DeltaSherlock: Identifying Changes in the Cloud

Ata Turk, Hao Chen, Anthony Byrne, John Knollmeyer, Sastry S. Duri, Canturk Isci, Ayse K. Coskun

Abstract—To track security and compliance requirements and perform problem diagnosis, administrators of cloud computing systems need to monitor significant system changes occurring on the set of cloud instances under their supervision. Considering the large number of instances (virtual machines, containers) possibly operating under multiple configurations, this is a difficult-to-track process. Standard solutions to this problem rely on manually-created rules to identify changes. These techniques suffer from a limited scope, rely on domain expertise, and are time-consuming and error-prone. Recently, more streamlined approaches that automatically determine the type of individual system changes have been proposed, but these techniques assume that system states right before and after each individual change can be captured, a rather difficult requirement to enforce in real world usage. This paper proposes DeltaSherlock, a practical system change discovery framework that can capture system states on-demand and detect multiple system changes between them. We evaluate DeltaSherlock over 25,000 system changes caused by software installations collected from virtual machines (VMs) deployed over a commercial cloud. DeltaSherlock can accurately identify multiple software installations with 96.8% accuracy when supplied with a non-overlapping record of system changes and with 77.8% accuracy when supplied with random irregular observations possibly containing overlapping or incomplete changes.

Index Terms—change discovery, multi-label classification, cloud operations management.

1 Introduction

Cloud computing, with its promise of efficient and always available computing, has observed widespread adoption in IT solutions and enterprise applications. State-of-the-art cloud deployments operated by large organizations frequently scale up to thousands of cloud instances (i.e., VMs, containers, etc.) [1]. Complications in managing such large scale deployments are exacerbated by the widespread adoption of DevOps methodologies, agile development, and Continuous Integration / Continuous Delivery practices, which led to a drastic increase in the rate with which the code and applications in these systems are updated. A main management challenge IT administrators are facing is tracking and understanding these fast-paced changes for various purposes such as compliance, security, or configuration drift.

Standard solutions for system monitoring and change discovery involve rule-based mechanisms that search for specific files or features to indicate that a certain change has been made to the system. System specialists design these rules based on information obtained from libraries and repositories such as National Vulnerability Database (NVD) [2] and Open Source Software (OSS) [3]. These rule-based solutions require heavy involvement of expert administrators. Efficient rules can only be designed with a good understanding of the software and systems. Such solutions have limited scope, are rather fragile, and are time-consuming to produce. A tiny modification in the changes may easily bypass the pre-designed rules. Today's software and packages are typically released multiple times a week if not multiple times a day. When user changes on instances are mixed with these updates, each instance in the cloud can evolve differently. System administrators face difficulties in constantly maintaining and updating these rules to keep up with the fast cycle of the software and system changes. Overall, rule-based approaches are inefficient in cloud setups and automated solutions are needed.

Recently, automated approaches for system change discovery that create a "fingerprint" from a system change and compare it to a knowledge base of known changes have been proposed [4], [5]. This approach allows a general solution for the change discovery problem and eliminates the manual case-by-case analysis process. However, it has the limitation of assuming that system change events occur one at a time and it is possible to capture the changes observed after each event.

In this paper, we advance the state-of-the-art by proposing a mechanism that can identify multiple system changes caused by software installations that occur between two system observations. Our contributions can be listed as follows: (i) we propose a multi-label classification approach that automatically and efficiently identifies multiple changes observed in system state, (ii) we utilize a set of features that can be extracted from file system structures to provide short and informative representation of the impact of software installations on systems, (iii) we evaluate effectiveness of proposed approaches over 25,000 system changes collected from VMs deployed on a commercial cloud, (iv) finally, we propose DeltaSherlock, a cloud discovery service that uses one-way fingerprints and machine learning methodologies to analyze system changes.

- *Ata Turk, Hao Chen, Anthony Byrne, John Knollmeyer, and Ayse K. Coskun are with ECE Department, Boston University. E-mail: {ataturk, haoc, abyrne19, jknollm, acoskun}@bu.edu*
- *Sastry S. Duri and Canturk Isci are with IBM T. J. Watson Research Center. E-mail: {sastry, canturk}@us.ibm.com*

- *Anthony Byrne and John Knollmeyer were partially funded by Boston University's Undergraduate Research Opportunities Program.*

The rest of the paper is organized as follows. Section 2 presents the relevant literature. The proposed multi-change discovery process is presented in Section 3. In Section 4, we describe our experimental framework and in Section 5, we evaluate proposed approaches. Section 6 introduces proposed change discovery cloud service, and we conclude in Section 7.

2 Related Work

System changes in the cloud today are mostly discovered and identified with rule-based approaches: system experts design lists of rules for change identification and discovery, which check for the existence of certain files and indicated properties, such as sizes of files, some specific contents of configuration files, etc. [6], [7]. Package managers and history logs are also utilized for monitoring system evolutions [2], [3].

Even though well-designed rules and careful manual checks can be used in accurate identification of system changes, maintaining such rules is very difficult in today's short package release cycles and large-scale system deployments. Well-designed rules require good understanding and specific knowledge of systems and packages, which can only be delivered by system experts, indicating substantial amount of manual effort and cost. Furthermore, rules are fragile and have poor re-usability. A rule for discovery of an old package release can easily fail to discover a newer release. Most software packages are released multiple times a week. Hence rules require constant maintenance and updates. Finally, different changes generally require specific distinguishable rules for discovery, which is inefficient for cloud deployments where thousands of different system changes happen simultaneously.

