Privacy-Preserving Scanning of Big Content for Sensitive Data Exposure with MapReduce[*]

Fang Liu, Xiaokui Shu, Danfeng (Daphne) Yao and Ali R. Butt
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
{fbeyond, subx, danfeng, butta}@cs.vt.edu

ABSTRACT

The exposure of sensitive data in storage and transmission poses a serious threat to organizational and personal security. Data leak detection aims at scanning content (in storage or transmission) for exposed sensitive data. Because of the large content and data volume, such a screening algorithm needs to be scalable for a timely detection. Our solution uses the MapReduce framework for detecting exposed sensitive content, because it has the ability to arbitrarily scale and utilize public resources for the task, such as Amazon EC2. We design new MapReduce algorithms for computing collection intersection for data leak detection. Our prototype implemented with the Hadoop system achieves 225 Mbps analysis throughput with 24 nodes. Our algorithms support a useful privacy-preserving data transformation. This transformation enables the privacy-preserving technique to minimize the exposure of sensitive data during the detection. This transformation supports the secure outsourcing of the data leak detection to untrusted MapReduce and cloud providers.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; C.2.4 [**Computer-Communication Networks**]: Distributed System—*distributed applications*

Keywords

Data leak detection; MapReduce; Scalability; Collection intersection

1. INTRODUCTION

The exposure of sensitive data is a serious threat to the confidentiality of organizational and personal data. Reports showed that over 800 million sensitive records were exposed in 2013 through over 2,000 incidents [13]. Reasons include compromised systems, the loss of devices, or unencrypted data storage or network transmission. While many data leak incidents are due to malicious attacks, a significant portion of the incidents are caused by unintentional mistakes of employees or data owners.

There exist several approaches for detecting data exfiltration, e.g., enforcing strict data-access policies on a host (e.g., storage capsule [7]), watermarking sensitive data sets and tracking data flow anomalies (e.g., DBMS-layer [2]) and inspecting outbound network traffic for anomalies. In the last category, the analysis proposed by Borders and Prakash [6] detects changes in network traffic patterns by searching for unjustifiable increase in HTTP traffic-flow volume, that indicates data exfiltration. The technique proposed by Shu and Yao [32] performs deep packet inspection to search for exposed outbound traffic that bears high similarity to sensitive data. Set intersection is used for the similarity measure. The intersection is computed between the set of n-grams from the content and the set of n-grams from the sensitive data.

This similarity-based detection is versatile, capable of analyzing both text and some binary-encoded context (e.g., Word or `.pdf` files). A naive implementation requires $O(nm)$ complexity, where n and m are sizes of the two sets A and B, respectively. If the sets are relatively small, then a faster implementation is to use a hashtable to store set A and then testing whether items in B exist in the hashtable or not, giving $O(n + m)$ complexity.

However, if A and B are both very large (as in our data-leak detection scenario), a naive hashtable may have hash collisions that slow down the computation. Increasing the size of the hashtable may not be practical due to memory limitation and thrashing.[1] One may attempt to distribute the dataset into multiple hashtables across several machines and coordinate the nodes to compute set intersections for leak scanning. However, such a system is nontrivial to implement from scratch and has not been reported in the literature.

In this paper, we present a data-leak detection system in MapReduce. MapReduce [14] is a programming framework for distributed data intensive applications. It has been

[*]This work has been supported in part by Security and Software Engineering Research Center (S^2ERC), a NSF sponsored multi-university Industry/University Cooperative Research Center (I/UCRC), and ARO YIP W911NF-14-1-0535.

[1]We experimentally validated this on a single host. The results are shown in Table 3 in the appendix.

used to solve security problems such as spam filtering [9, 10], Internet traffic analysis [20] and log analysis [4, 21, 36]. MapReduce algorithms can be deployed on nodes in the cloud or in local computer clusters.

However, none of these work addressed the privacy requirement of sensitive data, especially when it is outsourced to a third party for analysis. The reason is that the MapReduce nodes may be compromised or owned by semi-honest adversaries, who may attempt to gain knowledge of the sensitive data. For example, researchers demonstrated the possibility of exploring information leakage across VMs through side channel attacks in third-party compute clouds (e.g., Amazon EC2) in [30].

Although private multi-party set intersection methods exist [18], the high computational overhead is a concern for time-sensitive security applications such as data-leak detection.

In this work, we present a new MapReduce-based system to detect the occurrences of plaintext sensitive data in storage and transmission. The detection is distributed and parallel, capable of screening massive amount of content for exposed information. We address an important data privacy requirement. **In our privacy-preserving data-leak detection, MapReduce nodes scan content in data storage or network transmission for leaks without learning what the sensitive data is.**

Specifically, the data privacy protection is realized with fast one-way transformation. This transformation requires the pre- and post-processing by the data owner for hiding and precisely identifying the matched items, respectively. Both the sensitive data and the content need to be transformed and protected by the data owner, before it is given to the MapReduce nodes for the detection. In the meantime, such a transformation has to support the equality comparison required by the set intersection. This technique provides strong privacy guarantee for the data owner, in terms of the low probability for a MapReduce node to recover the sensitive data.

Besides the privacy guarantee, another advantage of our data leak solution is its scalability. Because of the intrinsic $\langle key, value \rangle$ organization of items in MapReduce, the worst-case complexity of our algorithms is correlated to the size of the leak (specifically a $\gamma \in [0, 1]$ factor denoting the size of the intersection between the content set and the sensitive data set). This complexity reduction brought by the γ factor is significant, because the value is extremely low for normal content without leak. In our algorithm, items not in the intersection (non-sensitive content) are quickly dropped without further processing. Therefore, the MapReduce-based algorithms have a lower computational complexity when compared to the traditional set-intersection implementation. Our contributions in this paper are as follows.

- We present a series of new MapReduce parallel algorithms for distributedly computing the sensitivity of content based on its similarity with sensitive data patterns. The similarity is based on collection intersection (a variant of set intersection that also counts duplicates). The MapReduce-based collection intersection algorithms are useful beyond the specific data leak detection problem.

- Our detection provides the privacy enhancement to preserve the confidentiality of sensitive data during the outsourced detection. Because of this privacy enhancement, our MapReduce algorithms can be deployed in distributed environments where the operating nodes are owned by third-party service providers. Applications of our work include data leak detection in the cloud and outsourced data leak detection.

- We implement our algorithms using the open source Hadoop framework. Our prototype outputs the degree of sensitivity for the content, and pinpoints the occurrences of potential leaks in the content. Higher sensitivity values indicate that the content is more likely to contain sensitive information. Our implementation has very efficient intermediate data representations, which significantly minimizes the disk and network I/O overhead. We performed two sets of experimental evaluations, one on Amazon EC2 and one on a local computer cluster, using large-scale email data. We achieved 225 Mbps throughput for the privacy-preserving data leak detection when processing 74 GB of content.

Our MapReduce algorithms reduce the worst-case computation complexity of set intersection by a factor of γ, where $\gamma \in [0, 1]$ is the average set intersection rate of the inputs. Because data leak is a low likelihood event, γ is usually very small in normal content, making this reduction a significant improvement.

2. THREAT MODEL AND DESIGN OVERVIEW

There are two types of input sequences in our data-leak detection model: content sequences and sensitive data sequences.

- *Content* is the data to be inspected for any occurrences of sensitive data patterns. The content can be extracted from file system and network traffic. The detection needs to partition the original content stream into **content segments**.

