Statically Defend Network Consumption Against Acker Failure Vulnerability in Storm

Wenjun Qian[1,2], Qingni Shen[1,2(✉)], Yizhe Yang[2,3], Yahui Yang[1,2], and Zhonghai Wu[1,2]

[1] School of Software and Microelectronics, Peking University, Beijing, China
wenjunqian@pku.edu.cn, {qingnishen,yhyang,wuzh}@ss.pku.edu.cn
[2] National Engineering Research Center for Software Engineering, Peking University, Beijing, China
yangyizhe1003@pku.edu.cn
[3] School of Electronics and Computer Engineering, Peking University, Shenzhen, China

Abstract. Storm has been a popular distributed real-time computation system for stream data processing, which currently provides an acker mechanism to enable all topologies to be processed reliably. In this paper, via the source code analysis, we point out that the acker failure and message retransmission result in the consumption of network resources. Even worse, adversary conducts a malicious topology to consume over unconstrained network resources, which seriously affects the average processing time of topology for normal users. Aiming at defending the vulnerability, we design an offline static detection against acker failure in Storm, mainly including the code decompile, the function call relationship and the judgement rules in offline module. Meanwhile, we validate the protection scheme in Storm 0.10.0 cluster, and experimental results show that our mentioned judgement rules can achieve well precision.

Keywords: Stream data · Storm · Acker failure
Message retransmission · Network consumption

1 Introduction

Before the development of the streaming computing platform, many Internet companies, in the face of real-time big data processing problems, usually set up network channels and multiple work nodes by themselves to deal with messages in real time. However, the approach could no longer meet the requirement for data processing, such as no losing data, scaling up the cluster, and manipulating easily. The appearance of Storm [2] solved the above problems, and Storm can deal with real-time massive data which is generated on social platforms. At present, there are many stream data computing systems, such as Storm, S4 [12], Spark Streaming [3], TimeStream [4] and Kafka [1]. S4 and Kafka implement

S. Qing et al. (Eds.): ICICS 2017, LNCS 10631, pp. 661–673, 2018.
https://doi.org/10.1007/978-3-319-89500-0_56

high availability through passive waiting strategy, while Storm, Spark Streaming and TimeStream achieve high availability via upstream backup strategy [6,11]. Compared with other stream processing platforms, Storm is the most widely-used platform in the industry from the aspects of system architecture, application interface, support language and high availability. Storm has advantages in terms of performance, but it raises some security problems.

At present, the academic and industry mainly focus on the security and privacy issues of batch processing platforms, such as information stealing, decision interference and denial of service attack [5,7,16]. And many solutions have been proposed for Hadoop, such as authentication, authorization, differential privacy technology and trusted computing base TCG (Trusted Computing Group), etc. These solutions are complete for secure hardware environment, trusted data processing platform, data encryption and secure computing process [8,13–15]. However, it is inevitable that the complete solutions are not optimal, and need to be improved and tested in practice. Moreover, security issues in big data environments are complex and diverse, and different computing frameworks may require different solutions. Compared with batch processing platforms, stream processing platforms mainly focus on the real-time and reliability. Unfortunately, there are few concern about such security vulnerability issues.

Typically, reliable mechanism in Storm is designed simply and there are security vulnerabilities in reliable mechanism. In this paper, we mainly focus on the security vulnerability of Storm platform. We verify that there are problems of network consumption caused by Acker failure and message retransmission through analyzing source codes and experimental results. Furthermore, we propose a protection scheme to examine malicious code statically and design the experiment in Storm. **Our contributions** can be summarized as follows:

