliminated using the protocol is that not only the main protocols have a corresponding hex value but also the sub protocols. For instance, TPKT (TCP continuation – generally a noise) is a sub protocol of TCP and has all characteristics of a TCP.

The packets can also not be filtered according to their length because it is well-known that every packet will have its own length (even "noise" packets have their own length depending on how "noisy" they are). One option is to filter the packets using the IP address if it is known. If the packets are filtered this way, they will be strained to a great extent and mostly only those required will be held back. Furthermore, the packets can also be filtered using the port through which the connection was established. This can be done only when the IP address and port number are known and is the most effective method. The second option would be to filter the packets by their ID. As already mentioned, all information is preceded by an ID. This ID is unique to every node. There is no node in the architecture without an ID. The IDs for all nodes are placed in a particular position in the packet. The packets can be filtered with the option that, if the hex value for the ID corresponds to NULL ("00"), then it is obvious that the packet has no data with it and it is highly probable that that particular packet is not causing any trouble. This filtering technique is also efficient. In our initial experiment, the packets were tested by filtering them by both their IP addresses and their ID values. These were done both together and separately in order to study consistency.

Once Eagle Eye finishes the filtration, the packets are saved separately according to their ID (representing individual node) or their IP address and port number. Once they are stored individually, the witnesses are then invoked. The witnesses, since they know the flow of the protocol, start a replay machine which reads the captured packet and replays the performed actions step by step. Once they finish executing, the result of the replay machine is compared to that generated by the node which is now under investigation. If they are the same, then the node is not faulty. It must be noted that the packets are captured on every node. If one node is doubted to be faulty, then the node from which the data was sent and the node to which the data is being sent must be investigated.

V. Evaluation

We developed and tested three application protocols: expression evaluation protocol, echo protocol, and distributed file system.

A. Expression evaluation protocol

This is a simple Java program which takes in two numbers from the client and sends them to the server. The server adds the two numbers and sends the sum to the client, where it is displayed to the user. If there seems to be an error, the witness is called. The witness then takes the logs from both machines and replays the scenario. Since the witness knows the flow of the protocol, it performs the replay and then checks the log files of the server and the client. Wherever inconsistency is found, the particular node is marked as faulty. The witness could be the server, the client, or another external node.

B. Echo protocol

Instead of being a server-client application, this protocol was developed as a peer to peer application. The data sent from a particular node had to be sent along with the destination IP address and port number each time. This IP address and the port number were given as arguments to the protocol. By this method, every node in the network would be the same. Every node is capable of both sending and receiving messages. The received messages were echoed back to the sender. The sender sends only the data; the ID is prefixed to the data by the protocol and then transmitted to the destination. Once the destination receives the data, it adds its own ID and sends back the data. All nodes are set to store the packets which they send and receive. When an investigation must be conducted, the same procedure as above is carried out using EagleEye at the witness/witnesses.

C. Distributed File System (DFS)

This DFS that we built is a simplified prototype for experimentation only. As shown in Fig. 4, there was a super node placed which is connected to all the other nodes. There are a number of data nodes which contain blocks of data. The data has to be created, replaced, or transferred. The action performed depends solely on the user. The data blocks are randomly distributed in the data nodes, and there are also nodes which do not have any data blocks in them (if some balancing strategy is employed, this situation can be eliminated). The super node has a table which listed where each file is present (i.e., which data node has which data block), and is updated accordingly.

The client first requests a file from the super node. The super node looks up which data nodes have the file and sends an order to these data nodes to transfer the file to the client.

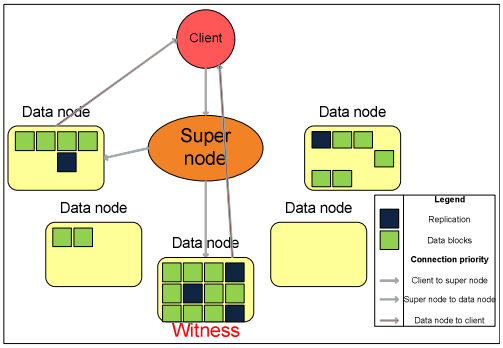

Fig. 4 DFS protocol representation

In case there are replications, the super node sends the request to any data node which is not a witness. The request from the sender to the data node contains information about the client, and the data node sends the file directly to the client. The bottleneck of the super node can be avoided by placing more than one super node in the system and also by reducing the work load of the super node so that it only allocates and does

not transfer data files, i.e., it only acts as a resource allocator and not as a data node. The witnesses have a copy of the original file which is being transferred. In case there are any errors, the witness replays the logs of both the sender and the receiver.