- *Sensitive data* contains the sensitive information that cannot be exposed to unauthorized parties, e.g., customers' records, proprietary documents. Sensitive data can also be partitioned to smaller **sensitive data sequences**.

2.1 Threat Model and Security Goal

In our model, two parties participate in the large-scale data leak detection system: data owner and data-leak detection (DLD) provider.

- *Data owner* owns the sensitive data and wants to know whether the sensitive data is leaked. It has the full access to both the content and the sensitive data. However, it only has limited computation and storage capability and needs to authorize the DLD provider to help inspect the content for inadvertent data leak.

- *DLD provider* provides detection service and has unlimited computation and storage power when compared with data owner. It can perform offline inspection without real time delay. However, the DLD provider is honest-but-curious (aka semi-honest). That is, it follows the prescribed protocol but may attempt to gain knowledge of sensitive data. The DLD provider is not given the access to the plaintext content. It can perform dictionary attack on the signature of sensitive data records.

Our goal is to offer DLD provider solutions to scan massive content for sensitive data exposure and minimize the possibility that the DLD provider learns about the sensitive information.

- **Scalability**: the ability to process content at a variety of scales, e.g., megabytes to terabytes, enabling the DLD provider to offer on-demand content inspection.
- **Privacy**: the ability to keep the sensitive data confidential, not disclosed to the DLD provider or any attacker breaking into the detection system.
- **Accuracy**: the ability to identify all leaks and only real leaks in the content, which implies low false negative/positive rates for the detection.

Our framework is not designed to detect intentional data exfiltration, during which the attacker may encrypt or transform the sensitive data.

2.2 Computation Goal

Our detection is based on computing the similarity between content segments and sensitive data sequences, specifically the intersection of two collections of n-grams. n-grams captures local features of a sequence and have also been used in other sequence similarity methods (e.g., web search duplication [8]). Following the terminology proposed by Broder *et al.* [8], we also refer to n-gram as *shingle*.

One collection consists of shingles obtained from the content segment and the other collection consists of shingles from the sensitive sequence. Collection intersection differs from set intersection, in that it also records duplicated items in the intersection, which is illustrated in Figure 1. Recording the frequencies of intersected items achieves more fine-grained detection. Thus, collection intersection is preferred for data leak analysis than set intersection.

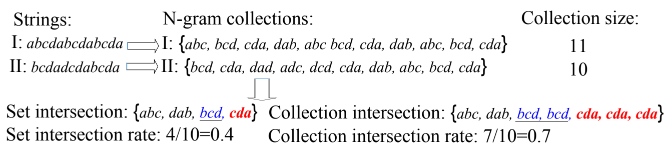
Strings:
N-gram collections:
Collection size:
I: abcdabcdabcda
I: {abc, bcd, cda, dab, abc bcd, cda, dab, abc, bcd, cda}
11
II: bcdadcdabcda
II: {bcd, cda, dad, adc, dcd, cda, dab, abc, bcd, cda}
10
Set intersection: {abc, dab, bcd, cda}
Set intersection rate: 4/10=0.4
Collection intersection: {abc, dab, bcd, bcd, cda, cda, cda}
Collection intersection rate: 7/10=0.7

Figure 1: An example illustrating the difference between set intersection and collection intersection in handling duplicates for 3-grams.

Notation used in our algorithms is shown in Table 1, including collection identifier CID, size CSize (in terms of the number of items), occurrence frequency *Snum* of an item in one collection, occurrence frequency *Inum* of an item in an intersection, and intersection rate *Irate* of a content collection with respect to some sensitive data.

Formally, given a content collection C_c and a sensitive data collection C_s, our algorithms aim to compute the intersection rate $Irate \in [0, 1]$ defined in Equation 1, where *Inum* is the occurrence frequency of an item i in the intersection $C_s \cap C_c$ (defined in Table 1). The sum of frequencies of all items appeared in the collection intersection is normalized by the size of the sensitive data collection, which yields the intersection rate *Irate*. The rate represents the percentage of sensitive data that appears in the content. *Irate* is also referred to as the *sensitivity score* of a content collection.

$$Irate = \frac{\sum_{i \in \{C_s \cap C_c\}} Inum_i}{|C_s|} \tag{1}$$

Syntax	Definition
CID	An identifier of a collection (content or sensitive data)
CSize	Size of a collection
Snum	Occurrence frequency of an item
Inum	Occurrence frequency of an item in an intersection
CSid	A pair of CIDs $\langle CID_1, CID_2 \rangle$, where CID_1 is for a content collection and CID_2 is for a sensitive data collection
Irate	Intersection rate between a content collection and a sensitive data collection as defined in Equation 1. Also referred to as the sensitivity score of the content.
ISN	A 3-item tuple of a collection ⟨identifier *CID*, size *CSize*, and the number of items in the collection⟩
CSS	An identifier for a collection intersection, consisting of a ID pair *CSid* of two collections and the size of the sensitive data collection *CSize*

Table 1: Notation used in our MapReduce algorithms.

2.3 Confidentiality of Sensitive Data

Naive collection-intersection solutions performing on *shingles* provide no protection for the sensitive data. The reason is that MapReduce nodes can easily reconstruct sensitive data from the shingles. Our detection utilizes several methods for the data owner to transform shingles before they are released to the MapReduce nodes. These transformations, including specialized hash function, provide strong-yet-efficient confidentiality protection for the sensitive information. In exchange for these privacy guarantees, the data owner needs to perform additional data pre- and post-processing operations.

In addition to protecting the confidentiality of sensitive data, the pre-processing operations also need to satisfy following requirements:

- Equality-preserving: the transformation operation should be deterministic so that two identical shingles within one session are always mapped to the same item for comparison.
- One-wayness: the function should be easy to compute given any shingle and hard to invert given the output of a sensitive shingle.

Our collection intersection (in Section 3) is computed on one-way hash values of n-grams, specifically Rabin fingerprints. Rabin fingerprint is a fast one-way hash function, which is computational expensive to invert.

Specifically, Rabin fingerprint of a n-bit shingle is based on the coefficients of the remainder of the polynomial modular operation with an irreducible polynomial $p(x)$ as the modulo as shown in Equation 2, where c_{n-i+1} is the i-th bit in the shingle.

$$f = c_1x^{n-1} + c_2x^{n-2} + ... + c_{n-1}x + c_n \ mod \ p(x) \quad (2)$$

Section 4 presents the security analysis of our approach especially on the confidentiality of sensitive data.

2.4 Workload Distribution

The details on how the workload is distributed between data owner and DLD provider is as follows and shown in Figure 2:

1. Data owner has m sensitive sequences $\{S_1, S_2, \cdots, S_m\}$ with average size $\mathcal{S}'$ and n content segments $\{C_1, C_2, \cdots, C_n\}$ with average size $\mathcal{C}'$. It obtains shingles from the content and sensitive data respectively. Then it chooses the public parameters $(n, p(x), L)$, where n is the length of shingle, $p(x)$ is the irreducible polynomial and L is the fingerprint length. The data owner computes Rabin fingerprints with Equation 2 and releases the sensitive collections $\{C_{S1}, C_{S2}, \cdots, C_{Sm}\}$ and content fingerprint collections $\{C_{C1}, C_{C2}, \cdots, C_{Cn}\}$ to the DLD provider.