- We show the problems of reliable mechanism in Storm, and the vulnerability of over network consumption, which is caused by Acker failure and message retransmission. If the *XOR* value of message traced by *Acker Bolt* is not zero, which means failing to process message, Spout will resend message and occupy the cluster's resources. Evenly, over network consumption affect processing efficiency for the normal users' topology in real time.
- We run eight Storm benchmarks, and the number of worker and executor are the same in each topology. According to the relative consumption of resources, we classify all benchmarks as memory-dependency topology, CPU-dependency topology and network-dependency topology. Finally, we select the network-dependency topology as the normal user's topology.
- In addition, we also compare the resource consumption dependency with different stream grouping methods for the same topology, and find that global grouping will consume more network resource than other stream grouping methods in the same topology. We design a malicious topology using global grouping to verify the over network consumption.
- We design a protection scheme based on malicious code static detection technology, which decompiles the topology offline, and then analyzes statically the source code according to some judgement rules. If the topology is malicious, it would be killed.

The rest of this paper is organized as follows. We introduce the background in Sect. 2, and discuss problems and challenges of current acker mechanism in Sect. 3. In Sect. 4, we implement and evaluate the effects of malicious attack. We then present the protection scheme and performance of static detection against acker failure in Sect. 5. We conclude the paper in Sect. 6.

2 Background

Our work is related to the fields of Acker in Storm as well as Stream Grouping. In this section, we succinctly introduce the background in these fields.

2.1 Acker in Storm

The reliable mechanism traces each message emitted by Spout relying on *Acker Bolt* in Storm. Tuple tree can be understood as a directed acyclic logic structure, which is formed by source tuples emitted by Spout and new tuples emitted by Bolts. Within timeout limit, *Acker Bolt* tasks conducted the simple *XOR* operation on each *tupleId* in a tuple tree (uniquely identified by *msgId*), and then judged whether the result of *XOR* operation was zero or not. If the *XOR* result was zero, the tuple tree would be processed successfully. Otherwise, the tuple tree was considered to fail. More specifically, the implementation of reliable mechanism is as follows:

- When sending a source tuple, Spout specifies an *msgId* (as the unique *RootId* to identify a tuple tree) and a *tupleId* for the source tuple. Then Spout acknowledges the source tuple and sends $\langle RootId, tupleId \rangle$ to *Acker Bolt*.
- After processing a received tuple successfully, Bolt sends one or more new tuples anchored to the received tuple (uniquely identified by $tupleId_{rec}$), and specifies a random $tupleId_{new}$ for each new tuple. Then it acknowledges the received tuple and sends $\langle RootId, tupleId_{rec} \wedge tupleId_{new} \rangle$ to *Acker Bolt*.
- *Acker Bolt* executes the *XOR* operation on all received acknowledgement messages, which belong to the same *RootId*. If the *XOR* result is zero, *Acker Bolt* acknowledges that the source tuple tagged with *RootId* is processed completely, and then sends *ack* to Spout. Otherwise, after timeout, *Acker Bolt* sends *fail* to Spout and the source tuple is judges as failed.

2.2 Stream Grouping

Stream grouping mechanism in Storm provides eight kinds of message grouping method for topology, and determines how the messages emitted by Spout or Bolt will be received by the downstream Bolt. In this paper, we mainly focus on three kinds of commonly used methods, including shuffle grouping, field grouping and global grouping.

- **ShuffleGrouping.** The shuffle grouping method determines that tuples are assigned between Spout and Bolt randomly. And the random assignment result causes the same number of tuples to be allocated on each Bolt.
- **FieldGrouping.** In a topology, the field grouping method can specify that all tuples emitted by the upstream Spout or Bolt, are grouped by one or more fields, and then distributed to multiple downstream Bolt tasks. That is to say, each downstream Bolt receives tuples in same group.
- **GlobalGrouping.** In a topology, the global grouping method assigns all tuples emitted by the upstream Spout and Bolt to one downstream Bolt task, specifically, the Bolt task with the smallest *taskId*.

3 Problems and Challenges

In this section, we point out the problems and challenges of current reliable mechanism in Storm, and present an attack model with the existed vulnerability.

