基于 MapReduce 的 HDFS 数据窃取随机检测算法

高元照[1,2]，李炳龙[1]，陈性元[1,2]
(1. 信息工程大学三院，河南 郑州 450001；2. 密码科学技术国家重点实验室，北京 100094)

摘　要：为了解决分布式云计算存储的数据窃取检测中，出现数据量大、内部窃取难以检测的问题，以 hadoop 分布式文件系统（HDFS, hadoop distributed file system）为检测对象，提出了一种基于 MapReduce 的数据窃取随机检测算法。分析 HDFS 文件夹复制产生的 MAC 时间戳特性，确立复制行为的检测与度量方法，确保能够检测包括内部窃取的所有窃取模式。设计适合于 MapReduce 任意的任务划分，同时记录 HDFS 层次关系的输入数据集，实现海量时间戳数据的高效分析。实验结果表明，该算法能够通过分段检测策略很好地控制漏检率和误检文件夹数量，并且具有较高的执行效率和良好的可扩展性。

关键词：随机检测算法；HDFS；MapReduce；MAC 时间戳；云计算存储
中图分类号：TP311.1
文献标识码：A
doi: 10.11959/j.issn.1000−436x.2018222

Stochastic algorithm for HDFS data theft detection based on MapReduce

GAO Yuanzhao[1,2], LI Binglong[1], CHEN Xingyuan[1,2]
1. Third Academy, Information Engineering University, Zhengzhou 450001, China
2. State Key Laboratory of Cryptology, Beijing 100094, China

Abstract: To address the problems of big data efficient analysis and insider theft detection in the data theft detection of distributed cloud computing storage, taking HDFS (hadoop distributed file system) as a case study, a stochastic algorithm for HDFS data theft detection based on MapReduce was proposed. By analyzing the MAC timestamp features of HDFS generated by folder replication, the replication behavior’s detection and measurement method was established to detect all data theft modes including insider theft. The data set which is suitable for MapReduce task partition and maintains the HDFS hierarchy was designed to achieve efficient analysis of large-volume timestamps. The experimental results show that the missed rate and the number of mislabeled folders could be kept at a low level by adopting segment detection strategy. The algorithm was proved to be efficient and had good scalability under the MapReduce framework.

Key words: stochastic detection algorithm, HDFS, MapReduce, MAC timestamp, cloud computing storage

1　引言

云计算巨大的优越性使其在全球范围内高速发展[1-2]。其中，“存储即服务（StaaS, storage as a service）”的云计算服务模式是当前被广泛认可的大数据存储方法[3]，许多云计算服务提供商如 Google、Amazon 和 Microsoft 都提供了云存储服务[4-5]。

云计算存储快速发展的同时，新的安全威胁随

收稿日期：2017−09−27；**修回日期**：2018−07−03
基金项目：国家高科技研究发展计划（“863”计划）基金资助项目（No.2015AA016006）；国家自然科学基金资助项目（No.61702550）
Foundation Items: The National High Technology Research and Development Program of China (863 Program) (No.2015AA016006), The National Natural Science Foundation of China (No.61702550)

之而来。云安全联盟2013年和2016年发布的云计算顶级威胁报告提出数据泄露是云安全面临的最大威胁[6-7]。《中国云计算安全政策与法律蓝皮书（2016）》也披露，近年来云平台的大规模数据泄露事件不绝于耳[8]。数据窃取是造成数据泄露的主要原因之一，并且相比于外部窃取，内部人员往往具有数据访问特权，当前的加密或访问控制等机制不能完全防止内部窃取[9-11]，因而给云计算和用户造成的危害更大[12-13]，并且难以检测。因此，本文主要研究云存储数据的内部窃取检测方法。

现有的内部窃取检测方法主要分为2类。

1) 基于用户或系统行为的异常进行检测。Stolfo等[9]通过识别系统中入侵用户不同于合法用户的异常数据访问行为模式检测内部数据窃取。Nikolai等[11]依据系统中异常的活动用户数量和传输数据量，检测内部人员对目标机的登录和数据窃取。Pitropakis等[14]基于Smith&Waterman算法识别虚拟机的异常行为，进而检测针对虚拟机的内部攻击。然而，云存储多采用分布式文件系统（DFS，distributed file system）作为主要的文件系统[15]，一个完整文件分散存储在多个物理或虚拟节点中，从单个节点窃取数据是不可行的；另一方面，上述方法能够检测的攻击模式需要内部人员登录到目标系统，然而在DFS中内部管理人员能够通过控制节点直接访问存储节点的数据而不需要登录系统。因此，上述方法针对的窃取模式以及对应检测方法不适用于DFS。

2) 基于文件系统元数据，通过检测文件的批量复制遗留的时间戳痕迹，判断数据窃取的发生。这类方法从根本上反映文件系统活动，能够检测包括内部窃取和外部窃取的所有窃取模式。Grier[16]首次提出了通过识别文件时间戳在文件批量复制中的显著特征，检测NTFS数据窃取的方法。但其他文件操作也可能批量更新文件时间戳，造成误报，因而Patel等[17]采用人工神经网络、分类和回归树等方法，对文件复制之外的操作产生的误报进行过滤，提升数据窃取检测的准确率。上述方法面向单磁盘文件系统，而DFS中文件操作的管理和记录方式与单磁盘文件系统差异较大，此类方法无法直接应用，并且云环境下用户量和数据量大，上述方法没有考虑海量时间戳数据的高效分析问题。

为解决上述问题，本文针对HDFS中数据的批量窃取行为，提出一种基于MapReduce的数据窃取随机检测算法。主要工作如下所示。

1) 分析HDFS文件夹复制产生的时间戳特性，建立HDFS行为随机模型，提出HDFS文件夹复制行为的检测与度量方法，即时检测数据窃取活动，并确定被窃取用户的信息。

2) 设计适合于MapReduce分布式处理、同时记录HDFS层次关系的数据集和以文件为数据单元、以文件夹为检测单元的算法执行过程，实现对海量时间戳数据的高效、正确分析。