In addition to the rule-based approaches, several prior papers have studied system problem detection and diagnosis based on comparison with existing examples. Registry entries and system event logs have been used in troubleshooting methods that identify problems on a given system [8]. EnCore [9] learns configuration rules from a given set of sample configurations, and then automatically detects software misconfigurations. Minersoft builds an inverted index file-tree structure using file metadata, and uses the file-tree to discover the software [10]. The "fingerprint" or the "signature" concept has been proposed as an abstraction of the raw data, which provides an interface for the application of statistical or machine learning techniques in identifying system state or compliance. Most of these fingerprinting methodologies are based on system performance metrics [11], [12].

Recently, the concept of "discovery by example" for system change discovery have been proposed [4], [13], [14]. The main philosophy behind this approach is to record all the system metadata that is modified during the changes caused by the system event, generate fingerprints from them and apply machine learning algorithms to train models for discovery. Comparing with rule-based approaches, "discovery by example" eliminates the requirement of manual or expert input. The models can be iteratively updated with newly collected data to automatically keep pace with software and systems updates. Furthermore, since "discovery by examples" automatically extracts abstracted semantics using the whole changed metadata set, it avoids sticking to any piece of specific feature, which makes this approach more robust to slight tweaks compared to rule-based approaches. However, these approaches simply assume that it is possible to perfectly single out and capture the changes associated with each system event. In a more realistic cloud setting, changes obtainable/observable may represent multiple or partial (incomplete) events. In this paper we propose multi-label discovery approaches for such cases. Our solution can be used to answer questions such as: (1) how many system events happen during a given period; (2) what these events are; (3) whether a specific event happened during the given period; and (4) whether there are any relations among those events.

3 Multiple Change Discovery

Our goal in this work is to identify multiple system events occurring in a given time interval. It is common to observe multiple system events that occur in close proximity in time, e.g., two packages installed together, software updates are deployed while user is configuring the OS, etc. System changes observed in such cases represent a mixed accumulation of the changes imposed by the multiple events. Discerning which change is associated with which event is challenging. Using system changes to detect system events becomes harder in these scenarios. In this work we address this harder problem.

Our approach is depicted in Fig. 1. We assume that a *changeset* that contains all system changes observed by the system during a given period is provided, and identification of events that cause these changes is requested. We tackle the multiple system event identification problem by first creating "smart" and "compact" features (we call these feature sets *fingerprints*) representing application nature from observed system changes, and then using these fingerprints to train multi-label classification machine learning models. Such models can identify multiple system events using the fingerprints created from (accumulated) changes observed during known (multiple) events. The trained models are then used in predicting the events that caused changes observed in real-life systems.

System changes are caused by a variety of system events, such as software, application and package installations, updates, system configurations, process executions and other user operations. In this study we focus on system changes created by multiple application installations, as application installations are known as one of the most significant factors leading to notable system changes [15]. However, we note that proposed

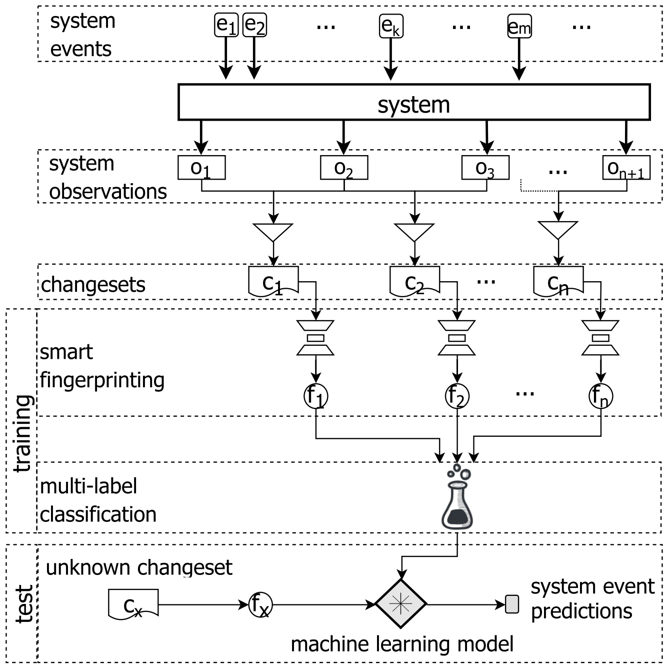


Fig. 1. Multiple change discovery steps. Multi-label classification models built from fingerprints created during controlled system change exercises are used for predicting types of system events observed in real-life systems.

approaches are generally applicable for system changes caused by other events, and the procedures remain essentially the same and are independent of the types of system events.

In this section we first explain how changesets are created, then we explain our fingerprinting techniques that extract representative features from changesets. After that we provide the details of our multi-label classification algorithms.

3.1 Changeset Creation

A changeset c_k can be created from two system observations (say o_{k-1} and o_k). *Observations* mark metadata associated with multiple system features including system configurations, active/passive connections, file system state, packages and processes. The differences of the two observations are stored in the changeset, i.e., c_k=o_k-o_{k-1}. An example changeset is provided in Fig. 2. First three layers of Fig. 1 exemplifies the process of changeset creation from observations made on a system encountering multiple system events.

A changeset created from the difference of two observations includes the created, modified and deleted features. Considering a changeset c_k and its corresponding observations o_{k-1} and o_k, if a feature is in o_k but not in o_{k-1}, then it is a created feature; If a feature exists in both observations, but its attributes differ, then it is a modified feature; and finally if a feature is in o_{k-1} but not in o_k, then it is a deleted feature.

```
CREATED: {
    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) > ...
    },
},
MODIFIED: {
        ... < similar entries to "Created" > ...
},
DELETED: {
        ... < similar entries to "Created" > ...
}
```

Fig. 2. An example changeset.

