Detecting and Identifying System Changes in the Cloud via Discovery by Example

Hao Chen*, Sastry S. Duri†, Vasanth Bala†, Nilton T. Bila†, Canturk Isci† and Ayse K. Coskun*
*Department of Electrical and Computer Engineering
Boston University, Boston, MA, 02215
†IBM T J Watson Research Center, 1101 Kitchawan Rd., Yorktown Heights, NY, 10598

***Abstract*—Discovering and identifying system changes caused by events such as software installation and updates, configuration changes, and security patches are important functionalities for change management, security, compliance and problem diagnosis in emerging cloud platforms. Currently, most discovery tools use manually written rules, which require specific knowledge of software and systems. Approaches based on manually written rules are often fragile and require constant maintenance in this era of continuous integration. In this paper, we propose a novel "discovery by example" approach to autonomously search for and identify system changes. Our approach learns characteristic features of system changes automatically, without requiring any explicit rule definitions or specific knowledge of the underlying software or systems. In this approach, given a system change, our method searches a repository that contains previous stored system changes and returns those that are similar to it. We further explore the use of various forms of "fingerprints" to represent system changes efficiently and faithfully in a compact manner. We propose and evaluate two types of fingerprints: the "base-name fingerprint" and the "1-D histogram fingerprint". We show that both fingerprints exhibit different efficiency and accuracy trade-offs, and they can be effectively employed in different use cases. We evaluate the performance of our approach with both techniques and further present an application of it in system real-time streaming monitoring.**

I. INTRODUCTION

A typical data center hosts thousands of virtual machine (VM) instances. These instances evolve from the time they are booted. Sometimes, two instances booted from the same image evolve so differently that a software update to the first instance completes successfully but fails on the second one. What are the differences between these instances? How many other instances in the data center are similar to the second one? To answer above questions, today, we write custom scripts and execute them on the individual system to find the changes made to the system since it is booted. The sources of system changes include software installation, update, system reconfiguration and process execution, etc. Software installation is the most significant one among them.

To discover software installed in a system, state-of-the-art techniques use *rules* to check for the existence of certain files and their attributes. Some rules can be simple. For example, if there exists a file named $SIGFILESDKXA64\_500500.SYS2$ and its size is *100KB*, it means that there is a software called *IBM SDK 5.0* for the system *Linux AMD/EMT 64*. While other rules, particularly, those that determine whether a given software fix is applicable to a given system [1], could be quite complex and involve dozens of conditions to be checked.

Discovery rules, however, often fail. Consider the following use case. Suppose on our Linux machine we have the software package *IBM SDK 5.0* for the system *Linux AMD/EMT 64*. If we consult *National Vulnerability Database (NVD)* [2], we would find a vulnerability alert, *CVE-2012-4821*, against the version of *IBM Java* in this software package. There is a fix available for this vulnerability that requires us to install JDK. After the fix is installed, we find that the fix does not change the file $SIGFILESDKXA64\_500500.SYS2$ (which is used as a rule to discover the software *IBM SDK 5.0*) in any manner. For this reason, the simple rule described above fails to distinguish whether the system contains a vulnerable version or a fixed version of the *IBM SDK 5.0*.

In general, there are three major shortcomings of a rule-based discovery approach. First, it is fragile and highly system/software dependent. Take the software installation discovery as an example. All the rule-based systems require rules to be renewed when the software is updated or a fixpack is released. While this is doable, it is very inefficient, or even not practical, as the development of the fix and the writing of discovery rules are typically handled by different entities. Second, the rule-based discovery has poor usability. One has to learn a new rule language and be familiar with specific software components, in order to write good discovery rules. Third, the rule-based discovery approach is not suitable for unknown system changes. Before designing rules to discover a specific system change, the attributes of that system change should be first studied, which typically involves large amount of labor and time. Today, software and updates are released multiple times a week, and systems in cloud are changed nearly every minute. It has been impractical to evolve rules at such a rapid pace.

On the other hand, to identify differences between VM instances, or to search for specific system changes across a number of VMs, it is necessary to keep track of system evolution periodically. With thousands of systems in the cloud running at any given time, a quarter million features per system on average, and with hourly snapshots of the system, the size of the repository needed to keep track of all the changes is tremendous and approaches big data proportions. As most traditional techniques fail rapidly in dealing with such a large amount of data, novel big data solutions are required urgently.

In this paper, we present a novel method to discover system changes, the *discovery by example*, which eliminates the need to manually write rules for discovering systems, changes, or software components in the cloud. Our method learns characteristic features of a system change through a

set of its examples automatically. To collect the example, we compute the differences in system state before and after the system change happens. These recorded differences of the system state are further processed to produce a *fingerprint*, a compact representation that includes only features relevant for discovery, such as file base-names, using feature extraction techniques. We propose two types of fingerprints in this paper: the *base-name fingerprint* and the *1-D histogram fingerprint*.

While we specifically experiment with the state changes caused by software installation in this paper, our method is applicable to identification of arbitrary system changes. The ability to work with arbitrary changes is useful for detecting drifts in systems (that may be signs of vulnerability or inefficiency) by periodically scanning systems and computing their state differences. Experimental results show that our *fingerprint based discovery by example method* (a) is distortion resistant and not affected by the noise in the system change; (b) is fast, storage efficient and highly scalable, which is significant in the big data context; (c) can learn incrementally as more examples are provided, and does not require knowledge of file / system specifics or manual updates.

The rest of this paper starts with an overview of "learning by example" techniques, and the state-of-the-art techniques of system discovery. Section III describes the main idea and structure of our proposed "discovery by example" approach. Section IV introduces two types of fingerprints and their usages in "discovery by example". Section V first evaluates the results of "discovery by example" with two proposed fingerprints, and then introduces a case study of applying the approach in the real-time streaming system monitoring. Section VI concludes the paper and discusses the future work.

II. RELATED WORK

Discovery by example technique is widely used in the multimedia data analysis domain, such as voice recognition [12], face and object recognition ([6], [18]), image and audio search [14], etc. Shazam [17] is a music search service that allows users to search for music using audio samples. Both Picasa [16] and Google image search [5] allow users to search for photos that are similar to a given photo. Amazon Flow [10] discovers products utilizing object recognition techniques.

Today's system and software discovery techniques are mostly rule-based. Open Source Software (OSS) Discovery [3] is an open source tool that scans machines to identify any open source components installed on that system by either consulting registry in a system or checking for existence of files with specified properties. OpenIOC [4] is an open framework for sharing threat information. It uses elaborate rules to examine registry, file contents and metadata to determine whether a security vulnerability exists. BigFix [1] is a commercial offering that uses rules to scan systems and apply fixes automatically.