3.1 Vulnerability Analysis

Through analyzing the source code of reliable mechanism, we detect that both reliable and unreliable topologies can be run in Storm. Besides, Storm developer provides programmer with flexible API. However, it is vital to deal with some business scenarios with message consistency, such as bank deposit, transformation and remittance business.

When programmer design their topology, Storm provides Spout and Bolt components with some basic interfaces and abstract classes, including *Icomponent*, *ISpout*, *IBolt*, *IRichSpout*, *IRichBolt*, *IBasicBolt* and some other kinds of basic interfaces, as well as *BaseComponent*, *BaseRichSpout*, *BaseRichBolt* and *BaseBasicBolt* and some other kinds of basic abstract classes. *BaseBasicBolt* class implements *IBasicBolt* interface, which acknowledges the received tuple automatically. Programmer requires for inheriting *BaseBasicBolt* abstract class when designing reliable Bolt, and *BaseRichBolt* abstract class when designing unreliable Bolt. Through analyzing the source code of acker mechanism, we point out some existed vulnerabilities in Storm as follows:

- **Q1:** *In a reliable topology, it is necessary to execute the ack or fail operation for each processed message, which enables the reliability of message in Storm. Correspondingly, tracing message consumes memory resources.*

  At the beginning of designing Storm, in order to give users a better experience, Storm provides reliable and unreliable interface and abstract class. However, it indicates that user can render Bolt tasks more reliable by means of calling *ack()* function in unreliable abstract class. Although this way is more flexible, it cannot avoid the danger of attackers.
- **Q2:** *If some Bolts (called unreliable Bolt) don't execute the ack or fail operation for its received messages, it will result in the value of ack in Acker Bolt is*

not equal to zero in the extended time. Spout cannot trigger the ack() function to evacuate the messages, so it will result in resources waste.

When users design a reliable Bolt by inheriting *BasicRichBolt* abstract classes, every Bolt task will process the received tuple, emit new tuple, and call the *ack()* function after completing tuple anchoring. Although the tuple tree has been processed completely actually within timeout limit, the tuple tree will always be traced and cannot be released. Over timeout, Bolt will be not able to send $tupleId_{rec}$ and $tupleId_{new}$ to *Acker Bolt*, and the tuple tree will be judged as failure.

- **Q3:** *When reaching extended time, Acker Bolt send fail to Spout about source tuple. If user triggers fail() function and implement resend tuple function from Spout, this will lead to source tuple resending over and over again, which consume CPU and network resources.*

In practice, it is not been implemented to re-transmit tuple in the *fail()* function of Spout component by system developer. If user implements fail() function, Spout will call it to re-transmit a source tuple. Similarly, the source tuple will also be judged as failure in the end. There are two ways of designing reliable Bolt as follows:

- Reliable Bolt can be implemented by calling *IRichBolt* interface. Firstly, it needs anchor each new tuple to the received tuple while sending new tuples, and then call *ack()* method to acknowledge the received tuple. If no anchoring or calling, the new tuple emitted by Bolt will not be traced.
- Reliable Bolt can be implemented by inheriting *BaseBasicBolt* abstract class, which automatically implement the emitting, anchoring and acknowledgement operations for each new tuple in *executor()* method.

3.2 Attack Model

Attack Target. If *Acker Bolt* do not receive the acknowledgment message from Bolt, Acker failure and message retransmission will result in the vulnerability of over network consumption. Once the vulnerability is used by bad attacker, who faked as a legal user in cluster, and submitted a malicious topology, it not only consumes network resource in cluster, but also affects the efficiency of topology for a normal user.

The attack model of malicious topology is designed as Fig. 1. The malicious topology implements a reliable Spout, a reliable Bolt_1 and an unreliable Bolt_2 by inheriting *IRichBolt*. However, Bolt_2 tasks are anchored to the received tuples and not acknowledged. The adversary runs a malicious topology as follows:

1. By running test.py script to write contents into */tmp/fluem/test.log*, Flume monitors the log file, and transfers the input data to test topic in Kafka.
2. Spout assigns *msgId* to each source tuple, and sends a key-value pair $\langle RootId, tupleId \rangle$ to *Acker Bolt* after acknowledging the source tuple.