Our studies show that a tremendous portion of modified features stem from “background noise”, e.g., backend executing processes, system monitoring tools, log files etc., with only some modifications in their timestamps. Thus, they are not good indicators for system events that we want to discover. In addition, when applications are installed, there is only a very tiny portion of deleted features, while most of features are created features. Since in this work we focus mostly on discovery of application installations, we generate fingerprints by only using created features in the rest of the paper.

Among all different types of features in observations, file features account for the most significant part of changesets, and moreover, in most scenarios they are sufficient in identifying system changes, not only for application installations, but also for software and system updates, re-configurations, etc. Therefore, in this work we mostly utilize file features in changesets to generate our fingerprints.

3.2 Changeset Fingerprinting Techniques

Directly applying raw changesets in application and system discovery is inefficient, due to the fact that raw changesets are usually quite large and include a number of irrelevant metadata records for the discovery purpose. These irrelevant records act as “noise” and reduce the accuracy in discovery. It is important to extract clean and concise features from the raw changesets. We call the set of extracted features *fingerprints*. Fingerprints represent raw changesets in a clean and condensed form, while also keeping strong distinguishing capabilities that can be leveraged in the discovery. In this work, we explore two fingerprinting approaches, namely histogram and file embedding fingerprints.

3.2.1 Filename and Histogram Fingerprints

The simplest fingerprint design is to represent the changeset with a list of file names of all created file features, which we call as the *filename fingerprint*. Filename fingerprints have strong distinguishing capabilities, as the combinations of filenames of all created file features caused by different system events are mostly unique.

A changeset may contain thousands of file features, and the filename fingerprint is still not sufficiently compact. Furthermore, a number of state-of-the-art machine learning algorithms are applied on quantified features rather than text-based features. For these reasons, we hash and project the filename fingerprint into a new type of fingerprint, the *histogram fingerprint*.

We generate histogram fingerprints using a simple hashing and binning mechanism. This can be achieved in multiple ways. The approach we took is as follows: We first calculate the American Standard Code for Information Interchange (ASCII) sum of all characters that the file name contains, then we generate a histogram of these numerical values. The number of bins, as well as the range of each bin in histogram is pre-determined. Finally, we normalize the histogram so that it is independent of the total number of filenames in the changeset. The length of histogram fingerprint is fixed at the number of bins. Details of the histogram fingerprint generation process and the determination of the number of histogram bins can be found in prior work [4].

3.2.2 File Embedding Fingerprints

To harvest the hidden semantic context among files and folders, we propose "file embedding fingerprints". File embeddings try to map files in a system to vectors of real numbers in a high-dimensional space (200 to 500 dimensions), while preserving the semantic relations between files. Our file embeddings are an adoption of "word embeddings" [16], commonly used as part of the natural language processing and deep learning approaches employed in Web data analysis. Word embeddings are known to transform words to vectors in a manner that preserves the contextual similarity among words, e.g. words with similar meanings have similar vectors. We strive to do the same with file embeddings.

Word embeddings are generally created by shallow neural network models that try to either predict the current word by using a fixed window of surrounding context words or predict the surrounding window of context words using a given word. word2vec is a successful and fast open source word embedding implementation developed by Google [17]. Instead of building our own neural network models, we use word2vec to create our filename embeddings. word2vec expects a list of sentences as input and provides a vector for each unique word in the provided corpora. To use it as a file embedding tool, we provide to word2vec lists of files/folders we create from the file-tree structure and from the files that are in the same folder with the files we are interested.

We propose two different models, *tree2vec* and *file2vec*, that respectively utilize the file-tree branch that a file belongs to and the folder that a file resides in, to define the "surrounding window of context words" for a given file. For example, assume that for a given file `/usr/texlive/2015/bin/x86_64-darwin/latex` the file-tree structure and folder structure is as follows:

```
usr
└─ texlive
   └─ 2015
      └─ bin
         └─ x86_64-darwin
            ├─ bibexport
            ├─ dvipdfm
            ├─ epstopdf
            ├─ latex
            ├─ ps2eps
            └─ texdiff
```

In this case, for the file `latex`, the tree2vec context is {`usr`, `texlive`, `2015`, `bin`, `x86_64-darwin`} and the file2vec context is {`bibexport`, `dvipdfm`, `epstopdf`, `ps2eps`, `texdiff`}.

In practice, for training and creating tree2vec and file2vec dictionaries that map files to vectors, we collect all the created files and their tree2vec and file2vec contexts to create a corpora that we feed into word2vec, which provides us with a tree2vec and file2vec dictionary for files. To create tree2vec and file2vec fingerprints, for each changeset we sum the tree2vec and file2vec vectors of the newly created files in the changeset and then normalize these vectors to unit vectors.

3.3 Multi-Label Classification for Multiple Event Identification

It is practically not possible to perform system observations such that changes caused by each system event can be captured in isolation in a changeset. Sometimes multiple system events occur at the same time, sometimes new events occur while a system observation is in progress, sometimes a system event can trigger multiple consecutive system events. In such scenarios, a changeset can contain multiple (possibly partial) changes from multiple system events. To be able to use such observations in a machine learning framework and to correctly learn from such multiple events that accrued between two observations, we label such changesets with the set of events that they represent. To be able to correctly identify the set of events that are represented in such accumulated changesets, we use multi-label classification algorithms.