The protocol is replayed, and, if there are any inconsistencies found, the witness marks the corresponding node to be faulty. If no errors are found in replaying the logs of the sender and the receiver, there is a chance that the super node caused the error. In such a case, the witness checks the log at the super node by replaying the sequence at the super node. If there are errors found, the super node is marked as faulty, and, if the super node tends to fail often, then another data node and this super node are made to exchange properties and data. The old super node is now only a data node; if this node again causes failures, then the data in this node, which is not replicated in any other node, is distributed to other data nodes and this node is permanently marked as faulty until the user wishes to reactivate that node.

The super node can only be an allocator in large distributed system environments or it can be a data node and an allocator in smaller distributed systems environments. The witness does not need to be a super node to perform its actions. The witness also does not need to monitor the actions taking place at every node continuously. Hence the witness can be a data node and when requested to witness an event, it can invoke itself and perform the required actions. Alternatively, the witness can also check the credibility of the nodes which it witnesses periodically. The period which it has to monitor can be specified by the user. This period then depends on the current system time, or all the nodes can follow the super nodes' time and all the super nodes can in turn keep themselves synchronized.

D. Experiment results

Only the analysis for the distributed file system is shown, as the other two protocols are fairly simple and can also be analyzed in the same way.

1) Average time for replay

The replay of the protocols is the main procedure involved in checking the credibility of the nodes. The replay time also depends on the number of replications of the file under investigation. If there are more than two replications present (considering the file is more frequently used) and if there seems to be an error in the transaction, the other replication should also be checked before the node is marked as faulty. The replay time plot against file size (in kb) and time (in ms) can be seen in Fig. 5. It can be observed that the time taken to perform the replay increases with the number of replications. This happens only when there are more replications; hence they are directly proportional.

2) Log file size

The size of the log files being stored is another factor which must be taken into account. Depending on this, the user can decide how long he wants to keep the log files and what can be discarded. The plot of the original file size being transferred against the size of the log captured is denoted in Fig. 6. It can be seen that the size of the log file is very close to that of the original file which was transferred. This is because the log contains the entire content of the file which was transferred.

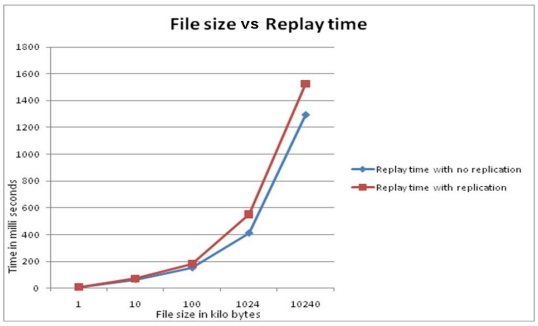

Fig. 5 File size Vs. Replay time

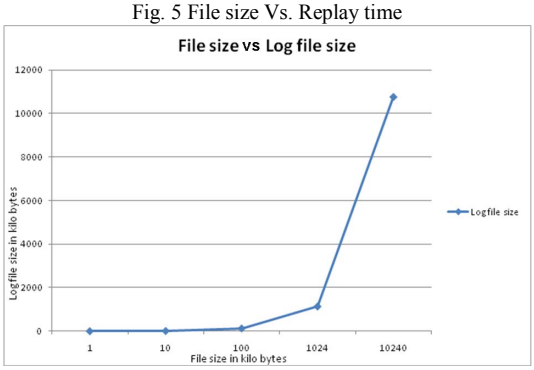

Fig. 6 File size Vs. log file size

The size of the log file is slightly greater than the original file which was sent, as the log file also contains information such as the time stamp, capture interface, protocol type, IP addresses, port numbers, etc. As seen in the graph, the file size and log file size are also directly.