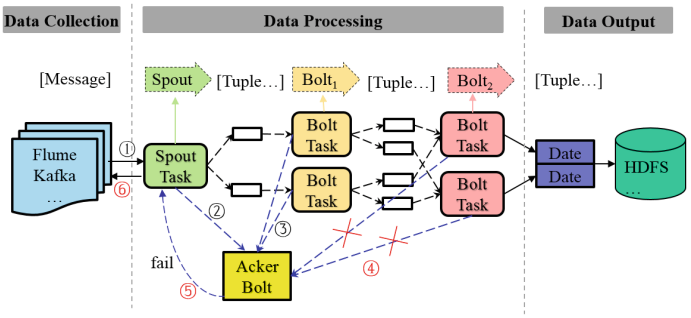


Fig. 1. The attack model of malicious topology

3. Bolt_1 emits new tuples and anchors these tuples to the received tuple. Then, Bolt calls *ack()* function to acknowledge all received tuples, and sends $tupleId_{rec}$ and $tupleId_{new}$ to *Acker Bolt*.
4. Bolt_2 pulls every tuple from Bolt_1 using global grouping method, and does not call *ack()* function to acknowledge all received tuples.
5. After timeout limit, it is not zero that the *XOR* value of *RootId* in *Acker Bolt*. The source tuple is processed failed.
6. Spout will call *fail()* function itself, and re-transmit the failed message from Kafka message queue. By repeating the previous steps, the failed message is still processed failed all the time.

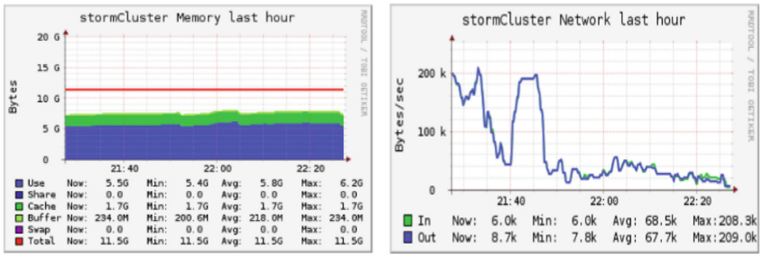
4 Experimental Evaluation

Through the Ganglia monitoring tool, we view the resource occupancy in cluster. Through the Storm UI, we view the malicious topology and normal user topology operation, and compare the average processing time of normal user's topology whether running malicious programs or not. We design three kinds of topologies, including general topology (simplified as GT_1), malicious topology (simplified as MT_2) and malicious topology (simplified as MT_3). GT_1 consists of a reliable Spout, a reliable Bolt_1 and a reliable Bolt_2, MT_2 consists of a reliable Spout, a reliable Bolt_1 and an unreliable Bolt_2, and MT_3 consists of a reliable Spout, a reliable Bolt_1 and an unreliable Bolt_2. The difference between GT_1 and MT_2 is that whether Bolt_2 is a reliable component, and the difference between MT_2 and MT_3 is that stream grouping method is global grouping in MT_3.

Network Consumption. Comparing the network consumption between GT_1 and malicious topologies, we respectively submitted three topologies into Storm 0.10.0 cluster. Figure 2 shows the total network and memory consumption when only running GT_1 in Storm cluster. And Fig. 3 shows the network consumption respectively when only running a malicious topology, namely MT_2 or MT_3.