Existing multi-label classification methods can be mainly grouped into two categories: *problem transformation methods* and *algorithm adaption methods*. Problem transformation methods transform the dataset to fit for the classification algorithms, while algorithm adaption methods modify the classification algorithms to adapt to the multi-label dataset. Widely used problem transformation methods include binary relevance and cali-

TABLE 1
Dataset properties.

	Controlled		Uncontrolled	
	Train	Test	30 sec	45 sec
Number of iterations	150	150	15	15
Number of changesets	13,350	12,975	398	299
Number of labels	39,750	70,125	1,304	1,284
Avg # labels / changeset	2.98	5.40	3.28	4.29
Min-Max # labels / changeset	1-5	1-10	2-5	2-6

brated label ranking, while popular algorithm adaption methods include Multi-Label k-Nearest Neighbor (ML-kNN), Multi-Label Decision Tree (ML-DT), and Ranking Support Vector Machine (Rank-SVM) ([18], [19], [20]).

In this work we apply the *binary relevance* method for multi-label classification. In binary relevance, a binary classifier is trained for each label independently. Then the given test sample is labeled as the combination of every output of these binary classifiers [21]. We apply the binary relevance approach onto widely used learning algorithms including k-Nearest Neighbor (kNN), Logistic Regression (LR), Decision Tree (DT), Support Vector Machine (SVM) (with different kernels, e.g., linear and the RBF kernel), and some ensemble boosting algorithms including Random Forests (RF), Adaptive Boosting (AdaBoost) and Gradient Boosting (GB).

On top of the binary relevance method, we further design a "confidence value based ranking" approach for discovery, when the number k of applications installed in the changeset is known ahead of time or can be predicted. In this approach, instead of directly reporting the outputs of binary classifiers, we sort the "confidence values" of all the classifiers and select the top k highest scoring labels as the final labels for the changeset. We note that the number of applications can be estimated by various approaches. One approach we adopt is observing the histogram of file creations along time. When an application is installed, an increase in the number of files created can be observed. We count the number of such "spikes" in the time period during which the changeset is taken, and use this number as an estimation of k. Note that this is only a rough estimation because on one hand, multiple applications can be installed simultaneously and thus contained in the same "spike", while on the other hand, stalls are not uncommon in even a single application installation, which can lead to multiple spikes. Still, as will be discussed in Section 5, using creation time histograms proves to be a reasonable estimation in general.

4 Experimental Methodology

Our focus in this study is on identifying application installations in systems, as application installations are one of the major sources of changes in cloud systems [15]. As identification targets, we use a catalog of 91 most popular non-graphical applications found in the standard CentOS 7 package repositories.

Algorithm 1 GetControlledObservations($AppList\ A$)

```
1: Observation sequence $\mathcal{O} \leftarrow \emptyset$
2: $o \leftarrow Observe()$
3: $\mathcal{O}.Append(o)$            ▷ record initial system state
4: $A^{rand} \leftarrow RandomPermute(A)$
5: for each app $a \in A^{rand}$ do
6:     $Install(a)$            ▷ install app $a$
7:     $o \leftarrow Observe()$   ▷ observe system state
8:     $\mathcal{O}.Append(o)$        ▷ save state to sequence $\mathcal{S}$
```

Algorithm 2 CreateControlledChangeSets($Seq\ \mathcal{O}, Size\ k$)

```
1: for $i \leftarrow 1$ to $length(\mathcal{O}) - k$ do
2:     $c \leftarrow \mathcal{O}[i+k] - \mathcal{O}[i]$         ▷ compute diff of two observations
3:     $C^{train} \leftarrow C^{train} \cup \{c\}$     ▷ store changeset to training set
```

We collect two sets of changeset datasets. The first dataset represents the type of changes observable in a controlled environment where observations do not overlap with system events of interest, while the second dataset represents on-demand usage of our system with possible overlaps in system observations and events of interest (i.e., observations are possibly taken when a system event is also happening). Generic properties of these datasets are presented in Table 1.

The first "controlled" dataset is collected by installing one application at a time to a target VM (in our case, a Google Compute Engine (GCE) VM of type `n1-standard-1` with 1 vCPU core and 3.75GB memory) and then making an observation to record the changes made to the file systems during the installation. This application installation and observation collection process is repeated 300 times, creating 300 "iterations", with each iteration randomly shuffling the list of 91 target applications. From the set of collected observations we create changesets that contain between one and ten application installations, leading to 26,325 changesets. We label these changesets with the application(s) installed during the interval they cover. The processes used for observation collection and changeset creation are detailed in Algorithm 1 and Algorithm 2, respectively.

The controlled dataset is divided into two parts. The first 150 iterations – evenly made up of one, two, three, four, and five application changesets – are used as training data in our models, and contain a total of 13,350 training changesets. The second 150 iterations – made up of one to ten-application changesets (distributed evenly) – are used as test data in our controlled dataset experiments, and contain a total of 12,975 changesets. It is important to note that no duplicate changesets exist between the train and test datasets.

For creating the second dataset that we call the "uncontrolled" dataset, the same set of 91 applications are installed in random order on target VMs. However the application installation and observation processes are not performed in synchrony. Between each application installation, a random wait time between 5 and 10 seconds is added and the observations are made in the background at fixed (30 and 45 seconds) intervals while

Algorithm 3 GetUncontrolledObservations($AppList\ A$, $Interval\ t$)

```
1: Observation sequence $\mathcal{O} \leftarrow \emptyset$
2: $\triangleright$ collect periodic observations every $t$ secs in background
3: while $true$ in background do
4:     $Sleep(t)$            $\triangleright$ wait $t$ seconds
5:     $o \leftarrow Observe()$   $\triangleright$ observe system state
6:     $\mathcal{O}.Append(o)$      $\triangleright$ save state to sequence $\mathcal{O}$
7: $A^{rand} \leftarrow RandomPermute(A)$
8: for each app $a \in A^{rand}$ do
9:     $Install(a)$          $\triangleright$ install app $a$
10:    $t^{rand} \leftarrow random()\%5 + 5$
11:    $Sleep(t^{rand})$        $\triangleright$ wait bw 5 to 10 secs
```

the applications are being installed.