Some recent work investigates solving system problems using examples. In system performance monitoring, a "fingerprint", or a "signature", which is constructed by the statistical selection and summarization of hundreds of performance metrics, is used to represent the system state for automatic classification and identification of performance crises ([7], [8]).

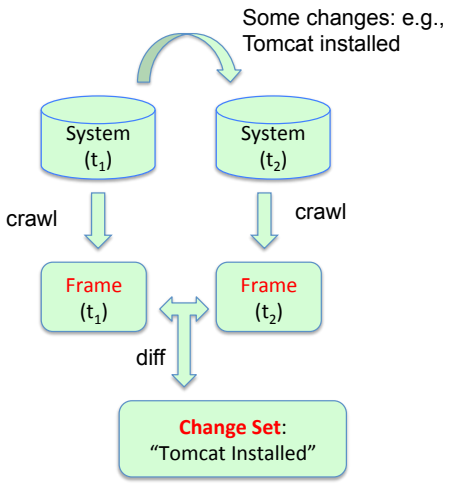

Fig. 1. The creation process of a change set.

Redstone et al. [13] propose to build an automated problem diagnosis system that collects problem symptoms, automatically searches databases of problem symptoms and fixes, and also allows ordinary users to contribute accurate problem reports in a structured manner. They propose the idea, however, without introducing specific design or evaluating performance in their work. Minersoft [9] scans of a collection of systems' file contents and metadata to build an inverted index, which can be used by users to find systems containing given software. Satyanarayanan et al. [15] examines the opportunities and challenges in interactively searching VM images in cloud environment and presents early evidence of its feasibility, by leveraging the early discard method [11].

To the best of our knowledge, our work is the first to design and apply a "discovery by example" approach in detecting and identifying system changes in the cloud context. We also design two novel fingerprints to represent the system changes in a condensed and efficient way. In addition, we propose a filter cascade structure applied with our "discovery by example" approach. Finally, a case study of leveraging "discovery by example" approach in the real-time streaming system monitoring is described in this work.

III. OUR APPROACH

The goal of our approach is to solve the following problems: (a) Given a change in the system state of an instance, detect similar system changes on the other instances in the cloud; (b) Identify unknown system changes using previously labeled system changes in the repository.

A system's state changes on a continuous basis. Some of the changes arise from clearly defined events such as software installation, application of security patches and configuration changes, while others are artifacts of system operations such as logs and system events. For the rest of the paper, we focus on system changes caused by software installation. Note that, however, the proposed method is applicable to a variety of state changes.

Our approach first records the system change as a *change set*, which contains all details of that state change. For discovery purposes, we find that a more compact representation

```
{
    os: {
        type: 'RHEL linux', distro: 'Red Hat', version: '4.2', ipaddr: '9.25.34.1', hostname: 'vm23.rescloud.ibm.com',
        mount-points: { '/dev/vda1' : 'ext3', '/dev/vda2': 'ext4' }, ...
    },
    file: {
        '/etc/hosts' : { permission: '-rw-r--r—', size: 236, user: 'root', group: 'wheel' },
        ... < one entry per file in the file system > ...
    },
    package: {
        tomcat6 : { version: '6.0.2', vendor: 'Apache', arch: 'x86_64' },
    ... < one entry per installed package >
    },
    process: {
        'httpd' : { pid: 23, exec: '/opt/apache/httpd', ports: [8080], open-files: ['/var/log/httpd/httpd.log', ...] },
    ... < one entry per running process >
    },
    config: {
        '/var/tomcat/web.xml' : {
            < contents of config file can also JSON-encoded. e.g.>
            Connector: { sslEnabled: true, maxPostSize: 2MB, port: 8080, URIEncoding: ISO-8859-1}
        },
        ... < one entry per config file (client-specified list) >
    }
}
```

Fig. 2. An example of the system state recorded in a frame.

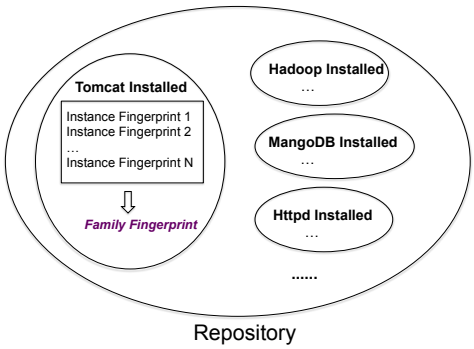

Fig. 3. The structure of the repository. Multiple instance fingerprints of the same event are grouped together as a family, and a family fingerprint is generated from instance fingerprints in the family, for each event. The family fingerprint is used to represent the event and distinguish the event from others.

that extracted from a change set, i.e., a *fingerprint*, is more useful and efficient. The fingerprint design is presented in detail in Section IV. Then the fingerprint is labeled by the event that causes the state change, and is stored in a *repository*. To solve problem (a), a designed filter cascade is applied on the fingerprint repository and splits the repository into two sets: the candidate set and the discard set. Fingerprints in the candidate set are considered similar to the given one. These fingerprints can be further used to solve problem (b).

Next, we describe the process used to capture change sets. After that, we introduce the concept of the *fingerprint*, and discuss how it is used in early discard, a technique we use to cascade the discovery process over a set of filters.

We use the following concepts in our discussion of the "discovery by example" technique:

System State consists of persistent state information such as configuration, disk, files, OS, and dynamic state such as processes, network connections.

Frame is a JSON representation of system state represented as a collection of following features: configuration, connection, disk, file, package, and process.

Feature is a JSON dictionary representing *attributes* of state entities. A file feature, for example, consists of the following attributes for a single file: path, name, size, access time, modification time, permissions, ownership, and type.

Change Set is the difference between two frames from the same system and contains following sections: *additions, deletions, modifications*, and *common*.

A. Change Set Creation

Figure 1 shows the process of creating a *change set*. In the example shown, the system change is caused by installing Tomcat server. First, the state of the system is collected into a frame $frame_1$. Then Tomcat is installed. Once the installation is complete, the system state is again collected into a new frame $frame_2$. Figure 2 shows an example of the frame. Then the difference of two frames, $frame_2$ - $frame_1$, is computed:

(1) If a feature is in $frame_2$ but not in $frame_1$, then it is added to *additions*;

(2) If a feature is in both frames, but their attributes differ, then it is added to *modification*;

(3) If a feature is in both frames, and attributes match, then it is added to *common*;

(4) If a feature is not in $frame_2$ but is in $frame_1$, then it is added to *deletions*.