3) 搭建完全分布式的hadoop框架，测试检测阈值对算法准确性的影响以及数据量和计算节点数量对算法效率的影响，检验了算法的准确率、执行效率和可扩展性。

2 文件系统行为随机模型

Grier提出一种基于文件系统行为随机模型（the stochastic model of file system behavior）的数据窃取检测方法[16]。Grier发现，当文件系统发生文件的批量复制时，被复制文件夹的MAC时间戳统计特性与常规文件访问产生的MAC时间戳变化特性明显不同，并分别用应急模式和常规模式表示。在Grier提出的模型中，MAC时间戳代表文件（或文件夹）的最近修改时间（mtime）、最近访问时间（atime）和创建时间（ctime）。

在常规模式下，文件访问通常是选择性的，只有被访问文件（或文件夹）的*atime*发生改变。在应急模式下，文件夹的复制表现为一种一致性行为，该文件夹内所有子文件夹和文件都被复制，引起文件*atime*的全部更新（或只更新所有子文件夹的*atime*，不同文件系统的*atime*更新规则不同）。

不考虑时间戳的恶意篡改，MAC时间戳总是单调递增的。如果一个文件夹被复制，即使在几个月以后，它的MAC时间戳仍将表现出以下特性。

1) 被复制的文件夹和所有子文件夹的*atime*都不小于复制发生时间。

2) 大量上述文件夹的*atime*近似等于复制发生时间。

3) 在Windows下，文件夹复制不更新文件的*atime*。很多文件的*atime*通常小于被复制文件夹的*atime*。

基于上述特性，Grier提出在NTFS文件系统中，对于一个文件夹和某一特定时刻，若任何子文件夹的 *atime* 都不小于该时刻，并且一定比例的子文件夹的 *atime* 等于该时刻，则称创建了一个“截止簇（cutoff cluster）”。*atime* 等于该时刻的子文件夹属于截止簇。Grier 对一个文件夹中属于截止簇的子文件夹数量及其占总文件夹数的比例进行量化分析，以判断该文件夹是否被复制过。

基于文件系统行为随机模型提出的数据窃取检测算法能够适用于 NTFS、Ext4 等多种单磁盘文件系统和 HDFS、GFS、Gluster 等通过 MAC 时间戳记录文件系统活动的 DFS。但不同文件系统的 MAC 时间戳含义和更新规则不同，量化分析的过程不是通用的。并且，除文件夹复制外，文件搜索和“ls -r”等操作也可能造成文件（或文件夹）*atime* 的批量更新，对检测准确性造成影响。因此，量化分析过程和数据窃取检测方法需要依据特定文件系统的 MAC 时间戳特性进行设计。

3　基于 MapReduce 的数据窃取检测算法

3.1　HDFS 时间戳特性量化分析

3.1.1　HDFS 时间戳特性分析

HDFS 采用一种主从式的架构，一个 HDFS 集群包含一个 NameNode、一个二级 NameNode 和众多的 DataNode[18]。NameNode 管理 HDFS 命名空间，集中存储 HDFS 的元数据。DataNode 用于存储实际的数据。二级 NameNode 用于执行周期性的检查点机制。HDFS 元数据文件有两种形式：FsImage 和 EditLog。FsImage 维护着完整的 HDFS 元数据映像，记录着文件的 MAC 时间戳。EditLog 是 HDFS 的事务日志，记录了每个 MAC 时间戳的变化。

HDFS 的检查点机制是在 hadoop 启动时或每经过固定周期，HDFS 将最近更新的 FsImage 与之后所记录的事务进行合并，创建一个新的 FsImage，并删除过期的 FsImage。虽然 FsImage 的数据来源于 EditLog，但由于 EditLog 只能记录两检查点间 HDFS 的变化情况，它无法反映 HDFS 中时间戳未发生变化的文件，不能体现某文件夹下文件的整体状态。此外，日志中的事务按照时间顺序记录，密集的时间戳变化可能由多用户同时访问 HDFS 引起，不能作为文件复制的依据。因此，本文以 FsImage 作为 MAC 时间戳的主要来源，进行数据窃取的检测。

与 NTFS、Ext4 等不同，HDFS 只记录文件夹的 *mtime* 和 *ctime* 以及文件的 *mtime* 和 *atime*。由于在 NTFS 中对文件夹复制不更新文件的 *atime*[16]，Grier 基于文件夹的 *atime* 和 *ctime* 对截止簇进行量化分析。在 HDFS 中，经过实验验证，文件夹的复制会引起该文件夹下所有文件 *atime* 的更新。因此，本文通过文件的时间戳进行截止簇的量化分析。

对于文件的 *ctime*，分析 EditLog 发现，HDFS 在创建文件时，会记录两次时间戳的变化。第一次是创建空文件时，此时 *mtime* 的值等于 *atime*。第二次是当文件创建完后关闭时，*mtime* 发生更新而 *atime* 保持不变。因此，本文假设文件的 *ctime* 等于文件创建时的 *atime*。

3.1.2　截止簇量化分析

首先对文件夹 f 做如下定义

$$D(f) = \{x \mid x \in f\} \tag{1}$$

$D(f)$ 表示 f 中所有文件的集合，x 表示单个文件。对于给定时刻 t，本文将 $D(f)$ 分割成 4 个不相交的子集。

$$Db_t(f) = \{x \mid x \in D(f) \wedge (A(x) < t) \wedge (C(x) < t)\} \tag{2}$$

$$De_t(f) = \{x \mid x \in D(f) \wedge (t \leqslant A(x) \leqslant t+\varepsilon) \wedge (C(x) < t)\} \tag{3}$$

$$Da_t(f) = \{x \mid x \in D(f) \wedge (A(x) > t+\varepsilon) \wedge (C(x) < t)\} \tag{4}$$

$$Di_t(f) = \{x \mid x \in D(f) \wedge (C(x) \geqslant t)\} \tag{5}$$

其中，$A(x)$ 表示 x 的 *atime*，$C(x)$ 表示 x 的 *ctime*。ε 稍大于复制所需的预计时间。基于文件系统行为随机模型的理论，$(t \leqslant A(x) \leqslant t+\varepsilon) \wedge (C(x) < t)$ 的文件属于截止簇，用 $De_t(f)$ 表示。$Db_t(f)$ 表示 f 中 *atime* 和 *ctime* 均小于 t 的文件的集合，$Da_t(f)$ 表示在 t 之前被创建在 $t+\varepsilon$ 之后被访问过的文件的集合，$Di_t(f)$ 表示在 t 之后才被创建的文件的集合。