For our set of 91 applications, the average installation time we observed in our systems is 1.9 seconds and the maximum installation time is 5.0 seconds. Hence, changesets generated from these observations have a varying number of applications (between 2 and 6). Note that the installation time here does not include the installation time of dependency packages of applications. We assume that all the dependencies have been installed in advance. In this dataset changes associated with an application installation process can be distributed across two changesets, with each changeset containing partial information about application installation event. On average, we have as many partial changes as the number of changesets in the dataset.

Algorithm 3 details the asynchronous observation collection process for test data collection. While collecting this dataset we used slightly more powerful VMs of type `custom-2-5120` with 2 vCPU cores and 5GB memory, so that the observation process would not interfere with the installation process. The observation process is repeated 30 times (15 times with each interval) with randomly shuffled application lists to obtain our uncontrolled test dataset.

The distribution of the changesets in the controlled and uncontrolled datasets based on the number of applications are presented in Fig. 3. As seen in the figure, for the controlled train dataset, the changesets are roughly equally divided between changesets that contain one to five applications. For the controlled test dataset the changesets are roughly equally divided between changesets that contain one to ten applications. For the uncontrolled dataset, the majority of changesets consist of changesets with three, four, and five applications, but there exists a small percentage of uncontrolled changesets with larger than five applications (∼7%).

We evaluated the improvements provided by two types of fingerprinting approaches, namely just histogram fingerprints, and combining file2vec, tree2vec, and histogram fingerprints, which we call the *combined fingerprints*. We note that histogram fingerprints are cheap to compute while file2vec and tree2vec fingerprints are rather costly as they require training neural network models using sizable data. The size of file2vec and tree2vec "sentences" we create from file tree structure and neighbor files of created files are 47GB and 1.5GB, respectively. Training new dictionaries from these sentences took 240 minutes and 8 minutes, respectively on a 2.3GHz, 8 CPU system with 8GB RAM. We note here that the whole training procedure is performed offline/off-the-critical-path.

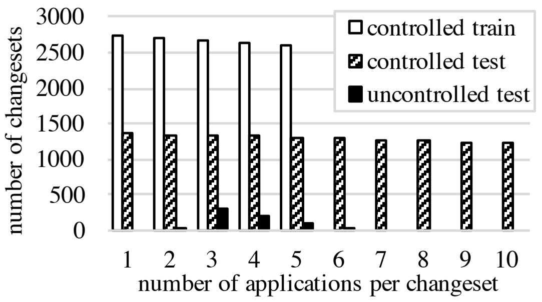


Fig. 3. Distribution of datasets based on the number of applications in their respective changesets.

We also investigate the performance of commonly used classification algorithms such as logistic regression (`lr`), support vector machines (`svm`), k-nearest-neighbor (`knn`), random forest (`rf`), and decision trees (`dt`) for prediction, along with the impacts of boosting algorithms adaptive boosting (`adaboost`) and gradient boosting (`gb`) on the performance of decision trees. All of these algorithms are used in a one-versus-all framework, where a binary classifier is built for each possible application. The performance of our "confidence value based ranking" approach that first predicts the number of application installations and then reports that many highest scoring applications as predictions is also reported.

The Python language and machine learning library scikit-learn [22] are used in implementing both the training models and the testing framework. The performance of the trained models on correctly predicting application labels is measured using the standard information retrieval metrics of recall, precision, and F1-score. Note that for the case of predicting multiple application installations within a changeset, recall indicates the fraction of actual installations that are correctly predicted, precision indicates the fraction of made predictions that are correct, and F1-score is the harmonic mean of these recall and precision values.

5 Evaluation

We evaluated the prediction accuracy of two fingerprinting algorithms (histogram and combined) and various machine learning algorithms (`dt, rf, lr, knn, svmlinear, svmrbf, adaboost, gb`). All machine learning algorithms presented are trained using the controlled-train dataset. Parameters of utilized algorithms are optimized over the controlled-train dataset using five-fold cross-validation. We report F1 scores as our primary measure of prediction accuracy, although we present and discuss precision and recall when they provide insight about performance of solutions.

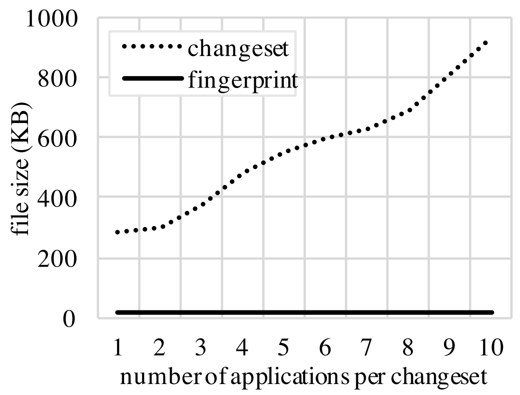


Fig. 4. Comparison of fingerprint and average changeset file sizes as the number of applications per changeset increase.