Among all types of features, the file feature is the most significant one for identifying system changes, and especially for software installations. In this work, we mainly focus on using file features to discover and identify those changes, note that, however, other features can also be used. For example, it is possible to uniquely identify a software by the list of running processes. Moreover, integrating multiple types of features in discovery may even improve accuracy of the discovery, which is discussed in Section VI.

We use the *yum* utility to install software in this work[1], which automatically resolves dependencies for installing software. For this reason, the resulting change set includes file features from different sources: Tomcat server files, files modified during installation (e.g., */etc/passwd* by adding Tomcat users), temporary files created during installation, files belonging to software installed to satisfy dependency requirements, yum repository file updates, files created and modified by other activities not related to Tomcat installation, etc. Therefore, for a given Tomcat version on a specific system environment, the file features contributed by the Tomcat server remain the same. However, the file features in the change set vary from installation to installation depending on what other dependent software is installed by the yum utility and what other parallel activities are running during the installation process. Therefore, every change set consists of some variations that impede it of being identified easily.

B. Fingerprint

Directly utilizing the change set for discovery is not a good choice, due to the fact that a change set is a complete record of raw system changes. Thus, it contains information that is very specific to the system, and includes a lot of information that is not relevant for discovery purposes. Moreover, as the size of the change set is usually large, using the change set for discovery leads to low discovery speed and high storage costs. Therefore, we extract a subset of features and their attributes from a change set and create a highly condensed *instance fingerprint*. From a given change set, one could create

[1]Our approach can also work with software installed through other methods. It is not limited to yum.

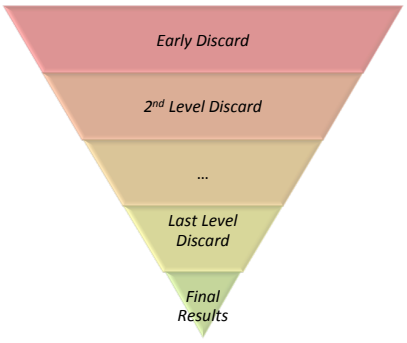

Fig. 4. The filter cascade of the "discovery by example" approach.

many different types of *instance fingerprints* by choosing different combinations of features and their attributes, and using different methods. In Section IV, different ways of generating *instance fingerprints* are presented.

All instance fingerprints are stored in a *repository*. A typical repository contains many change sets for a single system change event such as installing a Tomcat server. We group all the change sets of an event together as a family, and label the family by the name of event, such as "Tomcat Installation". Similar to change sets, instance fingerprints are grouped into families as well. Then we generate a *family fingerprint* for each family based on all the instance fingerprints in it. Depending on different algorithms used, a *family fingerprint* can be simply a set of all the instance fingerprints, or a collection of support vectors, parameters or decision rules trained from instance fingerprints. In this evaluation, a *family fingerprint* is simply represented as a set of all its instance fingerprints. Figure 3 shows the structure of the repository.

The repository is automatically maintained and updated in some fixed periods, e.g., in mid-night everyday, while the newly queried samples in that day are added as the training data for updates. In addition, an interface is also designed for maintainers to manually update the repository if necessary.

C. Framework of Discovery and Early Discard

We use a cascade of filters for discovery. Figure 4 shows the general idea behind this approach. A filter at the topmost layer is called as the *early discard filter*, which filters out as many dissimilar candidates as possible using simple and fast computations while guaranteeing to achieve a fairly low rate of incorrectly discarding the matching candidates. Candidates that pass through the early discard filter are sent to the following layers for further processing until either they are discarded or they pass through all the layers and are included in the final results.

To illustrate how filter cascade is used in our approach, we take an uncategorized change set C as an example. Without loss of generality, let us assume that the filter cascade consists of only two layers, i.e., the first filter, or the early discard filter, and the second filter. We extract two types of instance fingerprints from the change set, i.e. $type_1$, and $type_2$ for each filter layer, respectively. The process of finding change sets in the cloud that are similar to C and identifying C is as follows: The first filter examines the $type_1$ fingerprints for all the change sets in the repository. If a change set has a $type_1$ fingerprint that is considered similar to the $type_1$ fingerprint of C, then the change set is sent to the second filter. The second filter computes the similarity measure between $type_2$ fingerprints of those candidates sent to it and the $type_2$ fingerprint of C, and outputs those change sets that meet the similarity criterion. Note that each layer could use its own similarity criteria. The output after the second filter can then be used to identify C. For example, if the output contains "Tomcat Installation", then we recognize that C might be a change set of Tomcat installation on an instance.

A well designed fingerprint can improve both the accuracy and efficiency of the discovery significantly. For example, merely using the size of the change set as the fingerprint, while being simple and fast, does not identify change sets that vary due to the "noise", e.g., background running processes that are unrelated to the application installation. Hence, such a fingerprint is not noise resistant and robust, and will filter out matching candidates with a fairly high probability. In this work, we focus on designing and analyzing efficient fingerprints for the early discard filter. We also introduce ideas of designing good fingerprints for layers after early discard in Section VI.

IV. FINGERPRINT DESIGN AND DISCOVERY PROCESS

Designing a concise fingerprint that is also capable of capturing the essence of the change set is important for discovery, and especially for the early discard, which requires high processing efficiency. There are many ways of creating fingerprints depending on what features and attributes in the change sets are used and how they are represented. In this section, we first introduce two types of *instance fingerprints*, which have more compact representations than the corresponding change set. Then we present how these fingerprints are applied in our "discovery by example" approach with the filter cascade structure.

A. Base-name Instance Fingerprint

The first type of *instance fingerprint* is called the *base-name instance fingerprint*, which is a list of the base-names[2] of all added and modified file features in a change set. A list of base-names of file features has a strong distinguishable capacity, as the combinations of base-names caused by different system changes are mostly unique. In addition, using only the base-names of the files in a change set enables us to detect the change set no matter where in the system the corresponding files exist, and no matter in what status and authorities those files are set by users.

For a *base-name instance fingerprint* f^{bn}, we define its length $L_{f^{bn}}$, as the number of base-names in the fingerprint. Then for any two base-name instance fingerprints, f_1^{bn} and f_2^{bn}, the similarity score (α_1, α_2) between them is defined as the ratio of the number of common base-names in f_1^{bn} and f_2^{bn}, i.e., N_{com}, to the length of f_1^{bn} and f_2^{bn}, i.e., $L_{f_1^{bn}}$ and $L_{f_2^{bn}}$, respectively, i.e., $\alpha_1 = N_{com}/L_{f_1^{bn}}$ and $\alpha_2 = N_{com}/L_{f_2^{bn}}$. Based on the value of (α_1, α_2), there are four different relationship between f_1^{bn} and f_2^{bn}:

[2]Base-name is the name of the file, without its directory information.