2. DLD provider receives both the sensitive fingerprint collections and content fingerprint collections. It deploys MapRecuce framework and compares the n content collections with the m sensitive collections using our two-phase MapReduce algorithms. By computing the intersection rate of each content and sensitive collections pair, it outputs whether the sensitive data was leaked and reports all the data leak alerts to data owner.

3. Data owner receives the data leak alerts with a set of tuples $\{(C_{Ci}, C_{Sj}), (C_{Ck}, C_{Sl}), \cdots\}$. The data owner maps them to suspicious content segments and the plain sensitive sequences tuples $\{(C_i, S_j), (C_k, S_l), \cdots\}$. The data owner consults plaintext content to confirm that true leaks (as opposed to accidental matches) occur in these content segments and further pinpoint the leak occurrences.

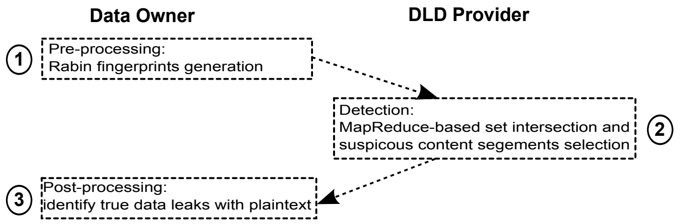
Data Owner
DLD Provider
1
Pre-processing:
Rabin fingerprints generation
Detection:
MapReduce-based set intersection and
suspicous content segements selection
2
3
Post-processing:
identify true data leaks with plaintext

Figure 2: Workload distribution for DLD provider and data owner.

2.5 MapReduce-Based Design and Challenges

MapReduce is a programming model for processing large-scale data sets on clusters. A MapReduce algorithm has two phases: `map` that supports the distribution and partition of inputs to nodes, and `reduce` that groups and integrates the nodes' outputs. MapReduce data needs to be in the format of ⟨*key, value*⟩ pair, where *key* serves as an index and the value represents the properties corresponding to the key/data item. A complex problem may require several rounds of map and reduce operations, requiring redefining and redistributing ⟨*key, value*⟩ pairs between rounds.

There exist several MapReduce-specific challenges when realizing collection-intersection based data leak detection.

1. **Complex data fields** Collection intersection with duplicates is more complex than set intersection. This requires the design of complex data fields for ⟨*key, value*⟩ pairs and a series of `map` and `reduce` operations.

2. **Memory and I/O efficiency** The use of multiple data fields (e.g., collection size and ID, shingle frequency) in ⟨*key, value*⟩ pairs may cause frequent garbage collection and heavy network and disk I/O.

3. **Optimal segmentation of data streams** While larger segment size allows the full utilization of CPU, it may cause insufficient memory problem and reduced detection sensitivity.

Our data-leak detection algorithms in MapReduce addresses these technical challenges in MapReduce framework and achieves the security and privacy goals. We design structured-yet-compact representations for data fields of intermediate values, which significantly improves the efficiency of our algorithms. Our prototype also realizes an additional post-processing partitioning and analysis, which allows one to pinpoint the leak occurrences in large content segments. We experimentally evaluate the impact of segment sizes on detection throughput and identify the optimal segment size for performance.

2.6 Detection Workflow

To compute the intersection rate of two fingerprint collections *Irate*, we design two MapReduce algorithms, DIVIDER and REASSEMBLER, each of which has a `map` and a `reduce` operation. Map and reduce operations are connected through a redistribution process. During the redistribution, outputs from map (in the form of ⟨*key, value*⟩ pairs) are sent to reducer nodes, as the inputs to the reduce algorithm. The key value of a record decides to which reducer node the it is forwarded. Records with the same key are sent to the same reducer.

1. DIVIDER takes the following as inputs: fingerprints of both content and sensitive data, and the information about the collections containing these fingerprints. Its purpose is to count the number of a fingerprint's occurrences in a collection intersection (i.e., *Inum* in Equation 1) for all fingerprints in all intersections.

   In `map` operation, it re-organizes the fingerprints to identify all the occurrences of a fingerprint across multiple content or sensitive data collections. Each `map` instance processes one collection. This reorganization traverses the list of fingerprints. Using the fingerprint as the key, it then emits (i.e., redistributes) the records with the same key to the same node.

In `reduce`, for each fingerprint in an intersection the algorithm computes the *Inum* value, which is its number of occurrences in the intersection. Each `reduce` instance processes one fingerprint. The algorithm outputs the tuple ⟨*CSS*, *Inum*⟩, where *CSS* is the identifier of the intersection (consisting of IDs of the two collections and the size of the sensitive data collection[2]). Outputs are written to MapReduce file system.

2. REASSEMBLER takes as inputs ⟨*CSS*, *Inum*⟩ (outputs from Algorithm DIVIDER). The purpose of this algorithm is to compute the intersection rates (i.e., *Irate* in Equation 1) of all collection intersections $\{C_{c_i} \cap C_{s_j}\}$ between a content collection C_{c_i} and a sensitive data collection C_{s_j}.

   In `map`, the inputs are read from the file system, and redistributed to reducer nodes according to the identifier of an intersection *CSS* (key). A reducer has as inputs the *Inum* values for all the fingerprints appearing in a collection intersection whose identifier is *CSS*. At `reduce`, it computes the intersection rate of *CSS* based on Equation 1.

In the next section, we present our algorithms for realizing the collection intersection workflow with one-way Rabin fingerprints. In Section 4, we explain why our privacy preserving technique is able to protect the sensitive data against semi-honest MapReduce nodes and discuss the causes of false alarms and the limitations.

3. COLLECTION INTERSECTION IN MAPREDUCE

We present our collection-intersection algorithm in the MapReduce framework. The algorithm computes the intersection rate of two collections as defined in Equation 1. Each collection consists of Rabin fingerprints of n-grams generated from a sequence (sensitive data or content).

RECORDREADER is a (standard) MapReduce class. We customize it to read initial inputs into our detection system and transform them into the ⟨*key, value*⟩ format required by the `map` function. The initial inputs of our RECORDREADER are content fingerprints segments and sensitive fingerprints sequences. For the DIVIDER algorithm, the RECORDREADER has two tasks: *i)* to read in each `map` split (e.g., content segment) as a whole and *ii)* to generate ⟨*CSize, fingerprint*⟩ pairs required by the `map` operation of DIVIDER algorithm.

3.1 DIVIDER Algorithm

DIVIDER is the most important and computational intensive algorithm in our system. Pseudocode of DIVIDER is given in Algorithm 1. In order to count the number of a fingerprint's occurrences in a collection intersection, the map operation in DIVIDER goes through the input ⟨*CSize, fingerprint*⟩ pairs, and reorganizes them to be indexed by fingerprint values. For each fingerprint in a collection, map records its origin information (e.g., *CID, CSize* of the collection) and *Snum* (fingerprint's frequency of occurrence in the collection). These values are useful for later intersection-rate computation.

[2]The sensitive data collection is typically much smaller than the content collection.

Algorithm 1 DIVIDER: To count the number of a fingerprint's occurrences in a collection intersection for all fingerprints in all intersections.