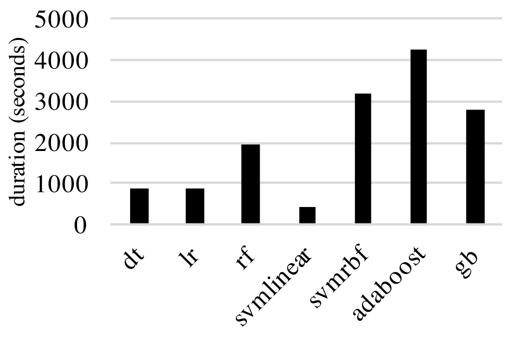


Fig. 5. Training duration of machine learning models while training with the controlled train dataset.

5.1 General Observations

Fig. 4 provides a comparison of fingerprint sizes with the average changeset file size as the number of applications in the changeset increase. Fingerprints have a two orders of magnitude smaller size than changesets and their size is independent of the number of events.

Fig. 5 shows the training times for the machine learning models using the controlled train dataset. These models are trained on a 2.3GHz, 8 CPU system with 8GB RAM. Algorithms with fast training times are preferable considering that numerous updates and changes are made to applications everyday and retraining models daily with new data is a requirement. The observed training times in Figure 5 indicate that none of the machine learning methods have a prohibitively long training time considering daily re-training requirements.

5.2 Experiments on the Controlled Dataset

Fig. 6 provides the average prediction accuracies (F1-scores) we observed for the two fingerprinting and various machine learning algorithms while predicting the application installation events of controlled test dataset. As seen from the figure, `lr`, `svmlinear`, `svmrbf` and `adaboost` perform considerably better than other machine learning algorithms. Note that the simpler histogram fingerprint features performed considerably well for most algorithms.

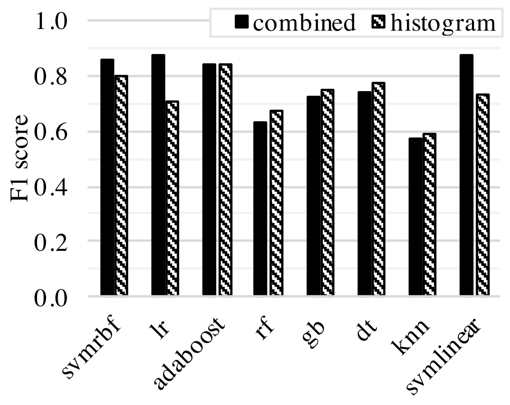
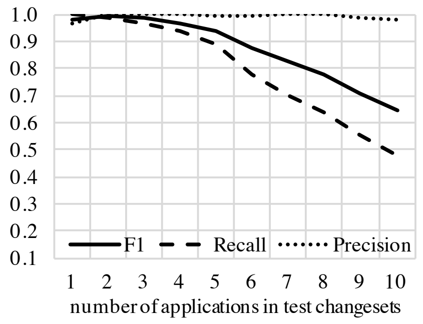


Fig. 6. Average F1-scores observed for various machine learning algorithms while testing over the controlled-test dataset using both combined and histogram fingerprints.

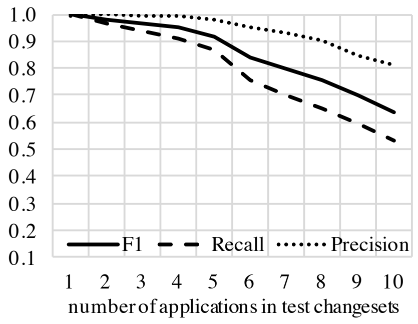
In Fig. 7 we put some of our best performing machine learning algorithm - fingerprint combinations under more scrutiny and observe their precision, recall and F1-scores as the number applications in the test changesets increases. As expected, all algorithms exhibit a downward performance trend as the changeset sizes increase, showing that the multi-label classification problem becomes harder as the number of labels to be predicted increase. The downward slope of prediction accuracy accentuates when the number of applications in the test changesets go beyond five applications, which indicates that the number of applications in the training changesets have some effect on performance even though we are performing one-vs-all classification.

We note that the precisions of the algorithms are generally high, and histogram fingerprints lead to higher precision values (higher than 0.90 for `adaboost` and close to 100% for `lr` and `svmrbf`), indicating that histogram fingerprinting algorithm rarely leads to false predictions. This is expected as histogram fingerprints are distinctive. However, as the number of applications in a changeset increase beyond five applications we start to observe a significant decrease in recall for all applications. The recall values of some algorithms go as low as 0.50 for changesets with ten applications indicating that these algorithms can fail to identify almost half of the events when there are many events in a changeset. A closer look at the confidence values provided by high-precision algorithms with histogram fingerprints indicated that the confidence values for missed labels were actually high, just not as high as the learned thresholds.

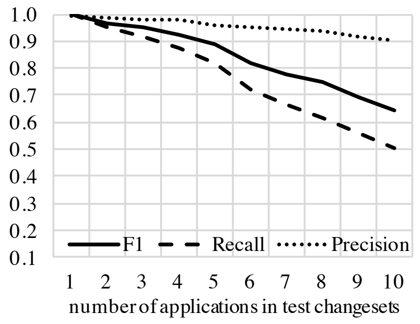
The observations above led us to hypothesize that if the number of events in a test changeset were known or provided, by simply picking the same number of highest confidence predictions of our algorithms, we could observe higher recall values. We note that predicting the number of events observed in a changeset is relatively an easier problem if sufficient metadata is available. For example, by just grouping the new file creations based on their timestamps we were able to predict the number of applications in all of our controlled changesets with