(1) If $\alpha_1 \approx \alpha_2 \approx 1$, then f_1^{bn} is similar to f_2^{bn};

(2) $\alpha_1 \approx 1$ and $\alpha_1 >> \alpha_2$, then f_1^{bn} is contained by f_2^{bn};

(3) $\alpha_2 \approx 1$ and $\alpha_2 >> \alpha_1$, then f_2^{bn} is contained by f_1^{bn};

(4) Neither α_1 nor α_2 is close to 1, then f_1^{bn} and f_2^{bn} are not similar.

The utilization of these relationship in discovery and in the early discard process is discussed in Section IV-C in detail.

B. 1-D Histogram Instance Fingerprint

A change set sometimes contains thousands of file features, as a result, the base-name instance fingerprint, which consists of base-names of all the file features, is still not sufficiently compact for early discard. In addition, different change sets may share many common base-names that are not from core parts of the system changes (e.g., a large number of common temporary files created as part of the software installation by yum), which may cause the base-name fingerprint to be no longer distinguishable. Furthermore, more learning algorithms can be applied if a base-name fingerprint is transformed into a quantified fingerprint. For these reasons, we propose a quantified fingerprint, the *1-D histogram instance fingerprint*, i.e., f^{1D}.

The idea of the *1-D histogram instance fingerprint* is inspired by the image processing technique. In image processing and pattern recognition, a pixel-based image is typically represented by a histogram feature, e.g., local binary pattern, color histogram, etc, which is capable of capturing the main attributes of the image with only a few numbers, and can be efficiently processed by many learning algorithms. Similarly, we apply the histogram feature technique in our discovery of system changes. The idea is to build the histogram by using some hashing functions to convert the strings of base-names to integers in bin range. More specifically, in our implementation, the f^{1D} is generated in the following way:

Step 1: For each base-name in f^{bn}, we calculate the ASCII sum of its characters. In our implementation, we select the ASCII sum as the hash function to convert strings to integers, because ASCII sum is simple and fast. Experimental results have shown that ASCII sum is an efficient choice. In future, we plan to evaluate the use of different hash functions.

Step 2: Each base-name string has been converted to an integer after step 1, so now we have several numbers of integers. We generate a counting histogram of these integers. The range of each bin of the histogram is determined by the number of bins, i.e., N_{bins} that is selected. Based on our observations, most of these ASCII sum integers are ranged in [200, 2000], thus, given the N_{bins}, the bin range in our case is designed as (0, 200, $200 + \frac{2000-200}{N_{bins}-1}$, $200 + 2*\frac{2000-200}{N_{bins}-1}$, ... , 2000 - $\frac{2000-200}{N_{bins}-1}$, 2000, ∞). In the counting histogram, the number of ASCII sum integers that falls in each bin, i.e., $C_i,\ i = 1, 2, ...N_{bins}$ is calculated;

Step 3: Finally we normalize the histogram by calculating $C_i^{norm} = C_i \ / \sum_{i=1}^{N_{bins}} C_i,\ i = 1, 2, ...N_{bins}$, so we have $\sum_{i=1}^{N_{bins}} C_i^{norm} = 1$. The histogram is normalized so that the discovery result is not affected by the length of the base-name list.

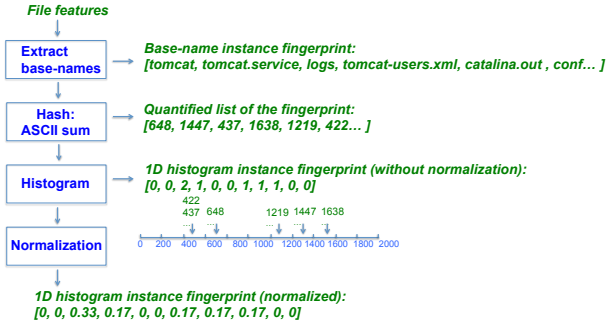

Fig. 5. The flow chart of the 1-D histogram instance fingerprint generation.

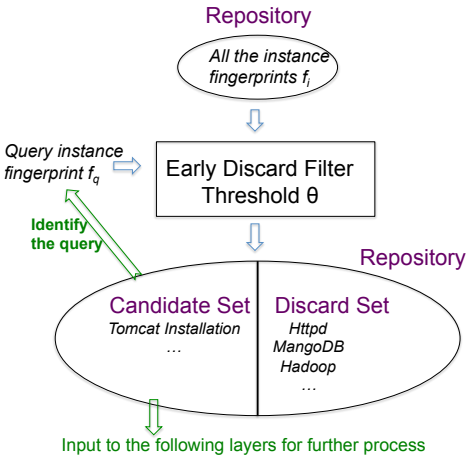

Fig. 6. The discovery process by using fingerprints.

The length of the 1-D histogram instance fingerprint is N_{bins}, which is much smaller than the length of f^{bn}. Figure 5 shows the generation process of the 1-D histogram instance fingerprint.

As the 1-D histogram instance fingerprint is a quantified and highly condensed feature vector, many learning algorithms and metrics can be directly applied on it. For example, the similarity of two 1-D histogram instance fingerprints can be measured by a distance metric. The smaller the distance is, the more similar the fingerprints are. There are various of distance metrics. In this work, we use the simple yet effective Euclidean distance. Namely, given two 1-D histogram instance fingerprints, f_1^{1D} and f_2^{1D}, the distance $d_{1,2}$ between them is calculated as $d_{1,2} = ||f_1^{1D} - f_2^{1D}||_2$. As both f_1^{1D} and f_2^{1D} are normalized, the maximal value of the distance is $\sqrt{2}$. We further normalize $d_{1,2}$ as the percentage of the maximal value, i.e., $d_{1,2} = \frac{||f_1^{1D} - f_2^{1D}||_2}{\sqrt{2}} * 100\%$, for convenience of making comparisons.