Input: Output of RECORDREADER in a format of ⟨*CSize, Fingerprint*⟩ as ⟨*key, value*⟩ pair.
Output: ⟨*CSS, Inum*⟩ as ⟨*key, value*⟩ pair, where *CSS* contains content collection ID, sensitive data collection ID and the size of the sensitive data collection. *Inum* is occurrence frequency of a fingerprint in the collection intersection.

```
function DIVIDER::MAPPER(CSize, Fingerprint)
    ▷ Record necessary information for the collection.
    ISN←CID, CSize and Snum
    Emit⟨Fingerprint, ISN⟩
end function
function DIVIDER::REDUCER(Fingerprint, ISNlist[$c_1, \ldots, c_n$])
    $j = 0, k = 0$
    ▷ Divide the list into a sensitive list and a content list
    for all $c_i$ in ISNlist do
        if $c_i$ belongs to sensitive collections then
            SensList[++j]← $c_i$
        else
            ContentList[++k] ← $c_i$
        end if
    end for
    ▷ Record the fingerprint occurrence in the intersection
    for all sens in SensList do
        for all content in ContentList do
            Size ←sens.CSize
            Inum ←Min(sens.Snum, content.Snum)
            CSS← ⟨content.CID, sens.CID, Size⟩
            Emit ⟨CSS, Inum⟩
        end for
    end for
end function
```

Steps of our MapReduce algorithms are illustrated with a running example (with four MapReduce nodes) in Figure 3. The example has two content collections C_1 and C_2, and two sensitive data collections S_1 and S_2. The sizes of their corresponding collections are 3, 4, 3 and 3, respectively. E.g., in node 1 of Figure 3, map outputs the pair $\langle a, (C_1, 3, 1)\rangle$, indicating fingerprint (key) a is from content collection C_1 of size 3 and occurs once in C_1.

The advantage of using the fingerprint as the key in the map's outputs is that it allows the reducer to quickly identify non-intersected items. After redistribution, entries having the same fingerprint are sent to the same reducer node as inputs to the reduce algorithm. E.g., in Figure 3, all occurrences of fingerprint a are sent to node 1, including two occurrences from content collections C_1 and C_2, one occurrence from sensitive data collection S_1.

Reduce algorithm is more complex than map. It partitions the occurrences of a fingerprint into two lists, one list (*ContentList*) for the occurrences in content collections and one for sensitive data (*SensList*). It then uses a double for-loop to identify the fingerprints that appear in intersections, e.g., a. Non-intersected fingerprints are not analyzed, significantly reducing the computational overhead. This reduction is reflected in our computational complexity analysis in Table 2, specifically the $\gamma \in [0, 1]$ reduction factor representing the size of intersection.

The for-loops also compute the occurrence frequency *Inum* of the fingerprint in an intersection. E.g., for node 3 in Figure 3, d appears once in $C_1 \cap S_1$ and once $C_1 \cap S_2$. The output

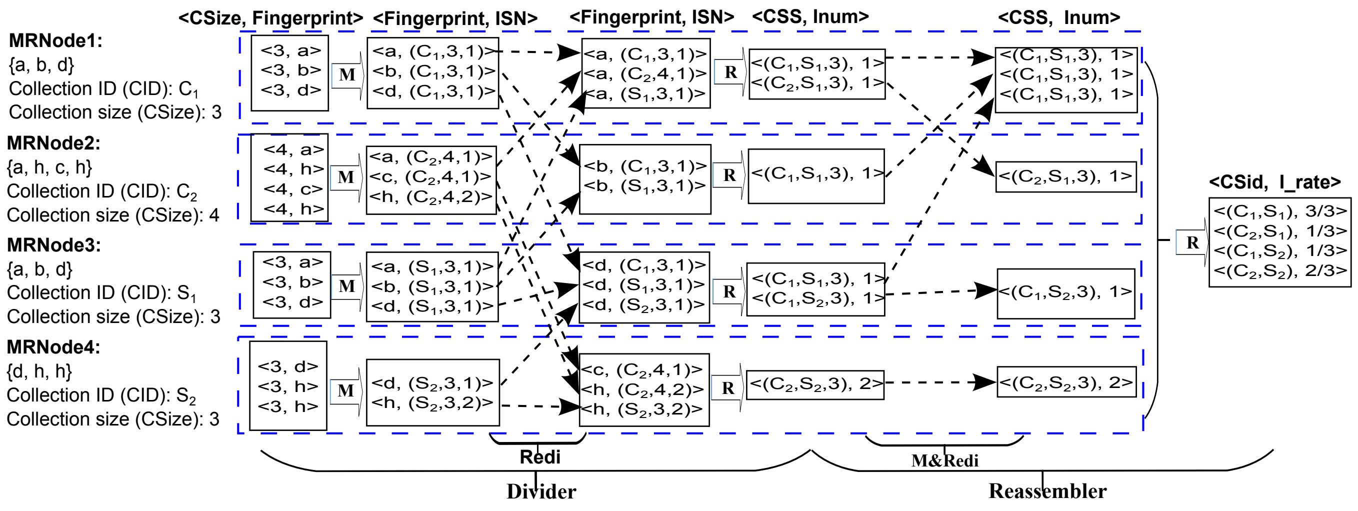

Figure 3: A Running example illustrating Divider and Reassembler algorithms, with four MapReduce nodes, two content collections C_1 and C_2, and two sensitive data collections S_1 and S_2. **M**, **R**, **Redi** stand for map, reduce, and redistribution, respectively. ⟨key, value⟩ of each operation is shown at the top.

of the algorithm is the ⟨*CSS, Inum*⟩ pairs, indicating that a fingerprint occurs *Inum* number of times in a collection intersection whose identifier is *CSS*.

3.2 Reassembler Algorithm

Algorithm 2 Reassembler: To compute the intersection rates *Irate* of all collection intersections $\{C_{c_i} \cap C_{s_j}\}$ between a content collection C_{c_i} and a sensitive data collection C_{s_j}.

Input: Output of Divider in a format of ⟨*CSS, Inum*⟩ as ⟨*key, value*⟩ pairs.
Output: ⟨*CSid, Irate*⟩ pairs where *CSid* represents a pair of a content collection ID and a sensitive collection ID, while *Irate* represents the intersection rate between them

```
1: function Reassembler::Mapper(CSS, Inum)
2:     Emit⟨CSS,Inum⟩
3: end function

1: function Reassembler::Reducer(CSS, Inum[$n_1, \dots, n_n$])
2:     intersection ← 0
3:     ▷ Add up all the elements in Inum[]
4:     for all $n_i$ in Inum[] do
5:         intersection ← intersection + $n_i$
6:     end for
7:     CSid ← CSS.CSid
8:     ▷ Compute intersection rate
9:     Irate ← $\frac{|intersection|}{CSS.CSize}$
10:    Emit ⟨CSid, Irate⟩
11: end function
```

The purpose of Reassembler is to compute the intersection rates *Irate* of all collection-and-sensitive-data intersections. Pseudocode of Reassembler is in Algorithm 2. The map operation in Reassembler emits (i.e., redistributes) inputs ⟨*CSS, Inum*⟩ pairs according to their key *CSS* values to different reducers. The reducer can then compute the intersection rate *Irate* for the content and sensitive data collection pair. I.e., this redistribution sends all the intersected items between a content collection C_{c_i} and a sensitive data collection C_{s_j} to the same reducer. E.g., in Figure 3, all the pairs with $(C_1, S_1, 3)$ as *key* are sent to MapReduce node 1. In node 1, the total number of fingerprints shared by C_1 and S_1 is 3. The intersection rate is 1.