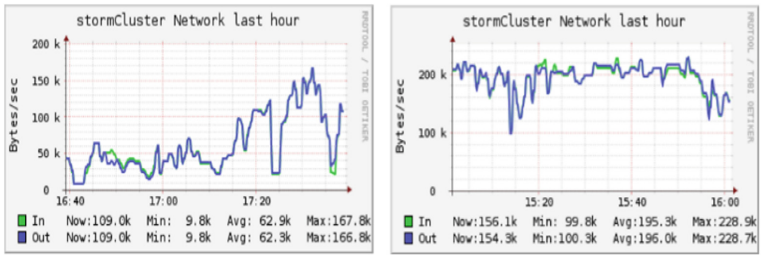
There are significant differences between GT_1 and malicious topologies. When GT_1 running within one hour in Storm cluster, the input network consumption

of GT_1 is average of 68.5k, and output network consumption is average of 67.7k. However, it is significant for two malicious topology on the increased network consumption. Compared with MT_2, especially for MT_3 using global grouping, the input network consumption is average of 195.3k, the output network consumption is average of 196.0k, which has more than 3x in network consumption. From MT_2 and MT_3, we can see that it is relatively high in network consumption for the malicious topology using global grouping topology. From GT_1 and MT_2, we can see that it is an obviously growth to consume network resources for the MT_2, and experimental results show that the growth is nearly 10x. However, it is roughly the same memory consumption for GT_1 and two malicious topologies. We find that the memory resource consumption value depends on the number of worker that is set in a topology by programmer.



(a) Memory consumption (b) Network consumption

Fig. 2. The network and memory consumption when only running GT_1 in Storm



(a) Running MT_2 (b) Running MT_3

Fig. 3. The network consumption when running MT_2 or MT_3 respectively in Storm

Resource Dependent Classification. There are eight benchmarks in Storm, and we only test six benchmarks in Storm cluster, excluding two benchmarks only run in local mode. We run each benchmark separately for one hour in Storm cluster. In order to eliminate the interference caused by the number of worker and executor in different topologies, we set the number of worker to three,

and the number of executor to twelve for each benchmark. Besides, six benchmarks are reliable topologies. By counting and comparing the consumption of memory, CPU, and network resources for each benchmark at runtime, as shown in Table 1, we find that *Slidwindow* consumes the highest amount of network resources, *Multipletogger* consumes the lowest amount of network resources. The in-network consumption of *Slidwindow* is 1.7x than the in-network overhead of the *Multiplelogger*. In the subsequent malicious topology attacks, we will conduct a malicious topology to impact the network resource-dependent topology, namely *Slidwindow*.

Table 1. Network, CPU and memory consumption statistics about examples in Storm

	Cluster				Supervisor1				Supervisor2				Supervisor3			
	Network		CPU	Memory	Network		CPU	Memory	Network		CPU	Memory	Network		CPU	Memory
	In	Out			In	Out			In	Out			In	Out		
Exclamation	35.4k	17.4k	1.5%	6.2G	9.9k	3.9k	2.1%	1.8G	8.2k	3.5k	1.5%	1.2G	10.3k	6.3k	1.8%	1.5G
Multiplelogger	31.6k	17.2k	1.6%	6.2G	9.2k	3.5k	2.2%	1.8G	7.6k	3.5k	1.5%	1.2G	9.8k	6.6k	1.8%	1.5G
Resourceaware	35.5k	18.0k	1.5%	6.2G	10.4k	4.0k	2.1%	1.7G	8.4k	3.5k	1.5%	1.2G	10.9k	6.4k	1.7%	1.4G
Slidetuple	49.2k	31.1k	2.1%	6.4G	14.7k	8.4k	2.8%	1.8G	12.6k	7.7k	2.3%	1.2G	15.0k	11.1k	2.5%	1.5G
Slidwindow	54.4k	36.0k	2.3%	6.4G	16.6k	10.6k	3.1%	1.8G	14.1k	9.1k	2.6%	1.3G	17.0k	12.8k	2.8%	1.5G
Wordcount	43.4k	30.0k	2.3%	6.1G	13.4k	9.2k	3.0%	1.7G	10.6k	7.1k	2.2%	1.1G	14.2k	10.0k	3.0%	1.6G

Average Processing Time. We designs an *experiment group* and a *control group* to verify the influence of malicious topology on the average processing time for normal topology. In control group, a normal user's topology (*WordCount*) and a general user's topology (GT_1) are run in Storm at the same time. Whereas in experimental group, *WordCount* and MT_3 are run.