C. Discovery Process with Fingerprints

In this section, we specifically present how the "discovery by example" and the early discard work with designed fingerprints. We first study the discovery by using the base-name instance fingerprint. The fingerprint of the query change set is first generated as f_q^{bn}. Then f_q^{bn} is input into the early

discard filter and compared with all the base-name instance fingerprints $f_i^{bn},\ i = 1, 2, 3...$, in the repository. Based on the results of comparisons and the designed "filtering policy", all the system change events, i.e., all the families, in the repository are divided into two sets, a *candidate set* and a *discard set*. A *candidate set* includes families that have *at least one instance fingerprint* that passes the filter (i.e., satisfies the "filtering policy"), while families in the *discard set* have none of their instance fingerprints satisfying the "filtering policy". The "filtering policy" of the early discard layer in this case is designed based on the similarity score introduced in Section IV-A, and a similarity threshold, $\theta^{bn} \leq 1$. For each instance fingerprint pair (f_q^{bn}, f_i^{bn}) and corresponding similarity score (α_q, α_i), if both $\alpha_q \geq \theta^{bn}$ and $\alpha_i \geq \theta^{bn}$, then f_i^{bn} is *similar* to f_q^{bn} and f_i^{bn} passes the filter, otherwise it is filtered out. Families in the candidate set are considered similar to the query sample by the early discard layer, and provide clues of identifying the query. For example, if the "Tomcat installation" family is in the candidate set, then the query sample may be a change caused by Tomcat installation. These families are further processed by the following layers in the discovery filter cascade. Figure 6 shows this process of discovery.

The process of discovery by the 1-D histogram instance fingerprint is similar to that of the base-name instance fingerprint, but with a different "filtering policy". The "filtering policy" in this case is defined based on the distance metric. The distance between the query 1-D histogram fingerprint f_q^{1D} and the 1-D histogram instance fingerprint in the repository, $f_i^{1D},\ i = 1, 2, 3...$, is calculated, and denoted as $d_{q,i}$, which has been introduced in Section IV-B. A distance threshold, $\theta^{1D} \leq 1$ is also defined, recalling that the distance is normalized by the maximal value. If $d_{q,i} < \theta^{1D}$, then f_i^{1D} is considered similar to f_q^{1D} on the early discard layer and passes the filter, otherwise it does not satisfy the "filtering policy" and is filtered out. The rest process of the 1-D histogram instance fingerprint is the same as that of the discovery using base-name instance fingerprint.

V. EXPERIMENTAL RESULTS

In this section, we first present how the repository is constructed and how the test benchmark data is collected. Then we evaluate results of the "discovery by example" approach using both the base-name fingerprint and the 1-D histogram fingerprint. Finally, a case study of leveraging "discovery by example" technique in real-time streaming system monitoring is discussed.

A. Data Collection and the Test Benchmark

We generate the example repository for our experiments by randomly selecting 161 software packages from the Linux yum repository, installing them on an Amazon Web Service (AWS) EC2 Fedora-19 Micro instance by using the yum utility, and recording the system changes during each of the installation process. In installation, software package dependencies are resolved and installed as well. Different software packages usually share common dependencies, so some dependencies that are needed by the current software may have already been installed during previous installations of other software. Thus, when we install a batch of software, changing the order of the software package installation affects files added or modified into each change set. Furthermore, there might be changes made from other unrelated activities happening in parallel of the installation. All these conditions introduce variations in the change set corresponding to a software installation. In order to capture this variation in our experiments, each software is installed three different times when we generate the repository, with various dependencies and unrelated activities during each of them. Then we randomly select 89 software from the 161 available packages in repository to create our test benchmark. We install each of them once and record the change set.

We use the *percentage of false discard* and the *average percentage of total discard* to evaluate the accuracy and efficiency of our approach. The *percentage of false discard*, i.e.,η_f is defined as the percentage of 89 tests whose target candidates in the repository are put in the discard set. As these 89 test software are selected from 161 software in repository, each of them has one target candidate. High η_f represents high false discard rate, thus, low accuracy. On the other hand, an over-permissive discard filter could include too many candidates in the candidate set, and thus provide low value for early discard. Therefore, we define the *average percentage of total discard*, i.e., η_t to measure the efficacy of the filter. For each test i, the percentage of total discard, i.e., β_i is calculated as the size of the discard set, i.e., S_i^D divided by the total number of the events in the repository (in our case equals to 161), i.e., $\beta_i = S_i^D/161 * 100\%$. Then η_t is computed as the average value of β_i across 89 tests, i.e. $\eta_t = \sum_{i=1}^{89} \beta_i/89$. In addition, we also measure the size of the repository and the average processing time of the test, to help evaluate the efficiency of our approach.

B. Early Discard with Base-name Fingerprint

We evaluate the performance of early discard on our designed test benchmark, using the base-name instance fingerprint, with various *similarity thresholds* θ^{bn}. The results of the *percentage of false discard*, the *average percentage of total discard* and the average processing time of the test are shown in Figure 7(a) to Figure 7(c), respectively. From Figure 7(a) and Figure 7(b) we can see that when θ^{bn} increases, both the *percentage of false discard* and the *average percentage of total discard* increase. This is as expected, as a larger *similarity threshold* requires candidates to have more matching base-names to pass the filter, and as a result, more candidates are discarded. Figure 7(c) shows that the average processing time of the test keeps around 33 milliseconds, and does not vary much with different θ^{bn}. While determining a good θ^{bn}, a low *percentage of false discard* is first required, and then a high *average percentage of total discard* is expected, as a very large candidate set after the filter leads to too many candidates being sent to the next layer, and therefore causes low efficiency. If an tolerable *percentage of false discard* is 10%, then both $\theta^{bn} = 50\%$ and $\theta^{bn} = 60\%$ satisfy this requirement from the figure, and with the *average percentage of total discard* as 55.3% and 74.5%, respectively. The *similarity threshold*, furthermore, can be automatically trained and updated, based on different accuracy and performance requirements, following the updates of the repository, so that the threshold is able to be always well tuned, to suit for different requirements and purposes in general.