Complexity Analysis The computational and communication complexities of various operations of our algorithm are shown in Table 2. We denote the average size of a sensitive data collection by $\mathcal{S}$, the average size of a content collection by $\mathcal{C}$, the number of sensitive data collections by m, the number of content collections by n, and the average intersection rate by $\gamma \in [0, 1]$. Without loss of generality, we assume that $|\mathcal{S}| < |\mathcal{C}|$ and $|\mathcal{S}m| < |\mathcal{C}n|$. We do not include post-processing in complexity analysis. Our total communication complexity $O(\mathcal{C}n + \mathcal{S}mn\gamma)$ covers the number of records (⟨*key, value*⟩ pairs) that all operations output. For a hashtable-based (non-MapReduce) approach, where each content collection is stored in a hashtable (total n hashtables of size $\mathcal{C}$ each) and each sensitive data item (total $\mathcal{S}m$ items) is compared against all n hashtables, the computational complexity is $O(\mathcal{C}n + \mathcal{S}mn)$.

4. SECURITY ANALYSIS AND DISCUSSION

MapReduce nodes that perform the data-leak detection may be controlled by honest-but-curious providers (aka semi-honest), who follow the protocol, but may attempt to gain knowledge of the sensitive data information (e.g., by logging the intermediate results and making inferences). We analyze the security and privacy guarantees provided by our MapReduce based data leak detection system. The privacy goal of our system is to prevent the sensitive data from being exposed to DLD provider or untrusted nodes.

Privacy guarantee Let f_s be the Rabin fingerprint of sensitive data shingle s. Using the algorithms in Section 3, a MapReduce node knows fingerprint f_s but not shingle s of the sensitive data. We assume that attackers are not able to

Algorithm	Comp.	Comm.
Our Pre-processing	$O(\mathcal{C}n + \mathcal{S}m)$	$O(\mathcal{C}n + \mathcal{S}m)$
Our Divider::M.	$O(\mathcal{C}n + \mathcal{S}m)$	$O(\mathcal{C}n + \mathcal{S}m)$
Our Divider::R.	$O(\mathcal{C}n + \mathcal{S}mn\gamma)$	$O(\mathcal{S}mn\gamma)$
Our Reassembler::M.	$O(\mathcal{S}mn\gamma)$	$O(\mathcal{S}mn\gamma)$
Our Reassembler::R.	$O(\mathcal{S}mn\gamma)$	$O(mn)$
Our Total	$O(\mathcal{C}n + \mathcal{S}mn\gamma)$	$O(\mathcal{C}n + \mathcal{S}mn\gamma)$
Hashtable	$O(\mathcal{C}n + \mathcal{S}mn)$	N/A

Table 2: Computation and communication complexity of each phase in our MapReduce algorithm and that of the conventional hashtable-based approach. We denote the average size of a sensitive data collection by $\mathcal{S}$, the average size of a content collection by $\mathcal{C}$, the number of sensitive data collections by m, the number of content collections by n, and the average intersection rate by $\gamma \in [0, 1]$.

infer s in polynomial time from f_s. This assumption is guaranteed by the one-way Rabin fingerprinting function [29].

In addition, the data owner chooses a different irreducible polynomial $p(x)$ for each session. Under this configuration, the same shingle is mapped to different fingerprints in multiple sessions. The advantage of this design is the increased randomization in the fingerprint computation, making it more challenging for the DLD provider to correlate values and infer preimage. This randomization also increases the difficulty of dictionary attacks. Other cryptographic mechanisms, e.g., XORing a secret session key with content and sensitive data before fingerprinting, achieve similar security improvement. Because the transformation needs to preserve equality comparison (in Section 2.3), the configuration needs to be consistent within a session.

Collisions Two types of collisions are involved in our detection framework: fingerprint collisions and coincidental matches. Fingerprint collisions rarely happen as long as the length of fingerprint is sufficiently long (64 bits fingerprints are sufficient long with the collision probability being less than 10^{-6}, according to the study by Broder [8]). Coincidental match occurs when some shingles in content happens to mach some in sensitive data. These shingle matches may be due to shorter shingles, large content segment or sensitive data containing widely used patterns. With proper shingle length and threshold setting, the detection accuracy would not be affected by the coincidental matches. We perform accuracy experiment in Section 5.4, which shows that the intersection rate for normal leak is much higher than that caused by coincidental matches.

Our data leak detection framework is designed to detect accidental data leaks in content, instead of intentional data exfiltration. In intentional data exfiltration, which cannot be detected by deep packet inspection, an attacker may encrypt or transform the sensitive data.

5. IMPLEMENTATION AND EVALUATION

We implement our algorithms with Java in Hadoop, which is an open-source software system implementing MapReduce. We set the length of fingerprint and shingle to 8 bytes (64 bits). This length was previously reported as optimal for robust similarity test [8], as it is long enough to preserve some local context and short enough to resist certain transformations. Our prototype implements an additional IP-based post-processing analysis and partition focusing on the suspicious content. It allows the data owner to pinpoint the IPs of hosts where leaks occur. We output the suspicious content segments and corresponding hosts.

We make several technical measures to reduce disk and network I/O. We use (structured) SequenceFile format as the intermediate data format. We minimize the size of ⟨*key, value*⟩ pairs. E.g., the size of *value* after `map` in DIVIDER is 6 bytes on average. We implement COMBINATION classes that significantly reduce the amount of intermediate results written to the distributed file systems (DFS). This reduction in size is achieved by aggregating same ⟨*key, value*⟩ pairs. This method reduces the data volume by half. We also enable Hadoop compression, which gives as high as 20-fold size reduction.

We deploy our algorithms in two different 24-node Hadoop systems, a local cluster and Amazon Elastic Compute Cloud (EC2). For both environments, we set one node as master node and the rest as slave nodes.

- *Amazon EC2:* 24 nodes each having a c3.2xlarge instance with 8 CPUs and 15 GB RAM.
- *Local cluster:* 24 nodes each having two quad-core 2.8 GHz Xeon processors and 8 GB RAM.

We use the Enron Email Corpus, including both email header and body to perform the performance experiments. The entire dataset is used as content and a small subset of it as the sensitive data. For the accuracy experiment, we use 10 academic research paper as the sensitive data and transformed Enron emails as content (insert `</br>` at each new line).

Our experiments aim to answer the flowing questions.

1. How does the size of content segment affect the analysis throughput? (Section 5.1)
2. What is the throughput of our analysis on Amazon EC2 and the local clusters? (Section 5.2)
3. How does the size of sensitive data affect the detection performance? (Section 5.3)
4. What is the detection accuracy of our data leak detection system? (Section 5.4)

Suppose we have n content segments with s of them containing sensitive sequences ($s < n$). During the detection, the content segments with intersection rates above the threshold t raise alerts in our detection algorithms. Suppose m content segments raise the alerts and s' of them are true leak instances. Then, we define the following metrics:

- detection rate as $\frac{s'}{s}$.
- false positive rate as $\frac{m-s'}{n-s}$.

5.1 Optimal Size of Content Segment

Content volume is usually overwhelmingly larger than sensitive data, as new content is generated continuously in storage and in transmission. Thus, we evaluate the throughput of different sizes and numbers of content segments in order to find the optimal segment size for scalability. A content segment with size $\hat{C}$ is the original sequence that is used to

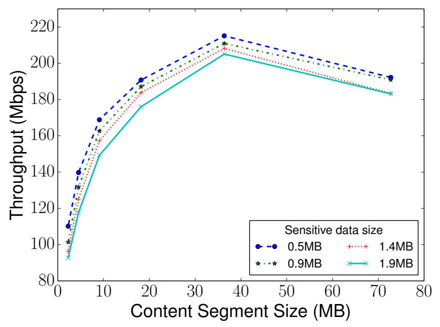

Figure 4: Throughput with different sizes of content segments. For each setup (line), the size of the content analyzed is 37 GB.