The average running time of topology for normal under experiment group and that under control group are shown as Fig. 4. The experimental results show that the average processing time of *WordCount* topology is about 1.49 ms

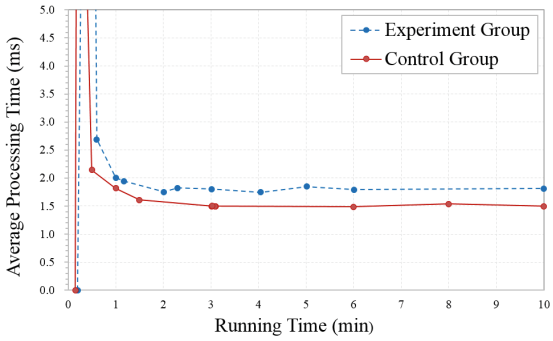


Fig. 4. The average processing time of normal topology in Storm

under normal circumstances. In experiment group, the average processing time of *WordCount* topology for normal users is about 1.79 ms, and the processing speed is slowed by 19.4%.

5 Protection Against Acker Failure

In this section, we first present an overview of static detection against acker failure in Storm, mainly including decompile, function call relationship and judgement rules in offline module, and then describe design and implementation of the judgement rules.

5.1 Design of Static Detection

In order to solve the security vulnerability of Acker failure, we propose a protection scheme from three angles as follows:

- Improving Acker code through designing a secure Spout or Bolt interface.
- Detecting Spout and Bolt in a reliable topology by static code detection.
- Detecting all abnormal behaviors of resources consumption by ganglia tool.

In the previous analysis of interface, we can see that Storm provides a secure and reliable anchoring mechanism for Spout and Bolt, and provides a reliable *IBasicBolt* interface and *BaseBasicBolt* abstract class for Bolt. Therefore, we does not need to design a secure Spout or Bolt interface. In addition, ganglia monitors the entire Storm cluster from the perspective of the application layer. Note that, if Ganglia detects the excessive consumption of network resources, maybe the abnormal behavior is not caused by acker failure and message retransmission in malicious procedures.

Based on the above analysis, we design and present an offline static detection against failure in Storm as shown in Fig. 5. If the detection result of a topology is legal, the topology will be processed in real time. The offline static detection against acker failure in Storm works as follows:

1. Decompile process makes executable topology code to be java source code, and needs that it is optimized to restore the source code form.
2. The class function call relationship graph can be achieved by using ModelGoon plugin tool. Specifically, this step needs present the implementation interface and the call method.
3. The source code will be analyzed according to the specified judgement rules. If a topology is designed reliably at first, only if all Spout and Bolt have invoked the ack method, the topology is legal. Otherwise, it is judged as a malicious topology. If a topology is unreliable, it is directly determined as a legal topology.

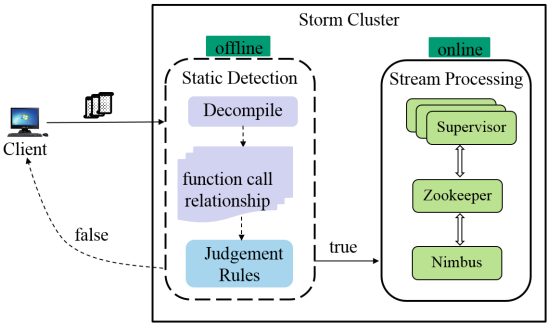


Fig. 5. The overview of offline static detection against acker failure in Storm

5.2 Judgement Rules

In the offline environment, the source code of a topology is analyzed based on the judgement rules. If the result is true, the topology will be submitted and processed in real time. Otherwise, the topology is rejected. Specifically, judgement rules are as follows:

- ***Rule-1*:** In a topology, if the developer calls Spout's *nextTuple()* method to send a source tuple that is not specified with a unique *msgId*, the topology is determined as an unreliable topology. Storm will not start the *Acker Bolt* component to trace the source tuple, and the return value is true. Otherwise, the topology is a reliable topology and needs to be judged continually by *Rule-2* and *Rule-3*.
- ***Rule-2*:** If the developer inherits the *BaseBasicBolt* class that implements the *IBasicBolt* interface, this indicates that the Bolt's *execute()* method will automatically implement the anchoring and acknowledgement operations, and the return value is true.
- ***Rule-3*:** If the developer inherits the *BaseRichBolt* class that implements the *IRichBolt* interface, and emits a new tuple that is anchored to the parent tuple at the same time, the parent tuple does not explicitly call the *ack()* method. Then the Bolt is determined as an unreliable Bolt, and the return value is false. Otherwise, the Bolt is a reliable Bolt, and the return value is true.

In particular, if Bolt is implemented by an unreliable interface in a topology, regardless that whether new tuple is anchored to the parent tuple, the topology is judged as a malicious topology eventually as long as the call of *ack()* method is not explicitly called.

5.3 Performance

After decompiling a topology, the executable .class.jar package is decompiled into a java source file. Then, through the ModelGoon plug-in, we draw the class

function relationship graph. In this section, we just test the decompile result and achieve a usable relationship graph. We conduct the code detection offline, which does not effect the real-time performance in Storm.

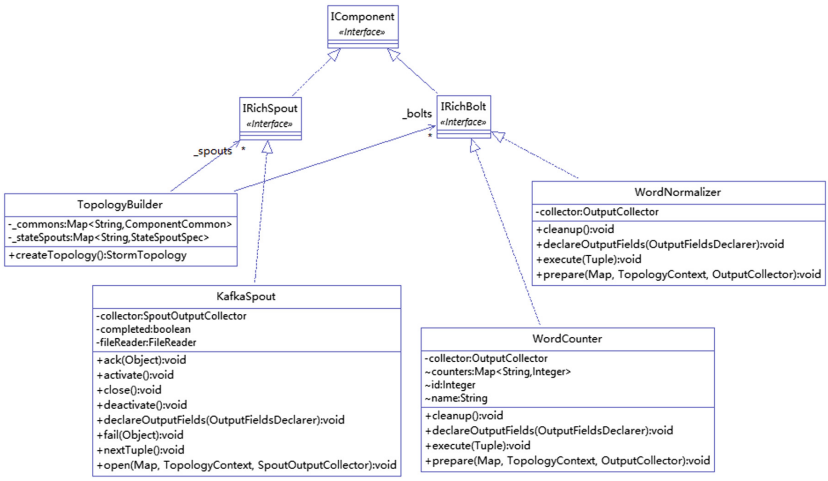


Fig. 6. The class relationship of decompiled topology

We decompile a topology submitted from client, and get a class relationship by Eclipse ModelGoon plug-in as shown in Fig. 6. It consists of a *KafkaSpout* component and two Bolt components. *KafkaSpout* implements *IRichSpout* interface, and WordNormalizer Bolt and WordCounter Bolt implement *IRichBolt* interface. Further, through the call hierarchy view function call, it was found that the *ack()* method was invoked only when the *execute()* method was called in the WordNormalizer class implementation, the message was not statically linked in the WordCounter class, and the *ack()* method was not explicitly called. The topology submitted by user is judged as a malicious topology, and can not be submitted into Storm cluster.