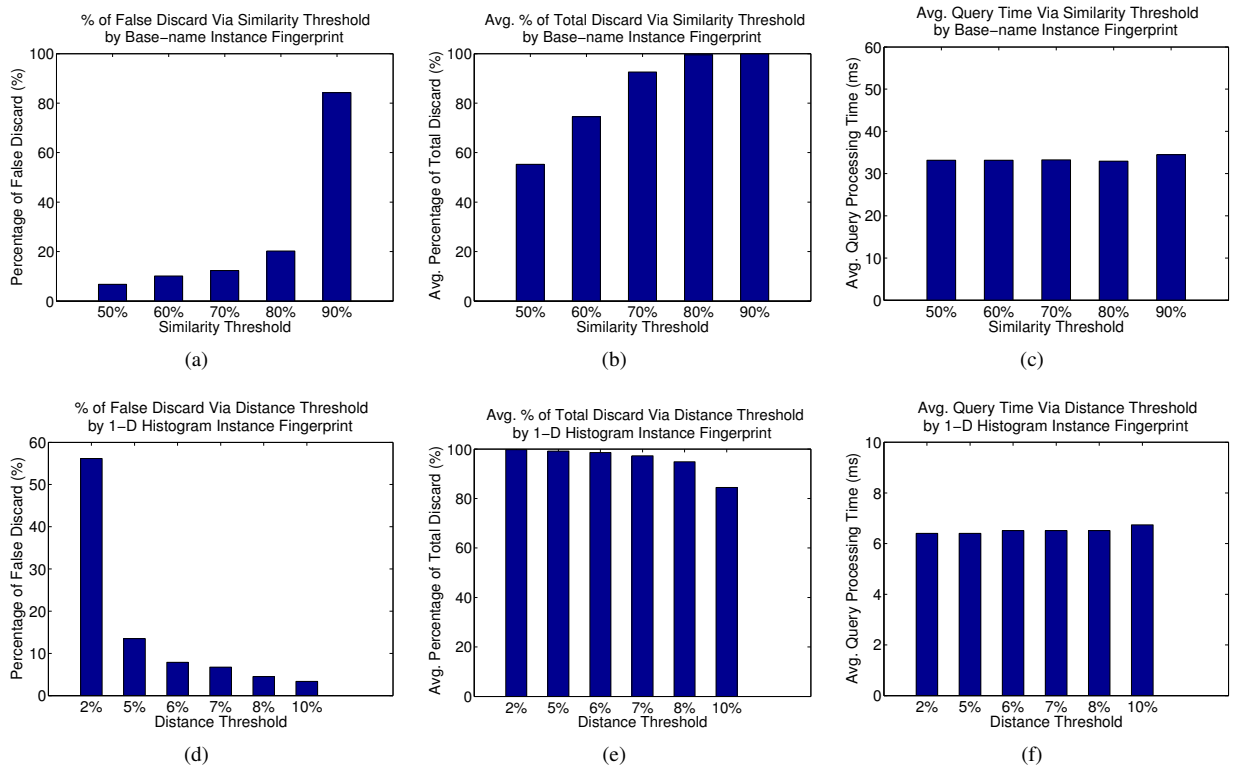

Fig. 7. The early discard performance by using the base-name instance fingerprint (a, b, c), via different similarity thresholds, and by using the 1-D histogram instance fingerprint (d, e, f), via different distance thresholds, (a) and (d) are results of the percentage of false discard. (b) and (e) are results of the average percentage of total discard, (c) and (f) are the average query processing time. The results are measured based on 89 test software installations.

In addition, we measure the size of the repository (161 software with each installed by three times, thus, in total 483 samples). The storage size of using the base-name instance fingerprint is 11MB, which is 28 times smaller than saving all the *change sets*. Overall, the results show that when applying the base-name fingerprint, an early discard filter can discard more than half of the total candidates, while at the same time guaranteeing that the probability of the false discard is less than 10%. Also, the processing speed is fast and the storage size of the base-name fingerprint is small.

C. Early Discard with 1-D Histogram Fingerprint

While using the 1-D histogram fingerprint to represent the change set, the histogram bin number (i.e., the dimension of the fingerprint) not only affects the accuracy of discovery, but also affects the size of repository and the query processing speed. Hence, we test the 1-D histogram fingerprint approach with different selections of the histogram bin number. Figure 8 shows the results of the *percentage of false discard* (Fig. 8(a)), the *average percentage of total discard* (Fig. 8(b)), and the average query processing time (Fig. 8(c)) of the 89 software installation tests, at a fixed *distance threshold* $\theta^{1D} = 5\%$. The size of the repository of storing 483 1-D histogram instance fingerprints is shown in Figure 8(d).

Results show that as the number of bins is increased, both the *percentage of false discard* and the *average percentage of total discard* first increase rapidly, and then saturate after certain points. The increase of the *average percentage of total discard* is close to a step function, which means a very small

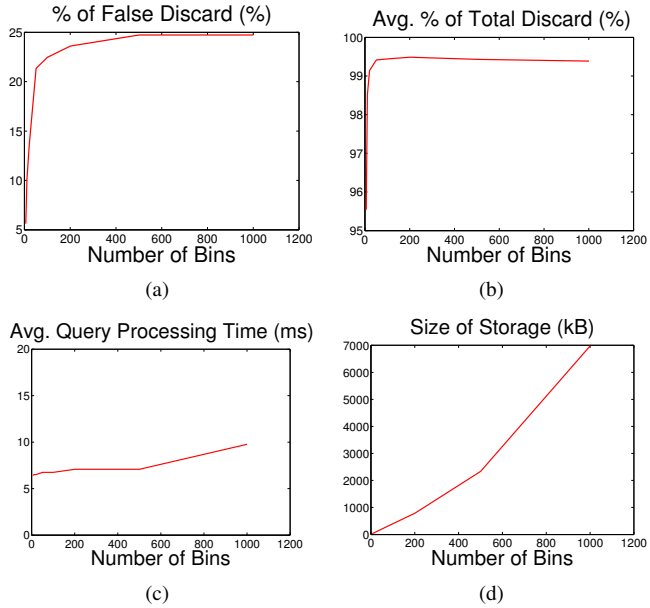

Fig. 8. The early discard performance by using the 1-D histogram instance fingerprint, via different number of bins used in histogram generation. (a) is the percentage of false discard, (b) is the average percentage of total discard, (c) is the average query processing time and (d) is the size of the storage of all the fingerprints. The results are measured based on 89 test software installations. The distance threshold used here is 5%.

number of bins can achieve fairly high *average percentage of total discard*, e.g., using 10 bins leads to an average percentage of total discard larger than 95%. In addition, both the average

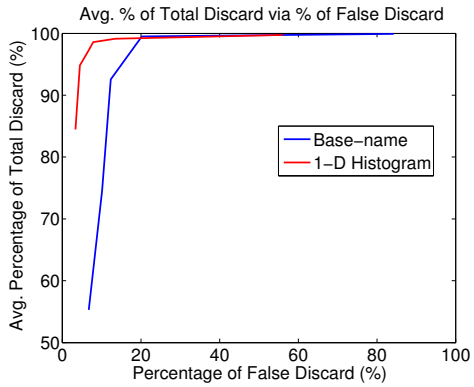

Fig. 9. The curve of the average percentage of the total discard via the percentage of false discard, by using the base-name fingerprint and the 1-D histogram fingerprint.

query processing time and the size of the storage of the 1-D histogram instance fingerprints increase accompanied with the growth of the number of histogram bins used. The average query processing time increases slowly and is not sensitive to the number of bins. However, the size of the storage increases notably at a linear trend with a steep slope. Overall from Figure 8, we conclude that neither a small nor a large number of bins is the good choice. If the number of bins is too small and close to 0, it leads to a low *average percentage of total discard* that is close to 0 and the system is useless; if the number of bins is large, the *average percentage of total discard* is not improved much, while the *percentage of false discard*, the average query processing time and the size of the storage required are all increased. Based on observations, a good choice of the number of bins used is estimated to be between 10-50.