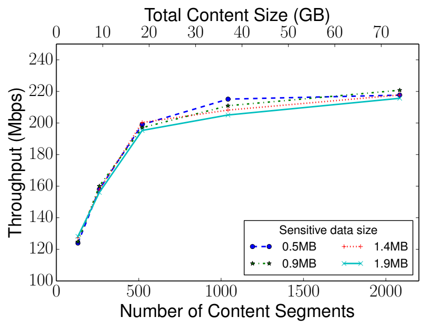

Figure 5: Throughput with different amount of content workload. Each content segment is 37 MB.

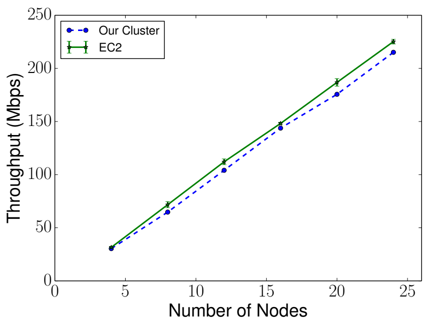

Figure 6: Throughput with different number of nodes on a local cluster or Amazon EC2.

generate the n-gram content collection. A sensitive sequence with size $\hat{S}$ is the original sequence that is used to generate the n-gram sensitive collection.

The total size of content analyzed is 37 GB, which consists of multiple copies of Enron data. Detection performance under different content segment sizes (from 2 MB to 80 MB) is measured. We vary the size of sensitive data from 0.5 MB to 2 MB. The results are shown in Figure 4.

We observe that when $\hat{C} < 37$ MB, the throughput of our analysis increases with the size $\hat{C}$ of content segment. When $\hat{C}$ becomes larger than 37 MB, the throughput begins to decrease. The reason for this decrease is that more computation resources are spend on garbage collection with larger $\hat{C}$. There are over 16 processes running at one nodes at the same time. We assign 400 MB memory for each process to process 37x8 MB shingles. Thus, we set the size of content segments to 37 MB for the rest of our experiments. This size also allows the full utilization of the Hadoop file system (HDFS) I/O capacity without causing out-of-memory problems.

We also evaluate the throughput under a varying number of content segments n, i.e., workload. The results are shown in Figure 5, where the total size of content analyzed is shown at the top X-axis (up to 74 GB). Throughput increases as workload increases as expected.

In both experiments, the size of sensitive data is small enough to fit in one collection. Larger size of sensitive data increases the computation overhead, which explains the slight decrease in throughput in both Figures 4 and 5.

5.2 Scalability

For scalability evaluation, we processed 37 GB content with different numbers of nodes, 4, 8, 12, 16, 20, and 24. The experiments were deployed on both on the local cluster and on Amazon EC2. Our results are shown in Figure 6. The system scales well, as the throughput linearly increases with the number of nodes. The peak throughput observed is 215 Mbps on the local cluster and 225 Mbps on Amazon EC2. EC2 cluster gives 3% to 11% performance improvement. This improvement is partly due to the larger memory. The standard error bars of EC2 nodes are shown in Figure 6 (from three runs). Variances of throughputs on the local cluster are negligible.

We break down the total overhead based on the DIVIDER and REASSEMBLER operations. The results are shown in Figure 7 with the runtime (Y-axis) in a log scale. DIVIDER algorithm is much more expensive than REASSEMBLER, accounting for 85% to 98% of the total runtime. With increasing content workload, DIVIDER's runtime increases, more significantly than that of REASSEMBLER.

These observations are expected, as DIVIDER algorithm is more complex. Specifically, both `map` and `reduce` in DIVIDER need to touch *all* content items. Because of the large content volume, these operations are expensive. In comparison, REASSEMBLER algorithm only touches the intersected items, which is substantially smaller for normal content without leaks. These experimental results are consistent with our complexity analysis in Table 2.

5.3 Performance Impact of Sensitive Data

We reorganize the performance results in Figure 4 so that the sizes of sensitive data are shown at the X-axis. The new figure is Figure 8, where each setup (line) processes 37 GB data and differs in their size for content segment. There are a few observations. First, smaller content segment size incurs higher computational overhead, e.g., for keeping tracking the collection information (discussed in Section 5.1).

The second observation is that the runtime increases as the size of sensitive data increases, which is expected. Experiments with the largest content segment (bottom line in Figure 8) have the smallest increase, i.e., the least affected by the increasing volume of sensitive data.

This difference in intercept is explained next. The total computation complexity is $O(\mathcal{C}n + \mathcal{S}mn\gamma)$ (Table 2). In $O(\mathcal{C}n + \mathcal{S}mn\gamma)$, $n\gamma$ serves as the coefficient (i.e., intercept), as the total size $\mathcal{S}m$ of sensitive data increases, where n is the number of content segments. When 37 GB content is broken into small segments, n is large. A larger coefficient magnifies the increase in sensitive data, resulting in more substantial overhead increase. Therefore, the line at the bottom of Figure 8 represents our recommended configuration with a large 37 MB content segment size.

5.4 Detection Accuracy

We used Enron emails as content and 10 academic papers as sensitive data. We pre-processed the content into four types: content not containing sensitive data (C_c), content containing sensitive data (C_s), transformed content not containing sensitive data (T_c) and transformed content containing sensitive data (T_s). We transformed the content data by

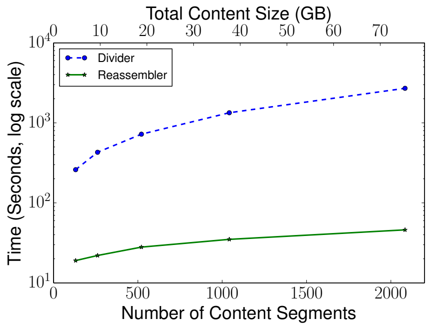

Figure 7: Runtime of DIVIDER and REASSEMBLER algorithms. The DIVIDER operation takes 85% to 98% of the total runtime. The Y-axis is in 10 based log scale.

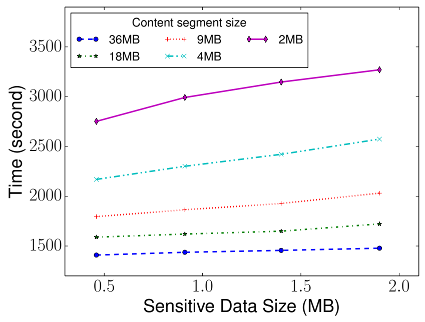

Figure 8: Runtime with a vary size of sensitive data. The content volume is 37.5 GB for each setup. Each setup (line) has a different size for content segments.

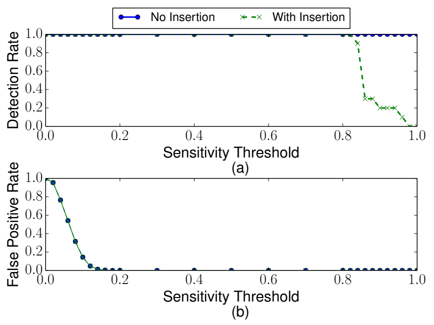

Figure 9: Detection accuracy with different threshold. The detection rate is 1 with very low false positive rate when the thresholds ranges from 0.18 to 0.8. In Figure (b), the false positive rates of the two lines are very close.

inserting `</br>` at each new line. Detection rates are computed from C_s and T_s. If alerts are raised by the segments from these two types of data, the alerts are true positives. The false positive rates are computed from C_c and T_c. If alerts are are raised by segments from these two types of data, the alerts are false positives. Our results on false positive rates and detection rates are shown in Figure 9.