基于上述 4 个子集，本文定义了截止簇的度量值 $C_t(f)$ 和 $M_t(f)$ 为

$$C_t(f) = \begin{cases} 0, & |Db_t(f)| > 0 \\ \dfrac{|De_t(f)|}{|De_t(f)| + |Da_t(f)|}, & \text{其他} \end{cases} \tag{6}$$

$$M_t(f)=\begin{cases}\infty, & |Db_t(f)|>0\\ |De_t(f)|+|Da_t(f)|, & 其他\end{cases} \tag{7}$$

其中，$C_t(f)$ 表示截止簇的相对大小和 f 在时刻 t 被复制的可能性，取值范围在[0,1]之间。若 $Db_t(f)$ 不为空集，表明 f 中存在 t 之前被访问的文件，f 不可能在时刻 t 被复制过，所以 $C_t(f)$ 为 0。$M_t(f)$ 表示 $C_t(f)$ 的可信度，$Db_t(f)$ 非空时可信度最高，用“∞”表示。其他情况下，$De_t(f)$ 和 $Da_t(f)$ 中文件的总量越多，用户通过常规模式更新所有文件 *atime* 的概率越低，f 在时刻 t 被复制过的可信度就越高。

结合 HDFS 检查点机制，设 FsImage 最近更新时刻为 t_u，令 $t=t_u-\varepsilon$，能够在每次 FsImage 更新后，对文件夹复制行为进行定期检测，及时发现数据窃取。

3.2 基于 MapReduce 的检测算法实现方法

3.2.1 数据预处理

由于 FsImage 不记录文件的 *ctime*，本文首先基于“文件 *ctime* 等于其初始创建时的 *atime*”的假设，生成文件的 *ctime*。HDFS 两个检查点间的文件系统变化存储在单独的 EditLog 中。在 FsImage 中，文件（或文件夹）以“*inode*”表示，每个 *inode* 具有唯一且单调递增的编号。FsImage 记录了本文件内的最大 *inode* 编号。对比最近更新的 FsImage 和前一个检查点的 FsImage，能够得出两个检查点间的新增文件。在最近的已完成 EditLog 中检索这些文件，能够得出它们的 *ctime*。

另一方面，MapReduce 将数据集划分成相互独立的子集（或分片）进行并行处理。为确保作业在数据集上是可以任意划分的，数据集的各数据项之间通常没有内在关联。而 HDFS 是一个多层次的树形结构，数据窃取的检测也正依赖于文件间的父子关系。由于文件的元数据并不包含其父文件夹的信息，若按照 MapReduce 默认方式分割任务很可能破坏目录树的完整性，导致分析检测的错误；若按照目录树的分支结构分割任务，不但目录树的遍历会产生额外开销，数据的均衡分割也将带来新的问题。本文设计如下数据项作为 MapReduce 的输入，以满足 MapReduce 任意的任务分割。

$$\begin{aligned}&\text{data entry: } <inode\_id, atime, ctime,\\ &inode_{P_1}\_id\ \text{ID}, inode_{P_2}\_id, \cdots, inode_{P_n}\_id>\end{aligned} \tag{8}$$

本文基于文件的时间戳进行窃取检测，每个输入数据项与文件一一对应。*inode_id* 表示文件的 *inode* 编号。将文件所有上层文件夹的 *inode* 编号保存在该文件的数据项中，以 $inode_P\_id$ 表示，n 表示上层文件夹的个数。每次检查点后，对最新的 EditLog 和 FsImage 进行预处理，并把更新的数据集存储在“.csv”文件中。FsImage 按照 Google protocol buffers 的格式对数据进行编码，然后以分隔的方式进行序列化后存储，其解析过程可参考文献[19]。

3.2.2 基于 MapReduce 的算法过程

MapReduce 将复杂的并行计算抽象为 2 个函数：map 和 reduce。作业分片由 map 任务以并行的方式处理，map 输出的中间结果由 reduce 任务进行合并。但集群上的可用带宽限制了 MapReduce 作业的数量。为减少 map 和 reduce 节点间的数据传输，hadoop 引入 combiner 函数。combiner 对 map 的输出在本地节点做进一步简化，但使用 combiner 需要保证 reduce 函数的输出不变。本文即采用 map、combiner 和 reduce 这 3 个函数进行数据处理，它们均以键/值对（*key*/*value*）作为输入输出。

map 函数以原始数据集作为输入，每一行作为一个 *key*/*value* 对。本文以“*inode_id*”作为 *key*，把一行中的其他数据作为 *value*。作业的输入示例如表 1 所示。其中，*atime* 和 *ctime* 采用 UNIX 纪元法，单位为毫秒，如“1490006411841”代表“2017-03-20 18:40:11”。对于给定时刻 t，map 函数主要判断文件 *atime*、*ctime* 与 t 的关系，并采用“if then”的结

表 1 MapReduce 作业输入示例

行号	*inode_id*	*atime*	*ctime*	$inodeP_1\_id$	$inodeP_2\_id$	…	$inodeP_n\_id$
0	19 449	1 490 006 411 841	1 488 326 413 425	19 448	19 028	…	16 385
1	19 450	1 490 006 412 091	1 488 326 413 464	19 448	19 028	…	16 385
2	19 479	1 490 006 680 113	1 488 326 413 506	19 478	19 028	…	16 385
3	19 480	1 490 006 680 213	1 488 326 413 522	19 478	19 028	…	16 385

构对其所有的上层文件夹进行处理。ε 表示稍大于复制所需的预计时间，本文将 ε 初始值设定为 7 200 s。具体算法如下所示。

算法 1　map 过程

输入　原始数据集，数据格式为<*key*, *value*>，具体函数中为<*inode_id*, *metadata*>。*metadata* 表示数据项中除 *inode_id* 外的其他元数据信息。