With the selected number of bins as 20, then we test the performance of using the 1-D histogram fingerprint under different selections of the *distance threshold* θ^{1D}. We again query the fingerprints of 89 test software in our test benchmark and measure the *percentage of false discard*, the *average percentage of the total discard*, and the average processing time of the test, shown in Figure 7(d) to Figure 7(f). Figure 7(d) and Figure 7(e) show that when θ^{1D} increases, both the *percentage of false discard* and the *average percentage of the total discard* decrease. This is as expected, as a larger *distance threshold* allows more candidates with larger distances to the query fingerprint to pass the filter, and as a result, leads to fewer candidates being discarded. Figure 7(f) shows that the average query processing time by using the 1-D histogram fingerprint keeps around 6.7 milliseconds, and does not vary much with different θ^{1D}. Similar to the *similarity threshold*, the *distance threshold* can be trained and updated automatically. In addition, the size of the repository by using the 1-D histogram fingerprint is 88 KB, which is 3500 times smaller than saving all the *change sets*. Overall, the results show that when applying the 1-D histogram fingerprint, an early discard filter can discard more than 90% of the total candidates while at the same time guaranteeing that the probability of the false discard is less than 5%, which is fairly efficient. Also the processing speed is very fast and the storage size of the 1-D histogram fingerprint is tiny.

Finally, we compare the performance of the early discard

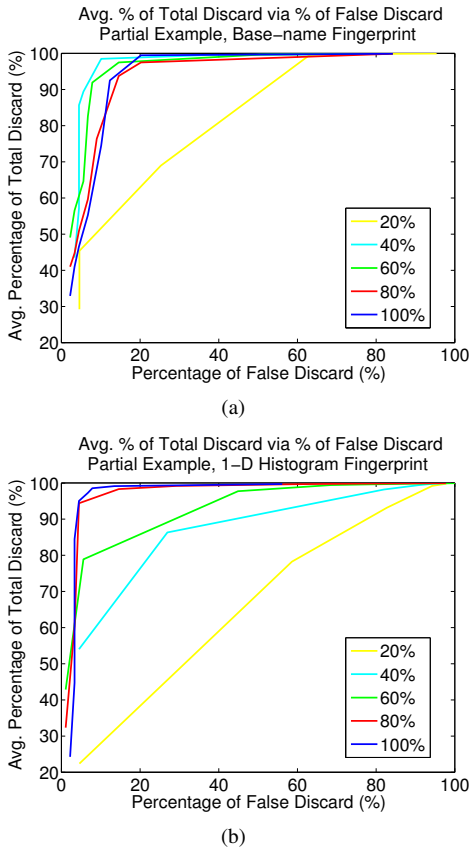

Fig. 10. The curve of the average percentage of the total discard, via the percentage of false discard, under different partial values (20% -100%). (a) is of the base-name fingerprint and (b) is of the 1-D histogram fingerprint.

by using the base-name fingerprint with that of using the 1-D histogram fingerprint. Figure 9 presents the curve of the *average percentage of total discard* versus the *percentage of false discard* for both types of fingerprints. For an early discard filter, it is desirable to have a small *percentage of false discard* and a large *average percentage of total discard*. From Figure 9 we can see that the performance of using the 1-D histogram fingerprint outperforms that of using the base-name fingerprint. For example, when we keep the *percentage of false discard* lower than 7% in both cases, using the 1-D histogram fingerprint provides the *average percentage of total discard* up to 97.3%, while that of using the base-name fingerprint is only up to 55.3%, which shows that using the 1-D histogram fingerprint is much more efficient. Furthermore, the 1-D histogram fingerprint is more condensed than the base-name fingerprint, with the size of storage of 483 instance fingerprints as 88KB versus 11MB, which is around 125 times saving. The query processing speed of using the 1-D histogram fingerprint is also 5 times faster than using the base-name fingerprint (6.7 ms versus 33 ms). Thus, to sum up, using the 1-D histogram fingerprint in discovery achieves both higher efficiency and higher accuracy.

D. A Case Study: Discovery by Example in Real-Time Streaming System Monitoring

Today, we have islands of information regarding system instances, vulnerabilities in databases like *National Vulnerability Database*, and fixes to these vulnerabilities in providers'

repositories, etc. If we close the loop between these islands, we could then make running infrastructure more secure. As part of this closed loop, we need to have the capability to block unwanted changes propagating through running infrastructures. This requires us to monitor changes happening in systems in real time, detect and stop anomalous system changes. In order to do this, we need to be capable of discovering system changes and events with only partial information of them, as it might be too late to avoid vulnerabilities and malicious changes if we wait for the anomalous changes being completed and the full change set being created.

In this case study, we evaluate the performance of our proposed fingerprint based "discovery by example" technique in real-time streaming system monitoring. More specifically, we want to see that given *partial change sets for query*, whether our approach is still able to efficiently search for similar candidates throughout the repository and identify the query example. We apply both the base-name instance fingerprint and the 1-D histogram instance fingerprint representations, and compare their performance. The query examples in this case are *partial* change sets[3] of 89 software installations in our test benchmark. We evaluate the results of different *partial values*, from 20% to 100%. 20% means that we only have a small part (20%) of the full change set, and 100% represents the full change set.

Figure 10(a) and Figure 10(b) present the curves of the *average percentage of total discard* versus the *percentage of false discard*, with each curve representing a *partial value* (20% -100%), by using the base-name fingerprint and the 1-D histogram fingerprint respectively. These figures show that when the *partial value* is small (e.g., 20%), neither two types of fingerprints provides good performance in discovery. This is as expected, as only having such tiny information is not sufficient for identifying the example. When the *partial value* increases, the performance of the discovery is getting improved.