With sensitivity threshold ranging from 0.18 to 0.82, the detection rate is 1, with 0 false positive rate for both the transformed leak and the non-transformed leak. The false positive rates of transformed data leak and non-transformed data leak are very close (the two lines mostly overlap). The false positive rate of the transformed data leak is on average 2.55% lower than that of the non-transformed leak, when the sensitivity threshold is smaller than 0.18. When the sensitivity threshold ranges from 0.3 to 0.7, our detection rate is 1 and false positive rate is 0.

Summary We summarize our experimental findings below.

1. Our MapReduce-based data leak detection algorithms linearly scale with the number of nodes. We achieved 225 Mbps throughput on Amazon EC2 cluster and a similar throughput on our local cluster. *Divider* algorithm accounts for 85% to 98% of the total runtime.

2. We observed that larger content segment size $\hat{C}$ (up to 37 MB) gives higher performance. This observation is due to the decreased amount of bookkeeping information for keeping track of collections, which results in significantly reduced I/O overhead associated with intermediate results.

   When the content segment size $\hat{C}$ is large (37 MB), we observed that the increase in the amount of sensitive data has a relatively small impact on the runtime. Give the content workload, larger $\hat{C}$ means fewer number of content segments, resulting in a smaller coefficient.

3. We validated that our detection system has high detection accuracy for some transformed data leaks. By setting the threshold to be a proper value, our algorithms can detect the leaks with low false alerts.

4. *Limitations* Our method is designed for detecting accidental data exposure in content, but not for intentional data exfiltration, which typically uses strong encryption. Detecting malicious data leaks, in particular those by insiders, is still an active research area. Coincidental matches may generate false positives in our detection, e.g., an insensitive phone number in the content happens to match part of a sensitive credit card number. A possible mitigation is for the data owner to further inspect the context surrounding the matching shingles and interpret its semantics.

6. RELATED WORK

One may adopt network-based and/or host-based approaches to prevent personal or organizational sensitive information from being leaked. Solutions from both paradigms are necessary, and complementary to each other.

Borders and Prakash [6] presented a network-analysis approach for estimating information leak in the outbound traffic. The method identifies anomalous and drastic increase in the amount of information carried by the traffic. The method was not designed to detect small-scale data leak. In comparison, our technique is based on intersection-based pattern matching analysis. Thus, our method is more sensitive to small and stealthy leaks than the approach in [6]. In addition, our analysis can also be applied to content in data storage (e.g., data center), besides network traffic.

Croft and Caesar [12] compared two logical copies of network traffic to control the movement of sensitive data. The work by Papadimitriou and Garcia-Molina [26] aims at finding the agents that leaked the sensitive data. Shu and Yao [32] presented privacy-preserving methods for protecting sensitive data in a non-MapReduce based detection environment. Shu *et al.* [33] further proposed to accelerate screening transformed data leaks using GPU. Blanton *et al.* [5] proposed a solution for fast outsourcing of sequence edit distance and secure path computation, while preserving the confidentiality of the sequence.

Examples of host-based approaches include Auto-FBI [43] and Aquifer [23]. Auto-FBI guarantees the secure access of sensitive data on the web. It achieves this guarantee by automatically generating a new browser instance for sensitive content. Aquifer is a policy framework and system. It helps prevent accidental information disclosure in OS.

MapReduce framework was used to solve problems in data mining [41], machine learning [25], database [4, 35], and bioinformatics [22]. MapReduce algorithms for computing document similarity (e.g., [1, 15, 39]) involve pairwise similarity comparison. Similarity measures may be Hamming distance, edit distance or Jaccard distance. MapReduce was also used by security applications, such as log analysis [19, 36], spam filtering [9, 10] and malware and botnet detection [17, 28, 42] for scalability. The security solutions proposed by Bilge *et al.* [3], Yang *et al.* [36] and Yen *et al.* [37] analyzed network traffic or logs with MapReduce, searching for malware signatures or behavior patterns. Our MapReduce-based data leak detection problem is new, which none of the existing MapReduce solutions addresses. In addition, our detection goal differs from the aforementioned solutions.

Several techniques have been proposed to improve the privacy protection of MapReduce framework. Such solutions typically assume that the cloud provider is trustworthy. For example, Pappas *et al.* [27] proposed a data-flow tracking method in cloud applications. It audits the use of the sensitive data sent to cloud. Roy *et al.* [31] integrates mandatory access control with differential privacy in order to manage the use of sensitive data in MapReduce computations. Yoon and Squicciarini [38] detected malicious or cheating MapReduce nodes by correlating different nodes' system and Hadoop logs. Squicciarini *et al.* [34] presented techniques that prevent information leakeage from the indexes of data in the cloud. In comparison, Our work has a different security model. We assume that MapReduce algorithms are developed by trustworthy entities, yet the MapReduce provider may attempt to gain knowledge of the sensitive information.

There exist MapReduce algorithms for computing the set intersection [4, 35]. They differ from our collection intersection algorithms, as explained in Section 2. Our collection intersection algorithm requires new intermediate data fields and processing for counting and recording duplicates in the intersection. Several techniques were developed for monitoring or improving MapReduce performance, e.g., to identify nodes with slow tasks [11], GPU acceleration [16] and efficient data transfer [21]. These advanced techniques can be applied to further speed up our prototype.

7. CONCLUSIONS AND FUTURE WORK

Our work is motivated by the increasing number of accidental data leak issues in organizational personal environments. We presented a MapReduce system for detecting the occurrences of sensitive data patterns in massive-scale content in data storage or network transmission. Our system provides privacy enhancement to minimize the exposure of sensitive data during the outsourced detection. We deployed and evaluated our prototype with the Hadoop platform on Amazon EC2 and a local cluster, and achieved 225 Mbps analysis throughput. For future work, we plan to explore the deployment of our detection system to hybrid cloud environments, which consist of private machines owned by the data owner and public machines owned by the cloud provider. The use of hybrid cloud infrastructure will likely improve the efficiency of our detection system. We also plan to explore the use of hybrid cloud (e.g., [24, 40]) for the collection-intersection computation.

8. ACKNOWLEDGEMENTS

The authors acknowledge the open access subvention fund from Virginia Tech. The authors thank Michael Wolfe for his support and feedback on the work.

9. REFERENCES

[1] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. Document similarity self-join with MapReduce. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 731–736. IEEE Computer Society, 2010.

[2] Elisa Bertino and Gabriel Ghinita. Towards mechanisms for detection and prevention of data exfiltration by insiders: Keynote talk paper. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 10–19, New York, NY, USA, 2011. ACM.

[3] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. Disclosure: Detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 129–138, New York, NY, USA, 2012. ACM.

[4] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[5] Marina Blanton, Mikhail J. Atallah, Keith B. Frikken, and Qutaibah M. Malluhi. Secure and efficient outsourcing of sequence comparisons. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 505–522, 2012.

[6] Kevin Borders and Atul Prakash. Quantifying information leaks in outbound web traffic. In *IEEE Symposium on Security and Privacy*, pages 129–140. IEEE Computer Society, 2009.