输出　数据格式为<*key*, *value*>，具体函数中为<*output_key*, *output_value*>。*output_key* 是文件上层文件夹的 *inode* 编号，*output_value* 存储了基于文件 *atime*、*ctime* 得出的判断结果。

```
map ( const key& key, const value& value ) {
  atime = value.atime, ctime = value.ctime;
  创建数组 folder_b[ ];  //存储 atime < t 文件的上层文件夹的 inode_id
  if ( ctime > t )
     跳过，读取下一项;
  else{
     if ( atime < t ) {  //文件属于 Db_t(f)
        for (inodeP in value.inodeP[ ]) {
           if not (inodeP in folder_b[ ]) {
              将 inodeP 添加到 folder_b[ ];
              output_key = inodeP;
              output_value.copy_n=0;  //表示该文件夹没有被复制
              output_value.total_n=0; }}}  //不关心该文件夹下总的文件数量
     else if ( (atime > t) && ( atime < t +ε) ) {  //文件属于截止簇
        for (inodeP in value.inodeP[]) {
           if not (inodeP in folder_b[ ]) {
              output_key = inodeP;
              output_value.copy_n=1, output_value.total_n=1; }}}  //表示该文件被复制
     else {  //文件属于 Da_t(f)
        for (inodeP in value.inodeP[]) {
           if not (inode in folder_b[ ]) {
              output_key = inodeP;
              output_value.copy_n=0, output_value.total_n=1; }}} //不确定该文件是否被复制
  }
  输出 (output_key, output_value) ;
}
```

map 函数的输出按照 *key* 值进行排序，即具有相同 *key* 值的数据项是连续存储的[20]，这为 combiner 函数的执行提供了便利。combiner 函数以 map 函数的输出作为输入，对 t 时刻之前存在访问行为的文件夹进行特殊处理，并统计其余文件夹中属于截止簇的文件数量和时刻 t 存在的总文件数量。具体算法如下所示。

算法 2　combiner 函数

输入　map 函数输出，数据格式为<*key*, *value*>

输出　数据格式为<*key*, *value*>，具体函数中为<*output_key*, *output_value*>。*output_key* 是文件上层文件夹的 *inode* 编号，*output_value* 存储了对于相同 *key* 值（即 *inode* 编号）的 *value* 值汇总结果。

```
combiner ( const key& key, const value& value ) {
  For each key { // 指 inode 编号互不相同的 key
     output_key = key;
     copy_n = value. copy_n, total_n = value. total_n;
     if 存在((copy_n==0) && (total_n==0)){
        output_value.copy_n=0, output_value.total_n= 0; }
     else {
        output_value.copy_n= 所有 key 值相同的项的 copy_n 的和;
        output_value.total_n= 所有 key 值相同的项的 total_n 的和; }
  输出 (output_key, output_value) ;}
}
```

combiner 函数执行后，map 节点将数据传输到 reduce 节点，并确保具有相同 *key* 值的数据项被同一个 reduce 节点处理。在执行 reduce 函数之前，数据项按 *key* 值进行排序。

本文主要对文件的批量复制进行检测，若犯罪人员针对特定文件进行窃取，可通过其他方法进行检测。因此，reduce 函数依据 $C_t(f)$ 和 $M_t(f)$ 的检测阈值（分别设为 C_f 和 M_f）对所有可能存在复制行为的文件夹进行过滤。检测阈值可作为参数输入，具体算法如下所示。

算法 3　reduce 函数

输入　combiner 函数输出，数据格式为<*key*, *value*>；C_f 和 M_f。

输出　数据格式为<*key*, *value*>，具体函数中为<*output_key*, *output_value*>。*output_key* 是符合阈值条件的文件夹 *inode* 编号，*output_value* 包含对应文件夹在时刻 *t* 拥有的截止簇文件数量、总文件数量（或 $M_t(f)$ 测量值）以及 $C_t(f)$ 测量值。

1) reduce (const *Key*& **key**, const *Value*& **value**, float C_f, int M_f) {

2)　For each **key** {　//指 *inode* 编号互不相同的 *key*

3)　　*copy_n* = **value**. *copy_n*, *total_n* = **value**. *total_n*;

4)　　if 存在((*copy_n*==0) && (*total_n*==0)){

5)　　　跳过; }　//过滤时刻 *t* 前存在访问行为的文件夹

6)　　else {

7)　　　*copy_sum*=所有 **key** 值相同项的 *copy_n* 的和;

8)　　　*total_sum*=所有 **key** 值相同项的 *total_n* 的和;

9)　　　$C\_test=\dfrac{copy\_sum}{total\_sum}$;

10)　　if(($C\_test > C_f$) && ($total\_sum > M_f$)){

11)　　　*output_key* ＝ **key**;

12)　　　输出(*output_key*, (*copy_sum*, *total_sum*, *C_test*));}}}

}

由于文件夹与 *inode* 的一一对应性和 HDFS 的层次性，基于输出的 *inode* 编号能够确定文件所属的用户。$C_t(f)$ 和 $M_t(f)$ 越大，该文件夹被复制的可能性越高。算法的准确率可通过调整 C_f 和 M_f 进行提升。

除文件复制外，文件系统的其他操作也可能批量更新文件的 *atime*，产生应急模式，对复制操作的判定造成影响。本文考虑以下 5 类操作：文件夹加密和压缩、文件搜索和计数以及“ls”命令。HDFS 支持文件加密，但加密过程在用户端完成，上传到 HDFS 中的数据是经过加密的[20]。HDFS 能够处理多种文件压缩格式，但压缩操作同样在数据存储前完成，HDFS 不提供文件压缩命令。对于搜索、计数和“ls”命令，经实际验证，EditLog 不记录这 3 种操作的发生时间，因此 FsImage 不会更新文件的 *atime*。此外，在 HDFS 中（尤其在云环境下），不同于个人控制的操作系统，用户能够执行的操作较少，本文不考虑由杀毒软件或文件压缩工具等额外工具实现的上述操作。因此可以认为只有正常文件访问和复制对 HDFS 文件的 *atime* 进行更新。

4　实验结果与分析

4.1　集群环境

hadoop 分布式集群由一个 NameNode 节点、一个二级 NameNode 节点和 8 个 DataNode 节点组成。集群环境基于 Xen 虚拟化平台[21]搭建，所有节点是 Xen 管理下的虚拟机。Xen 平台建立在一台浪潮服务器上，如图 1 所示。服务器、集群节点、Xen 和 hadoop 的配置如下。