6 Conclusion

In this paper, we mainly study two mechanisms in Storm, including reliable mechanism and stream grouping mechanism. Meanwhile, we analyze and point out the security vulnerability of current reliable mechanism, in which unreliable Bolt enables the source tuple to fail, retransmit it continually and bring with over network consumption. Via experimental results and source code analysis, we find that malicious topology influent the normal user's topology, over 19.4% at the average processing time. Motivated by the vulnerability of acker failure, we presents a protection scheme using malicious code static detection technology.

Our work still need to study and improve a better detection against acker failure. Firstly, we only consider the static detection. Dromard et al. [9] and Wang et al. [17] proposed different methods of anomaly detection respectively, which are worthy of reference to improve the current static detection scheme. Secondly, during the detection of malicious programs, the selected feature is anchoring and acknowledgement. Furthermore, our next work is to select different features, and use the method of SVM (Support Vector Machine) to detect anomaly like [10].

Acknowledgments. This work is supported by the National Natural Science Foundation of China under Grant No. 61672062, 61232005, and the National High Technology Research and Development Program ("863" Program) of China under Grant No. 2015AA016009. Thanks to Lingyun Guo and Liming Zheng for the support of experimental data collection and stream grouping analysis.

References

1. Apache kafka. http://kafka.apache.org
2. Apache storm. http://storm.apache.org
3. Spark streaming. http://spark.apache.org/streaming
4. TimeStream. https://github.com/TimeStream/timestream
5. Alguliyev, R., Imamverdiyev, Y.: Big data: big promises for information security. In: IEEE International Conference on Application of Information and Communication Technologies, pp. 1–4 (2014)
6. Aritsugi, M., Nagano, K.: Recovery processing for high availability stream processing systems in local area networks. In: TENCON 2010–2010 IEEE Region 10 Conference, pp. 1036–1041 (2010)
7. Bertino, E., Ferrari, E.: Big data security and privacy. In: IEEE International Congress on Big Data, pp. 757–761 (2015)
8. Dinh, T.T.A., Saxena, P., Chang, E.C., Ooi, B.C., Zhang, C.: M2R: enabling stronger privacy in MapReduce computation (2015)
9. Dromard, J., Roudiere, G., Owezarski, P.: Online and scalable unsupervised network anomaly detection method. IEEE Trans. Netw. Serv. Manag. **PP**(99), 1 (2017)
10. Khaokaew, Y., Anusas-Amornkul, T.: A performance comparison of feature selection techniques with SVM for network anomaly detection. In: International Symposium on Computational and Business Intelligence, pp. 85–89 (2016)
11. Nagano, K., Itokawa, T., Kitasuka, T., Aritsugi, M.: Exploitation of backup nodes for reducing recovery cost in high availability stream processing systems. In: Fourteenth International Database Engineering & Applications Symposium, pp. 61–63 (2010)
12. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: distributed stream computing platform. In: IEEE International Conference on Data Mining Workshops, pp. 170–177 (2011)
13. Ohrimenko, O., Costa, M., Fournet, C., Gkantsidis, C., Kohlweiss, M., Sharma, D.: Observing and preventing leakage in MapReduce (2015)
14. Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: security and privacy for MapReduce. In: Usenix Symposium on Networked Systems Design and Implementation, NSDI 2010, 28–30 April 2010, San Jose, CA, USA, pp. 297–312 (2010)

15. Sweeney, L.: k-anonymity: a model for protecting privacy. Int. J. Uncertainty Fuzziness Knowl. Based Syst. **10**(05), 557–570 (2002)
16. Takabi, H., Joshi, J.B.D., Ahn, G.J.: Security and privacy challenges in cloud computing environments. IEEE Secur. Priv. **8**(6), 24–31 (2010)
17. Wang, Z., Yang, J., Zhang, H., Li, C., Zhang, S., Wang, H.: Towards online anomaly detection by combining multiple detection methods and storm. In: Network Operations and Management Symposium, pp. 804–807 (2016)