3) Message overhead

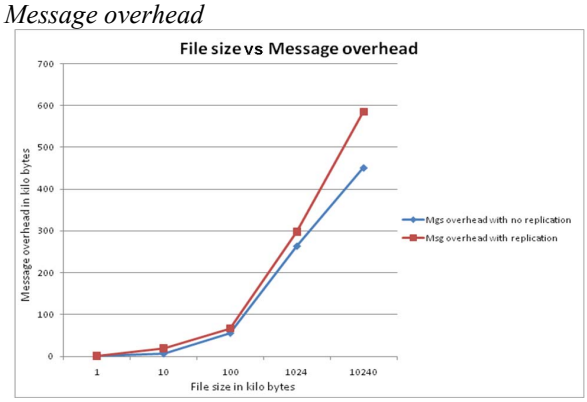

Fig. 7. File size Vs. message overhead

The expense of maintaining certain properties such as connection, etc., is the message overhead. The overhead of the entire distributed system is responsible for the overall performance of the system. Hence analyzing message overhead is essential. One main overhead factor of distributed file systems is the packets generated by protocols in order to maintain connectivity between the nodes. Once any two nodes start to communicate, they also generate "noise" packets. The plot of file size against message overhead can be seen in Fig. 7. Though the overhead also increases with time, it is negligible compared to with the size of the file. This is because the overhead packets must terminate at some point and hence are almost the same at any given point of time. The overhead generated with respect to file size increases proportionally with file size because the traffic generated is greater, which in turn generates greater message overhead.

VI. Conclusion and Future Work

Accountability is an integral part of today's technological world. We have seen accountability in different areas of computer science. We have proposed a logging framework for accountability in distributed systems and implemented it over a number of protocols to compare its efficiency with already existing protocols.

As seen so far, EagleEye is a powerful tool which can integrate accountability into an application minor influence of the host applications. EagleEye also guarantees the detection of all Byzantine faults, as long as the logs are present. However for a larger system, EagleEye would still be efficient, as the operation takes place distributively on all nodes and does not occur in one particular node and thus cause it to become heavy. Also, with the help of Eagle Eye in distributed systems, every packet can be knotted to the machine that sent it. This assurance allows both long-term security and highly spontaneous management of continuing attacks.

EagleEye focuses on minimizing the amount the user has to have to include any code or modify the source code drastically to the application being created, in contrast to PeerReview. Since PeerReview should be included in the application itself, it can indeed perform some operations which EagleEye cannot perform and vice versa. The running time for EagleEye and PeerReview might be more or less the same depending on the application being implemented. Hence EagleEye and PeerReview have their own merits and demerits. It is ultimately the developer's option to choose which mechanism to use, depending on what kind of monitoring he requires and how his application works.

Acknowledgement

This work was supported in part by the US National Science Foundation (NSF) under grants CNS-0737325, CNS-0716211, CCF-0829827, and CNS- 1059265.

References

[1] A. Haeberlen, P. Kouznetsov, and P. Druschel, "PeerReview: Practical accountability for distributed systems," Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, ACM New York, NY, USA, 2007, pp. 175-188.

[2] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel, The Case for Byzantine Fault Detection, Second Workshop on Hot Topics in System Dependability (HotDep '06), Seattle, WA, November 2006

[3] D. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker, "Holding the Internet accountable," ACM HotNets-VI, 2007.

[4] D.G. Andersen, N. Feamster, T. Koponen, D. Moon, and S. Shenker, "Accountable internet protocol (AIP)," Proceedings of the ACM SIGCOMM 2008 conference on Data communication, ACM New York, NY, USA, 2008, pp. 339-350.

[5] Department of Defense. Trusted Computer System Evaluation Criteria. Technical Report 5200.28-STD, Department of Defense, 1985.

[6] Y. Xiao, "Flow-Net Methodology for Accountability in Wireless Networks," IEEE Network, Vol. 23, No. 5, Sept./Oct. 2009, pp. 30-37.

[7] A.R. Yumerefendi and J.S. Chase, "The role of accountability in dependable distributed systems," Proc. of HotDep, 2005.

[8] A.R. Yumerefendi and J.S. Chase, "Trust but verify: accountability for network services," Proc. of 11th workshop on ACM SIGOPS 2004, p. 37.

[9] A.R. Yumerefendi and J.S. Chase, "Strong accountability for network storage," ACM Transactions on Storage, vol. 3, 2007.

[10] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, S. Shenker, and L. EPFL, "Loss and delay accountability for the Internet," IEEE International Conference on Network Protocols, 2007. ICNP 2007, 2007, pp. 194–205.