服务器配置：型号为 NF5280M4；CPU 为 E5-2620v3 共 2 个；内存为 96 GB；硬盘为 3 TB；操作系统为 Ubuntu 12.04 LTS 64 位；IP 为 192.168.122.1。

集群节点配置：Intel Core I5 双核 CPU@2.6 GHz；

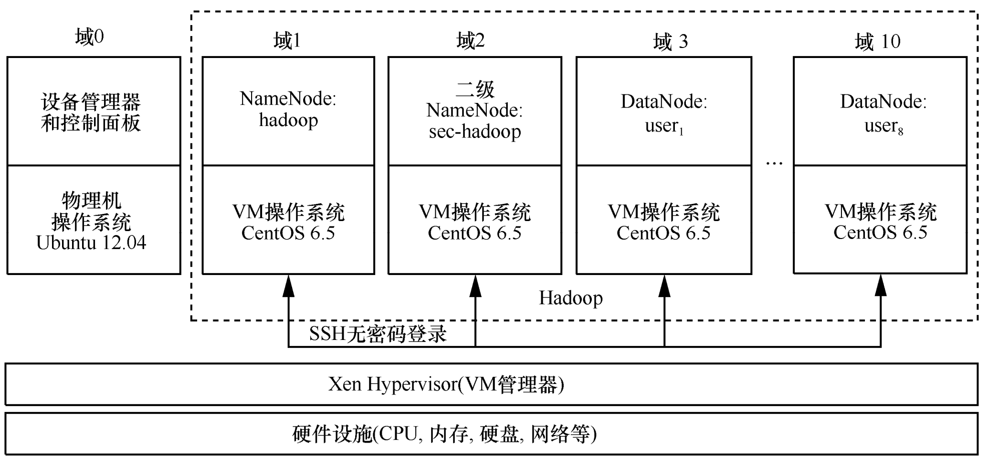

图 1　基于 Xen 的实验环境架构

4 GB 内存；200 GB 硬盘；操作系统为 CentOS 6.5 x86_64；域 1~域 10 IP 地址为 192.168.122.10~192.168.122.19。所有节点配置到其他各节点的 SSH 无密码登录，以便于命令与数据的自动化交互。

Xen 配置：版本为 4.1；hypervisor 为 64 位。

hadoop 配置：版本为 2.6.0 稳定版；数据块大小为 128 MB；复制因子为 3。在 HDFS 中创建 10 个用户：hadoop（在 NameNode 上创建），sec-hadoop（在二级 NameNode 上创建），$user_1$~$user_8$（在 8 个 DataNode 上创建）。其中，hadoop 为超级用户。

4.2　数据集

本文采用人工生成的数据集进行测试。生成过程如下：在 2017 年 3 月 1 日，10 个 hadoop 用户分别上传文件到 HDFS，每个用户的文件数量超过 100 万个，总文件量超过 1 500 万。文件类型主要为 txt、jpg、doc 等小文件。在实验室条件下，为仿真云平台中大的用户量和频繁的数据访问，在创建文件后，将数据的访问过程持续 30 天，以增加文件操作的数量。文件访问行为遵循帕累托分布[22]（即大多数的文件访问集中于少量文件，大多数文件在被上传后没有被访问）。在此期间，随机选择 10 个时刻对部分文件夹进行复制，以产生截止簇。

访问过程结束后，从二级 NameNode 中获取存储有测试文件 *ctime* 信息的 EditLog 和最近更新的 FsImage。由于用户的数据访问请求提交给 NameNode，从二级 NameNode 上获取元数据，能够减轻数据传输给 hadoop 的正常运行造成的影响。将提取的文件信息按照式(8)的格式生成数据集，存储在“.csv”文件中。本实验中，数据集中的数据条目达 15 008 320 条，数据量为 953 MB。

4.3　截止簇检测阈值的影响

本小节测试 $C_t(f)$ 和 $M_t(f)$ 的检测阈值，即 C_f 和 M_f 对算法准确率的影响。其中，C_f 测量值为 0.1、0.3、0.5、0.7、0.9，M_f 测量值为 50、100、200、500、1 000、2 000、5 000。针对特定 C_f 和 M_f 组合，输出大于检测阈值的文件夹信息。

实验过程中，首先固定 C_f，测试不同 M_f 值对算法准确率的影响，评估该算法能够检测的文件数量的下限。依据 M_f 的测量值，将 $M_t(f)$ 划分为 7 个检测段：$50 \leqslant M_t(f) < 100$，$100 \leqslant M_t(f) < 200$，$200 \leqslant M_t(f) < 500$，$500 \leqslant M_t(f) < 1000$，$100 \leqslant M_t(f) < 2\,000$，$2\,000 \leqslant M_t(f) < 5\,000$，$M_t(f) \geqslant 5\,000$。所有的 C_f 测试完毕后，针对特定 $M_t(f)$ 段，评估 C_f 的最优值。

由于 FsImage 默认 1 小时更新 1 次，从文件创建完成时开始，待检时刻 *t* 以 1 小时为间隔递增，检测 30 天中出现的所有文件复制行为，并对正确检测的文件夹个数、漏检个数、误检个数进行统计。被复制的文件夹从 A~J 依次编号。其中，$user_2$ 的文件夹下包含 20 个子文件夹，选择其中的 2 个进行复制，分别用 D-1 和 D-2 表示。表 2 表示 A~J 中大于对应 M_f 的子文件夹的个数（包括自己）。

依据各文件夹在不同 $M_t(f)$ 段的子文件夹数量，选取 A、C、G 和 H，展示不同 C_f 和 M_f 组合下的算法准确率，如图 2~图 5 所示。综合全部测试文件夹的检测结果，针对不同 C_f，各个 $M_t(f)$ 段的整体漏检率如表 3 所示。由于在实验过程中没有进行其他文件复制，算法检测到的其他复制行为全部属于误检。针对不同 C_f，各个 $M_t(f)$ 段的文件夹错误检测个数如表 4 所示。