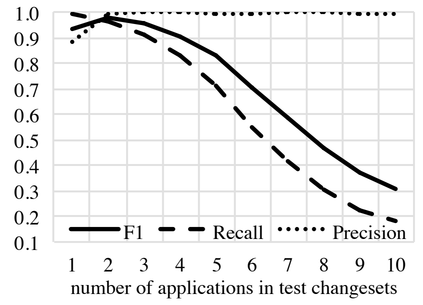
(a) Logistic Regression - Combined



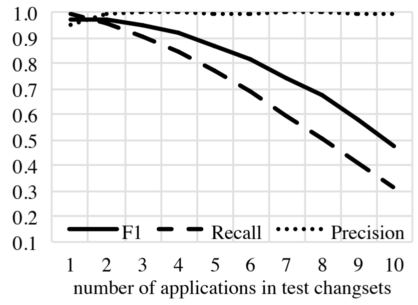
(b) SVM-RBF - Combined



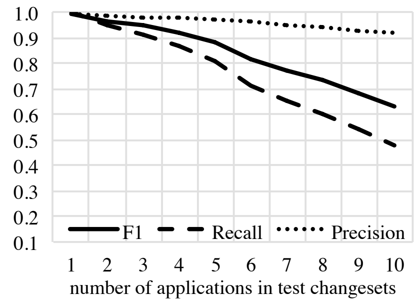
(c) Adaboost - Combined



(d) Logistic Regression - Histogram



(e) SVM-RBF - Histogram



(f) Adaboost - Histogram

Fig. 7. Precision, recall and F1-score of algorithms as number of applications in the changesets increase.

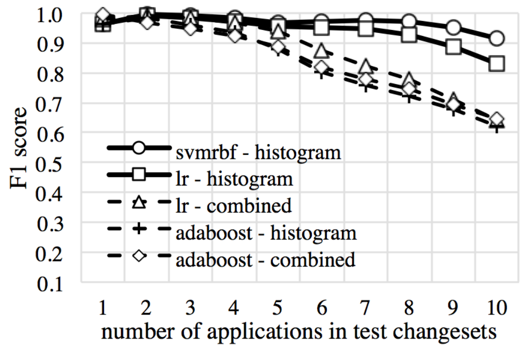


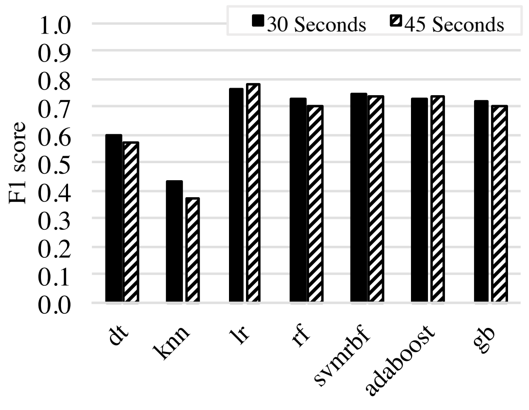
Fig. 8. Results of our "two-step" prediction procedure for algorithms with high precision.

an experimental error of less than 1.6%.

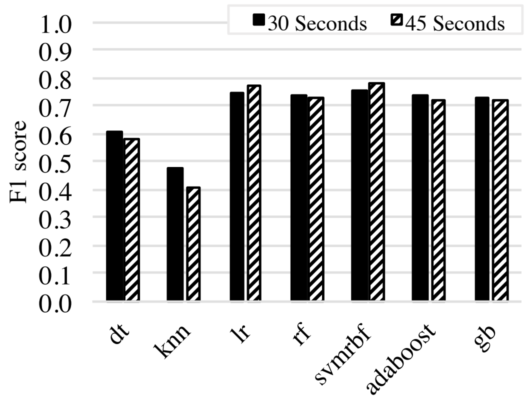
We test our hypothesis on the set of machine learning algorithms that have provided high precision values even for large number of applications. The results of our analysis are shown in Fig. 8. In these experiments we first use the aforementioned file creation time analysis to predict the number, say k, of events that occurred within a changeset. Then we select the top k predictions of the machine learning algorithms. Note that this approach is only applicable to learning algorithms that provide confidence values. We observe that providing the number of applications in the changesets proved to be very helpful to some algorithms. Our new approach was immensely helpful to `svm-rbf` using histogram fingerprints, which saw F1 scores greater than 0.9 in all cases, and `lr` using histogram fingerprins, which saw F1 scores greater than 0.9 in all but 9-app and 10-app cases, but had no impact on the performance of `adaboost` algorithm and any algorithm using the combined fingerprints. We believe this is due to the fact that the thresholds and weights learned by `adaboost` are highly dependent on the training dataset properties.

5.3 Experiments on the Uncontrolled Dataset

We test the performance of the models trained using the controlled-train dataset over the more difficult uncontrolled dataset using our two-step predictive models with histogram and combined fingerprints as well. Observed F1-scores are depicted in Fig. 9. Observed F1-scores are significantly lower due to the fact that almost all changesets created in this scenario contain partial installations. Some of these partial installations contain little or no information so corresponding labels are harder to predict. As seen in the figure, across the board, increasing the interval between snapshots seem to have little effect. Even though 45 second changesets have more applications (1 more application on average) they contain less number of partial events leading to such mixed results for 30 and 45 sec experiments. A quick check of accuracies show that performance of histogramn and combined fingerprints are similar, and