[7] Kevin Borders, Eric Vander Weele, Billy Lau, and Atul Prakash. Protecting confidential data on personal computers with storage capsules. In *USENIX Security Symposium*, pages 367–382. USENIX Association, 2009.

[8] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching, 11th Annual Symposium*, volume 1848 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2000.

[9] Godwin Caruana, Maozhen Li, and Hao Qi. SpamCloud: A MapReduce based anti-spam architecture. In *Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, pages 3003–3006. IEEE, 2010.

[10] Godwin Caruana, Maozhen Li, and Man Qi. A MapReduce based parallel SVM for large scale spam filtering. In *Eighth International Conference on Fuzzy*

Systems and Knowledge Discovery, pages 2659–2662. IEEE, 2011.

[11] Qi Chen, Cheng Liu, and Zhen Xiao. Improving MapReduce performance using smart speculative execution strategy. *Computers, IEEE Transactions on*, 63(4):954–967, April 2014.

[12] Jason Croft and Matthew Caesar. Towards practical avoidance of information leakage in enterprise networks. In *6th USENIX Workshop on Hot Topics in Security, HotSec'11*. USENIX Association, 2011.

[13] Data Loss DB. `http://datalossdb.org/statistics`.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[15] Tamer Elsayed, Jimmy J. Lin, and Douglas W. Oard. Pairwise document similarity in large collections with MapReduce. In *ACL (Short Papers)*, pages 265–268. The Association for Computer Linguistics, 2008.

[16] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K. Govindaraju. Mars: Accelerating MapReduce with graphics processors. *IEEE Trans. Parallel Distrib. Syst.*, 22(4):608–620, 2011.

[17] Jérôme François, Shaonan Wang, Walter Bronzi, Radu State, and Thomas Engel. BotCloud: Detecting botnets using MapReduce. In *IEEE International Workshop on Information Forensics and Security*, pages 1–6. IEEE, 2011.

[18] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques*, volume 3027 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2004.

[19] Xiaoyu Fu, Rui Ren, Jianfeng Zhan, Wei Zhou, Zhen Jia, and Gang Lu. LogMaster: Mining event correlations in logs of large-scale cluster systems. In *IEEE 31st Symposium on Reliable Distributed Systems*, pages 71–80. IEEE, 2012.

[20] Youngseok Lee, Wonchul Kang, and Hyeongu Son. An Internet traffic analysis method with MapReduce. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, pages 357–361, April 2010.

[21] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, and Kenneth Yocum. In-situ MapReduce for log processing. In *USENIX Annual Technical Conference*. USENIX Association, 2011.

[22] Andréa M. Matsunaga, Maurício O. Tsugawa, and José A. B. Fortes. Cloudblast: Combining MapReduce and virtualization on distributed resources for bioinformatics applications. In *eScience*, pages 222–229. IEEE Computer Society, 2008.

[23] Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *ACM Conference on Computer and Communications Security*, pages 1029–1042. ACM, 2013.

[24] Kerim Yasin Oktay, Vaibhav Khadilkar, Bijit Hore, Murat Kantarcioglu, Sharad Mehrotra, and Bhavani Thuraisingham. Risk-aware workload distribution in hybrid clouds. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 229–236, Washington, DC, USA, 2012. IEEE Computer Society.

[25] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. Planet: Massively parallel learning of tree ensembles with MapReduce. *Proc. VLDB Endow.*, 2(2):1426–1437, August 2009.

[26] Panagiotis Papadimitriou and Hector Garcia-Molina. Data leakage detection. *IEEE Trans. Knowl. Data Eng.*, 23(1):51–63, 2011.

[27] Vasilis Pappas, VasileiosP. Kemerlis, Angeliki Zavou, Michalis Polychronakis, and AngelosD. Keromytis. Cloudfence: Enabling users to audit the use of their cloud-resident data. In *Research in Attacks, Intrusions, and Defenses*, volume 8145 of *Lecture Notes in Computer Science*, pages 411–431. Springer Berlin Heidelberg, 2013.

[28] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser: Analysis of web-based malware. In *First Workshop on Hot Topics in Understanding Botnets*. USENIX Association, 2007.

[29] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Aliken Computation Laboratory, 1981.

[30] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[31] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 297–312. USENIX Association, 2010.

[32] Xiaokui Shu and Danfeng (Daphne) Yao. Data leak detection as a service. In *SecureComm*, volume 106 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 222–240. Springer, 2012.

[33] Xiaokui Shu, Jing Zhang, Danfeng (Daphne) Yao, and Wu-Chun Feng. Rapid screening of transformed data leaks with efficient algorithms and parallel computing. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, New York, NY, USA, 2015. ACM.

[34] Anna Cinzia Squicciarini, Smitha Sundareswaran, and Dan Lin. Preventing information leakage from indexing in the cloud. In *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*, pages 188–195. IEEE, 2010.

[35] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.

[36] Shun-Fa Yang, Wei-Yu Chen, and Yao-Tsung Wang. ICAS: An inter-VM IDS log cloud analysis system. In

Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on, pages 285–289, Sept 2011.

[37] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 199–208, New York, NY, USA, 2013. ACM.

[38] Eunjung Yoon and A Squicciarini. Toward detecting compromised mapreduce workers through log analysis. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 41–50, May 2014.

[39] Peisen Yuan, Chaofeng Sha, Xiaoling Wang, Bin Yang, Aoying Zhou, and Su Yang. XML structural similarity search using MapReduce. In *Web-Age Information Management, 11th International Conference*, volume 6184 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 2010.

[40] Chunwang Zhang, Ee-Chien Chang, and R.H.C. Yap. Tagged-mapreduce: A general framework for secure computing with mixed-sensitivity data on hybrid clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 31–40, May 2014.

[41] Weizhong Zhao, Huifang Ma, and Qing He. Parallel *k*-means clustering based on MapReduce. In *Cloud Computing, First International Conference, CloudCom 2009*, volume 5931 of *Lecture Notes in Computer Science*, pages 674–679. Springer, 2009.

[42] Li Zhuang, John Dunagan, Daniel R. Simon, Helen J. Wang, Ivan Osipkov, and J. Doug Tygar. Characterizing botnets from Email spam records. In *First USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '08*. USENIX Association, 2008.

[43] Mohsen Zohrevandi and Rida A. Bazzi. Auto-FBI: A user-friendly approach for secure access to sensitive content on the web. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 349–358, New York, NY, USA, 2013. ACM.

APPENDIX

A. PERFORMANCE OF A SINGLE HOST

To verify that one host alone cannot perform large-scale data leak detection, a single host version of the similarity-based detection algorithm was implemented and tested on one machine containing two quad-core 2.8 GHz Xeon processors and 8 gigabytes of RAM.

We tested the performance with different size of content and sensitive data and monitored the system. The performance is shown in Table 3.

This single machine-based detection system crashes with large content or sensitive data because of lacking sufficient memory. Thus, when the content or sensitive data are large, a single host is not capable of completing the detection due to memory limitation and thrashing. This experiment confirms the importance of our parallel data leak detection framework with MapReduce.

Content \ Sensitive	0.9 MB	1.4 MB	1.9 MB
588 MB	22.08	20.81	*
1229 MB	24.21	*	*
2355 MB	25.7	*	*
4710 MB	*	*	*

Table 3: Detection throughput (Mbps) on a single host with different content size and sensitive data size. * indicates that the system crashes before the detection is finished.