表 2　测试文件夹构成信息

M_f	A	B	C	D-1	D-2	E	F	G	H	I	J
50	13	17	31	1	1	61	4	317	140	107	53
100	13	17	31	1	1	11	4	179	85	107	19
200	13	17	31	1	1	1	4	55	51	84	1
500	13	17	16	0	1	1	4	1	30	50	1
1 000	13	17	1	0	1	1	4	1	17	1	1
2 000	10	14	1	0	1	1	4	1	9	1	1
5 000	7	9	1	0	0	1	4	1	4	1	0

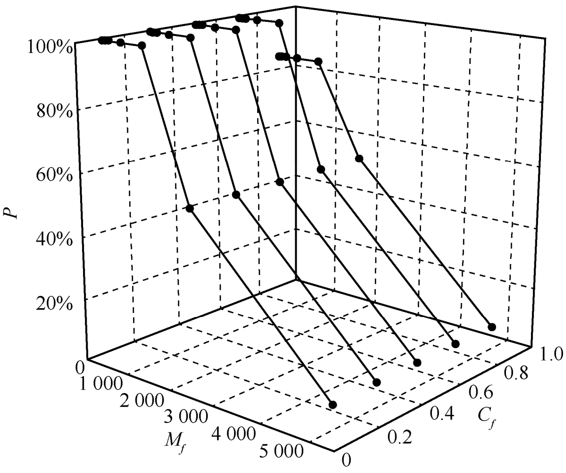

图2　A文件夹在不同 C_f 和 M_f 条件下的算法准确率

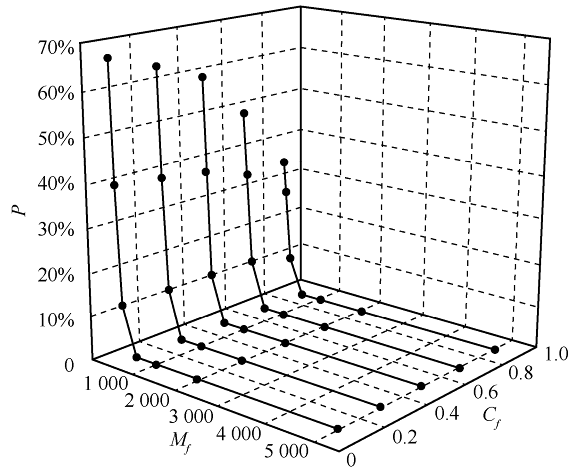

图4　G文件夹在不同 C_f 和 M_f 条件下的算法准确率

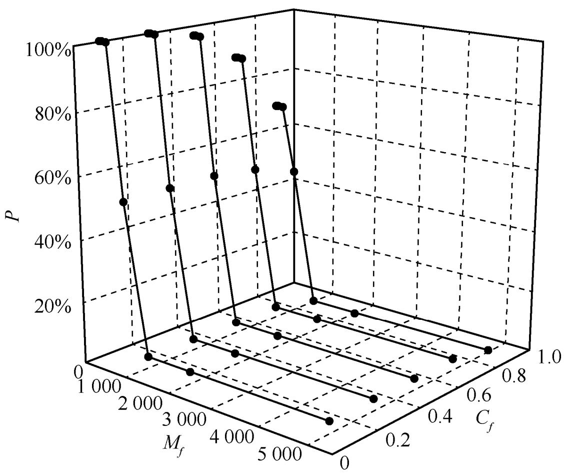

图3　C文件夹在不同 C_f 和 M_f 条件下的算法准确率

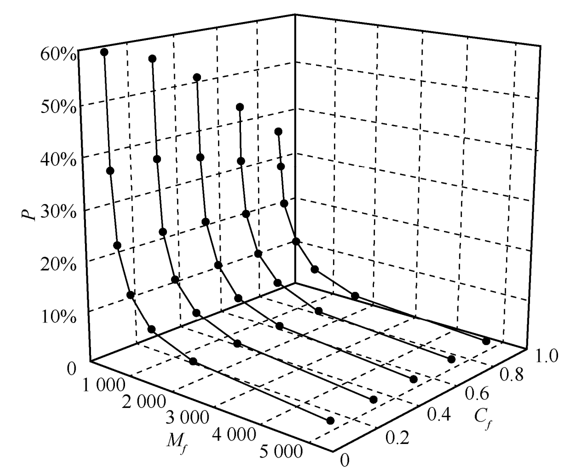

图5　H文件夹在不同 C_f 和 M_f 条件下的算法准确率

表3　各 $M_t(f)$ 段不同 C_f 条件下的整体漏检率

$M_t(f)$取值范围	C_f=0.1	C_f=0.3	C_f= 0.5	C_f=0.7	C_f=0.9
$50 \leqslant M_t(f) < 100$	6.50%	13.4%	26.4%	40.1%	57.0%
$100 \leqslant M_t(f) < 200$	1.44%	3.35%	8.61%	19.1%	34.9%
$200 \leqslant M_t(f) < 500$	0	1.60%	4.00%	11.2%	25.6%
$500 \leqslant M_t(f) < 1\,000$	0	0	0	2.60%	3.90%
$1\,000 \leqslant M_t(f) < 2\,000$	0	0	0	0	7.14%
$2\,000 \leqslant M_t(f) < 5\,000$	0	0	0	0	6.67%
$M_t(f) \geqslant 5\,000$	0	0	0	3.57%	10.7%

表4　各 $M_t(f)$ 段不同 C_f 条件下的误检文件夹个数

$M_t(f)$ 取值范围	C_f=0.1	C_f=0.3	C_f= 0.5	C_f=0.7	C_f=0.9
$50 \leqslant M_t(f) < 100$	129	87	43	17	13
$100 \leqslant M_t(f) < 200$	23	9	1	1	1
$200 \leqslant M_t(f) < 500$	3	2	0	0	0
$500 \leqslant M_t(f) < 1\,000$	0	0	0	0	0
$1\,000 \leqslant M_t(f) < 2\,000$	0	0	0	0	0
$2\,000 \leqslant M_t(f) < 5\,000$	0	0	0	0	0
$M_t(f) \geqslant 5\,000$	0	0	0	0	0