[11] J. Mirkovic and P. Reiher, "Building accountability into the future Internet," Secure Network Protocols, 2008. NPSec 2008. 4th Workshop on, 2008, pp. 45-51.

[12] Re-ECN: Adding Accountability for Causing Congestion to TCP/IP, Bob Briscoe (BT & UCL), Arnaud Jacquet, Toby Moncaster and Alan Smith (BT), IETF Internet-Draft (Mar 2009).

[13] Re-ECN: The Motivation for Adding Accountability for Causing Congestion to TCP/IP, Bob Briscoe (BT & UCL), Arnaud Jacquet, Toby Moncaster and Alan Smith (BT), IETF Internet-Draft (Mar 2009).

[14] Y. Xiao, "Accountability for Wireless LANs, Ad Hoc Networks, and Wireless Mesh Networks," IEEE Communication Magazine, Vol. 46, No. 4, Apr. 2008, pp. 116-126.

[15] Z. Xiao and Y. Xiao, "P-Accountable Networked Systems," Proceeding of INFOCOM 2010, Work in Progress (WIP) Track.

[16] S. Garfinkel, Practical UNIX and Internet Security. O'Reilly, 2003.

[17] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting Computer System Failures Using Support Vector Machines," in Proceedings WASL, San Diego, 2008.

[18] B. Tierney and D. Gunter, "NetLogger: A toolkit for distributed system performance tuning and debugging," Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management, 2003.

[19] S. Tricaud, "Picviz: finding a needle in a haystack," in Proceedings WASL, San Diego, 2008.

[20] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P Narasimhan, "SALSA: Analyzing Logs as State Machine," in Proceedings WASL, San Diego, 2008.

[21] Z. Xiao, Y. Xiao, and J. Wu, "A Quantitative Study of Accountability in Wireless Multi-hop Networks," Proceedings of 2010 39th International Conference on Parallel Processing (ICPP 2010)

[22] D. Takahashi and Y. Xiao, "Retrieving Knowledge from Auditing Log-Files for Computer and Network Forensics and Accountability," Security and Commun. Net., vol. 1, no. 2, Feb. 2008, pp. 147–60.

[23] http://www.wireshark.org/

[24] K. Meng, Y. Xiao, and S. V. Vrbsky, "Building a Wireless Capturing Tool for WiFi," (Wiley Journal of) Security and Communication Networks, Vol. 2, No. 6, Nov./Dec. 2009, pp. 654 – 668.

[25] H. Wang and X. Jia,"Editorial, " International Journal of Security and Networks, Vol. 5, No.2/3 pp. 77 - 78, 2010,

[26] X. Leng, Y. Lien, K. Mayes, and K. Markantonakis, "An RFID grouping proof protocol exploiting anti-collision algorithm for subgroup dividing," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 79 - 86, 2010.

[27] G. C. Dalton II, K. S. Edge, R. F. Mills, and R. A. Raines, "Analysing security risks in computer and Radio Frequency Identification (RFID) networks using attack and protection trees," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 87 - 95 , 2010.

[28] M. Mahinderjit-Singh and X. Li, "Trust in RFID-enabled Supply-Chain Management," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 96 - 105, 2010.

[29] M. Hutter, T. Plos, and M. Feldhofer, "On the security of RFID devices against implementation attacks," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 106 - 118, 2010.

[30] Y. Imasaki, Y. Zhang, and Y. Ji, " Secure and efficient data transmission in RFID sensor networks," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 119 - 127, 2010.

[31] L. Sun, "Security and privacy on low-cost Radio Frequency Identification systems," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 128 - 134, 2010.

[32] X. Zhang, Q. Gao, and M. K. Saad, "Looking at a class of RFID APs through GNY logic," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 135 - 146, 2010.

[33] S. G. Azevedo and J. J. Ferreira, "Radio frequency identification: a case study of healthcare organisations," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 147 - 155, 2010.

[34] M. Raad, "A ubiquitous mobile telemedicine system for the elderly using RFID," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 156 - 164 , 2010.

[35] M. J. Rodrigues and K. James, "Perceived barriers to the widespread commercial use of Radio Frequency Identification technology," International Journal of Security and Networks, Vol. 5, No.2/3 pp. 165 - 172, 2010.