The curve of the base-name fingerprint saturates fast - the performance of the 40% *partial value* has already been close to that of using the full query change set (100% *partial value*), which shows that the discovery of using the base-name fingerprint is not very sensitive to the *partial value*, as long as the *partial value* is not too small (less than 40%). Having a large *partial* value, in addition, sometimes may involve larger noise that affects the discovery, thus even leads to a decrease of the discovery performance. For the 1-D histogram fingerprint, the performance of discovery keeps improving following the increase of the *partial value*, which means that the 1-D histogram fingerprint is sensitive to the *partial value*. Comparing these two figures, we can further see that when the *partial value* is large (80% and 100%), using the 1-D histogram fingerprint has a better discovery performance, and when the *partial value* is small (40% and 60%), using the base-name fingerprint outperforms using the 1-D histogram fingerprint. Therefore, in the real-time streaming system monitoring, applying these two types of fingerprints in a complementary way can improve the overall performance: when the query is a small piece of the full change set, the base-name fingerprint is applied for discovery, and when the *partial value* is getting larger, we switch to apply the 1-D histogram fingerprint.

Overall, the results show that our "discovery by example" approach with proposed fingerprints can still be highly efficient and accurate, when only partial query samples are accessible. These results present the great potential of our approach being applied in a real-time streaming system monitoring context. Meanwhile, they also help demonstrate the robustness of the approach.

VI. CONCLUSION AND FUTURE WORK

In this paper we have proposed applying the "discovery by example" approach to detect and identify changes to system instances in the cloud. We introduced a filter cascade discovery structure and specifically studied the early discard filter. We have also proposed two types of highly condensed fingerprints (the *base-name fingerprint* and the *1-D histogram fingerprint*) to represent the system changes in early discard. Experimental results show that by applying our fingerprints, the early discard filter can discard up to 90% of unrelated candidates while guaranteeing that the probability of the false discard is less than 5%. In addition, the processing speed is less than 10 milliseconds for each query and the storage space required of saving examples is reduced by up to 3500 times. Further results in the case study show that our proposed approach can still keep high efficiency and accuracy while the query samples are partial, and thus is suitable for real-time streaming system monitoring. In particular, we have shown that our condensed 1-D histogram can perform better for full fingerprints, while the base-name fingerprints are more suitable for realtime, streaming detection. Overall, our approach can automatically detect and identify system changes from examples. It does not require any specific knowledge or manual processing. It is distortion resistant, fast, storage efficient, and can be used to identify various forms of system changes.

Our ongoing work includes the following. First, both fingerprints that we have designed so far take only base-names of file features into consideration. We believe that a fingerprint that takes into account other file attributes may have more discriminating power, which would be suitable for filters after the early discard. To incorporate additional attributes, *multi-dimensional histogram fingerprints* can be constructed, with each dimension representing one type of metadata. Also, while current work focuses on file features, we believe this approach can easily accommodate other features, such as processes, configuration, connections, and their relationship. Though the processing speed and the storage efficiency may decrease by taking all of them into account, it is still suitable for filters after the early discard, as which are supposed to take more details into account, and be more tolerable for the processing speed and the storage size than the early discard filter.

Second, we are testing the effectiveness of our approach on other use cases besides discovering software installations, such as discovering software updates and system reconfigurations. For example, our early results show that the fingerprint based "discovery by example" approach is capable of distinguishing the same software but with different versions, and as a result, software updates can be accurately discovered.

[3]Features in the partial change set are not randomly picked out from the original full change set, instead, they are continuous in the time sequence, and together form as a piece of the original full change set. They are recorded during a partial time period of the full application installation, namely, they are generated in a real-time streaming way.

Third, in this work, detection of similar fingerprints and identification of the query are based on comparing the query fingerprint with all instance fingerprints in the repository. We would like to explore whether we could synthesize a more efficient *family fingerprint* rather than a simple set of the instance fingerprints, so the processing speed and the storage efficiency can be further improved. In future, we plan to explore more machine learning algorithms such as SVM, decision trees and neural networks for the fingerprint classification and detection.

REFERENCES

[1] Endpoint manager relevance language guide. *http://pic.dhe.ibm.com/infocenter/tivihelp/v26r1/topic/com.ibm.tem.doc_8.2/Relevance_Guide_PDF.pdf*.

[2] National vulnerability database. *http://nvd.nist.gov/*.

[3] Open source software discovery. *http://ossdiscovery.sourceforge.net*.

[4] Openioc. *http://www.openioc.org/*.

[5] Search by image. *http://www.google.com/insidesearch/features/images/searchbyimage.html*.

[6] Serge Belongie, Jitendra Malik, and Jan Puzicha. Shape matching and object recognition using shape contexts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(4):509–522, 2002.

[7] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B Woodard, and Hans Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124. ACM, 2010.

[8] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 105–118. ACM, 2005.

[9] Marios D Dikaiakos, Asterios Katsifodimos, and George Pallis. Minersoft: Software retrieval in grid and cloud computing infrastructures. *ACM Transactions on Internet Technology (TOIT)*, 12(1):2, 2012.

[10] Geoffrey A. Fowler. One-minute review: Amazons flow image recognition beats barcode scans. *The Wall Street Journal, http://blogs.wsj.com/digits/2014/02/05/one-minute-review-amazons-flow-image-recognition-beats-barcode-scans/?mod=WSJBlog*, 2014.

[11] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, Mahadev Satyanarayanan, Gregory R Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *FAST*, volume 4, pages 73–86, 2004.

[12] Frederick Jelinek. *Statistical methods for speech recognition*. MIT press, 1997.

[13] Joshua Redstone, Michael M Swift, and Brian N Bershad. Using computers to diagnose computer problems. In *HotOS*, pages 86–91, 2003.

[14] Yong Rui, Thomas S Huang, and Shih-Fu Chang. Image retrieval: Current techniques, promising directions, and open issues. *Journal of visual communication and image representation*, 10(1):39–62, 1999.

[15] Mahadev Satyanarayanan, Wolfgang Richter, Glenn Ammons, Jan Harkes, and Adam Goode. The case for content search of vm clouds. In *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, pages 382–387, 2010.

[16] Steve Schwartz. *Organizing and Editing Your Photos with Picasa: Visual QuickProject Guide*. Peachpit Press, 2005.

[17] Avery Wang. The shazam music recognition service. *Communications of the ACM*, 49(8):44–48, 2006.

[18] Wenyi Zhao, Rama Chellappa, P Jonathon Phillips, and Azriel Rosenfeld. Face recognition: A literature survey. *ACM Computing Surveys (CSUR)*, 35(4):399–458, 2003.