分析测试文件夹的检测结果和算法的错误检测结果，得出以下结论。

1) 漏检率受 M_f 的影响很大。H 包含各个检测段的文件夹，随着 M_f 的增大，检测正确率明显下降。采用分段检测策略，依据 $M_t(f)$ 所处范围，选择合适的 C_f，组成检测对十分必要。

2) 分段检测对的设置还需考虑对误检测的控制。本文只考虑文件夹复制和常规访问对文件 *atime* 的更新，根据帕累托分布原理，当文件夹中文件数量较大时，通过常规访问在较短时间内更新所有文件 *atime* 的可能性很小。因此误检测数量随着 $M_t(f)$ 的增大逐渐降低并最终减为 0。

3) 分析 G 的检测结果，当 $M_t(f) < 200$ 时，准确率受 C_f 的影响较大，但若 C_f 设置过低，则错误检测较多。因此，本文将算法的检测下限设为 M_f。对于少量特定文件的窃取可通过其他方法检测。

4) 分析 C 的检测结果，当 $200 \leqslant M_t(f) < 1\,000$ 时，只有 $C_f \geqslant 0.7$ 时才会对正确率产生明显影响；分析 A 的检测结果，当 $1\,000 \leqslant M_t(f) < 5\,000$ 时，漏检率主要受 M_f 的影响，C_f 对检测结果的准确率影响很小。

5) 分析 C 的检测结果，当 $C_f = 0.9$ 且 $M_f = 5\,000$ 时，正确率为 0。这是由于 C 由 $M_t(f) < 1\,000$ 的中、小型文件夹组成，并且下层文件访问频繁，产生了没有完全检测出数据复制范围的情况。鉴于一个文件夹的下层文件夹构成是不确定的，即使 $M_t(f) \geqslant 5\,000$，C_f 也不宜设置过高。

基于分段检测策略，为控制漏检率和错误检测数量，随着 $M_t(f)$ 的增大，C_f 的上、下限变化如图 6 所示。

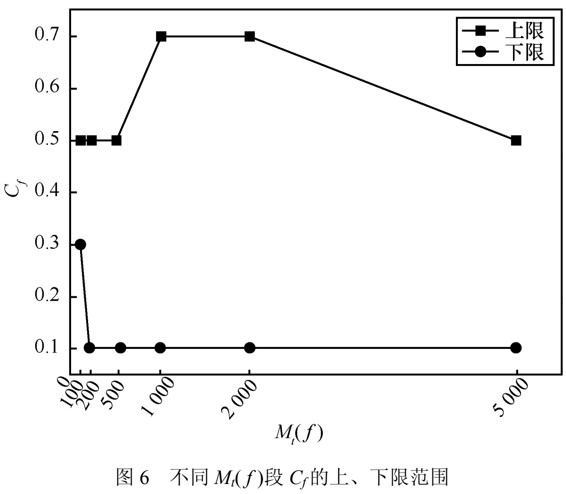

图 6　不同 $M_t(f)$ 段 C_f 的上、下限范围

被检测出复制行为的文件夹，可通过其 *inode* 编号确定相关联的用户信息。基于相关用户的行为规律和复制发生时间等，能够进一步判断是否发生数据窃取。例如，若截止簇产生在某工作日的上午，可能是由于工作人员在一段时间内对这些文件进行的正常复制；若截止簇产生在凌晨时分，则发生数据窃取的可能性较高，进一步分析执行复制操作的用户（或云内部人员）的行为规律，若发现他之前没有在凌晨访问被复制文件的记录，则可将该人员视为可疑人员，对其本地终端或移动终端进行取证分析，获取更深层次的证据，证实是否发生了数据窃取。

4.4　数据集数据量及计算节点数据的影响

本节测试数据量和计算节点数量对算法执行时间的影响，以检验该算法的可扩展性。原始数据集设为 *S*，数据量为 953 MB。测试数据量以 *Q* 表示，分别取 *S* 的 1 倍、2 倍、4 倍和 8 倍。多倍数据量通过对 *Q* 复制生成。测试节点数以 *N* 表示，取值 1、2、4 和 8。针对每个 *Q*、*N* 组合，进行 10 次实验，计算算法的平均执行时间，测试结果如表 5 所示。

从表 5 中可以看出，当 *Q*=*S* 时，除 *N*=1 外，节点数越多，执行时间反而越长。这是由于一个分片通常对应一个 HDFS 数据块（本文设为 128 MB），一个 map 或 reduce 任务分配的内存默认为 1 024 MB，因此一个节点最多能同时处理 4 个任务。当数据量相对于节点数量过小时，计算资源无法充分利用，而数据在 4 或 8 个节点上存储相对于 2 个节点更加分散，任务的调度花费更多的额外开销。当计算节点数量刚好满足需处理的数据量时，任务调度开销最小。

表 5　算法执行时间/s

N 值	*Q*=*S*	*Q*=2*S*	*Q*=4*S*	*Q*=8*S*
N=1	88	182	396	783
N=2	44	84	166	317
N=4	46	49	90	167
N=8	47.5	54.5	59	106

随着 *Q* 的增大，MapReduce 的多节点并行处理特性逐渐显示出优势。针对相同的数据量，本文将多节点相对单节点执行效率的加速比作

为衡量可扩展性的指标，图 7 展示了 Q=4S 和 Q=8S 时的加速比，此时 8 个计算节点都能得到充分运用。其中，当 Q=4S 且 N=8 时，MapReduce 平均每秒处理超过 113 万条数据，加速比值为 7.39，表现出良好的可扩展性。云存储的用户量和数据量巨大，并且随着云计算的发展不断增大，由 HDFS 元数据生成的数据集的存储和处理将产生巨大的开销，利用 MapReduce 以分布并行的方式实现数据窃取的检测算法能够显著提高检测效率。

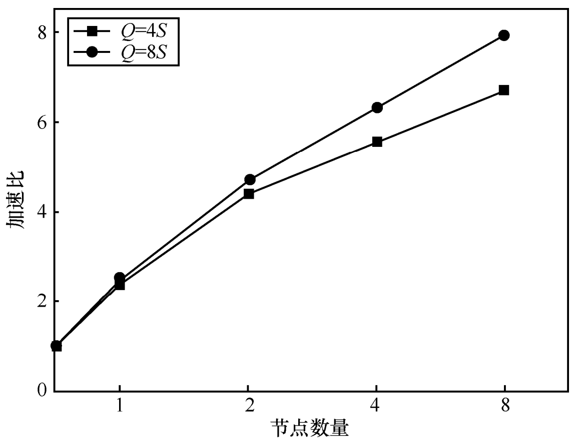

图 7　Q=4S 和 Q=8S 时的算法执行加速比

5　结束语

面对云存储的大数据量，内部人员窃取难以高效检测是云存储数据窃取检测的难点问题。本文以 HDFS 为检测对象，提出了一种基于 MapReduce 的数据窃取随机检测算法。量化分析 HDFS 文件夹复制产生的独特时间戳特性，确保能够检测恶意内部人员以合法权限发起的数据窃取，及时发现被窃取的数据范围及其所属用户。依据文件夹包含的文件数量采用分段检测的策略，通过合理设定检测阈值，能够有效控制漏检率和误检文件夹数量。利用 MapReduce 的并行处理能力，通过设计适合于 MapReduce 任务划分的数据集和算法执行过程，算法具有较高的效率，并且在 MapReduce 框架下具有良好的可扩展性。

本文算法主要对文件的批量窃取进行检测，将检测下限设为 $M_f = 100$ 。下一步将针对 DFS 环境下的用户行为特征进行建模，结合文件系统 MAC 时间戳，进一步提升算法的检测范围和准确性。

参考文献：

[1] 中国信息通信研究院. 云计算白皮书[R]. 北京: 中国信息通信研究院, 2016.
China Academy of Information and Communication Technology. White Papers of Cloud Computing[R]. Beijing: China Academy of Information and Communication Technology, 2016.

[2] 张玉清, 王晓菲, 刘雪峰, 等. 云计算环境安全综述[J]. 软件学报, 2016, 27(6): 1328-1348.
ZHANG Y Q, WANG X F, LIU X F, et al. Survey on cloud computing security [J]. Journal of Software, 2016, 27(6): 1328-1348.

[3] CHANG V, RAMACHANDRAN M. Towards achieving data security with the cloud computing adoption framework[J]. IEEE Trans Services Computing, 2016, 9(1): 138-151.

[4] MARTINI B, CHOO K K R. Cloud storage forensics: owncloud as a case study[J]. Digital Investigation, 2013, 10(4): 287-299.

[5] LI Y, GAI K, QIU L, et al. Intelligent cryptography approach for secure distributed big data storage in cloud computing[J]. Information Sciences, 2017, 387: 103-115.

[6] ALVA A, CALEFF O, ELKINS G, et al. The Notorious nine: cloud computing top threats in 2013[R]. Cloud Security Alliance, 2013.

[7] BROOK J M, FIELD S, SHACKLEFORD D, et al. The treacherous 12 - cloud computing top threats in 2016[R]. Seattle: Cloud Security Alliance, 2016.

[8] 中国云计算安全政策与法律工作组. 中国云计算安全政策与法律蓝皮书（2016）[R]. 上海: 中国云计算安全政策与法律工作组, 2016.
Cloud Computing Security Policies and Laws Group. Cloud computing security policies and laws blue book (2016)[R]. Shanghai: Cloud Computing Security Policies and Laws Group, 2016.

[9] STOLFO S J, SALEM M B, KEROMYTIS A D. Fog computing: mitigating insider data theft attacks in the cloud[C]// IEEE Symposium on Security and Privacy Workshops. 2012: 125-128.

[10] SRIRAM M, PATEL V, HARISHMA D, et al. A hybrid protocol to secure the cloud from insider threats[C]// IEEE International Conference on Cloud Computing in Emerging Markets. 2014: 1-5.

[11] NIKOLAI J, WANG Y. A system for detecting malicious insider data theft in IaaS cloud environments[C]// 2016 IEEE Global Communications Conference (GLOBECOM). 2016: 1-6.

[12] SUBASHINI S, KAVITHA V. A survey on security issues in service delivery models of cloud computing[J]. Journal of Network & Computer Applications, 2011, 34(1): 1-11.

[13] ROCHA F, CORREIA M. Lucy in the sky without diamonds: Stealing confidential data in the cloud[C]// IEEE/IFIP International Conference on Dependable Systems and Networks Workshops. 2011: 129-134.

[14] PITROPAKIS N, LYVAS C, LAMBRINOUDAKIS C. The greater the power, the more dangerous the abuse: facing malicious insiders in the

cloud[J]. Cloud Computing 2017, 2017: 156-161.

[15] MARTINI B, CHOO K K R. Distributed filesystem forensics: XtreemFS as a case study[J]. Digital Investigation, 2014, 11(4): 295-313.

[16] GRIER J. Detecting data theft using stochastic forensics[J]. Digital Investigation, 2011, 8(8): 71-77.

[17] PATEL P C, SINGH U. A novel classification model for data theft detection using advanced pattern mining[J]. Digital Investigation, 2013, 10(4): 385-397.

[18] SHVACHKO K, KUANG H, RADIA S, et al. The hadoop distributed file system[C]// 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). 2010: 1-10.

[19] GAO Y, LI B. A forensic method of efficient file extraction in HDFS based on three-level mapping[J]. Wuhan University Journal of Natural Sciences, 2017, 22(2): 114-126.

[20] WHITE T. hadoop: The definitive guide (3th Edition) [M]. Sebastopol: O'Reilly Media, Inc, 2015.

[21] BARHAM P, DRAGOVIC B, FRASER K, et al. Xen and the art of virtualization[C]// ACM SIGOPS Operating Systems Review. 2003: 164-177.

[22] MACKAY E B, CHALLENOR P G, BAHAJ A S. A comparison of estimators for the generalised Pareto distribution[J]. Ocean Engineering, 2011, 38(11): 1338-1346.

[作者简介]

高元照（1992−），男，河北衡水人，信息工程大学博士生，主要研究方向为云计算取证、大数据安全。

李炳龙（1974−），男，河南卫辉人，博士，信息工程大学副教授、硕士生导师，主要研究方向为数字取证。

陈性元（1963−），男，安徽无为人，博士，信息工程大学教授、博士生导师，主要研究方向为网络与信息安全。