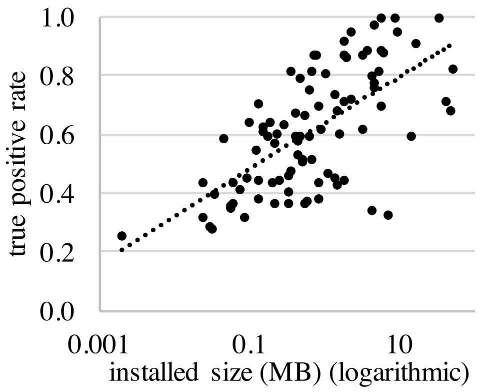


(a) Combined

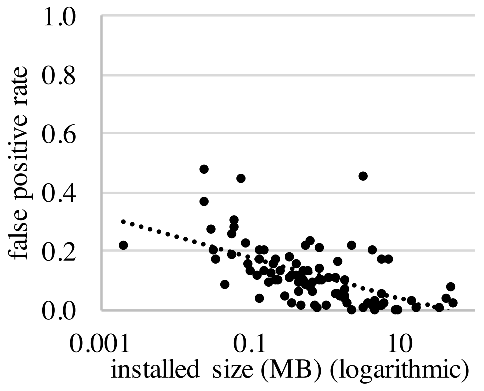


(b) Histogram

Fig. 9. Performance of tests over uncontrolled dataset with 30 second and 45 second observation intervals using (a) combined and (b) histogram fingerprints.



(a) True Positive Rate



(b) False Positive Rate

Fig. 10. Logarithmic correlation between the size of an appliaction and (a) True Positive Rate and (b) False Positive Rate we observe during prediction.

for both fingerprint types `lr` and `svmrbf` are the top contenders, achieving F1-scores in the range 0.76 – 0.78.

5.4 Application Size Analysis

Our experiments indicate that there is a significant correlation between correct prediction likelihood and the "installed size" of an application (according to each application's record in the CentOS yum package repository). We note that sizes of dependency packages are not included included an application's size. Fig. 10 shows this logarithmic trend in the true- and false-positive prediction rates of individual applications averaged over all prediction algorithms. This leads us to surmise that smaller applications tend to be harder to predict correctly and easier to predict falsely. False predictions are likely due to overly-general fingerprints created for small applications.

6 Change Discovery as a Service

To demonstrate a real-life implementation of our work, we have started architecting DeltaSherlock, a system that provides change discovery functionality as a service. Fig. 11 provides an overview of the DeltaShelock architecture. In a typical deployment, the service would be installed on a client device and configured to perform observations of the file system on a fixed interval. Every time an observation is performed, the client would prepare a changeset representing any changes made between the last observation and the current one. Afterwards, it would prepare a fingerprint using the newly-generated changeset and dictionaries provided by a separate server device. The prepared fingerprint is then sent off to the server for analysis, which in turn sends its prediction(s) back to the client after analysis is complete. The client finally stores the results in its log and, depending on the results received, could take appropriate actions ranging from emailing an alert to automatically quarantining the system from the network or shutting it down.

In order to make accurate predictions, the server has to maintain large and frequently-updated databases of dictionaries, fingerprints, and changesets representing the events it intends to identify. To accomplish this task, we have designed a "swarm of trainers" model, made up of several trainer node machines constantly installing and observing software from various repositories, feeding new changesets and updated dictionaries to

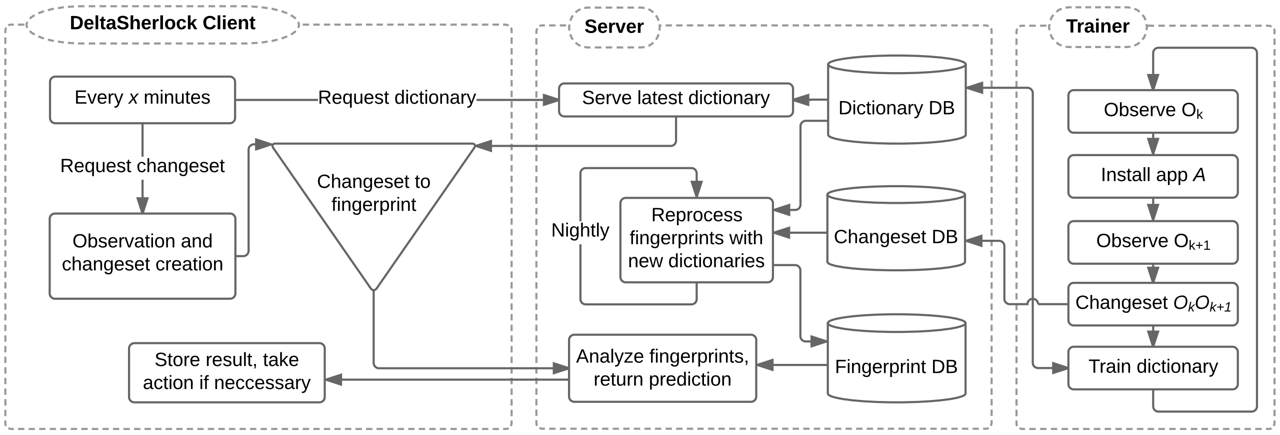


Fig. 11. High level view of the proposed DeltaSherlock service.

the server's databases along the way. Every night, after receiving this new data, the server would regenerate new fingerprints to be used for analysis and prediction. We also envision allowing users to generate their own changesets by creating a set of pre- and post-installation "hooks" that could be installed into a package manager in order automate the observation process.

7 Conclusion

In this study, we proposed DeltaSherlock, a framework for automatic and efficient identification of multiple changes observed by cloud instances. Our experiments over 25,000 system changes in the form of software installations collected from VMs deployed over a commercial cloud reveal that the proposed approach can identify multiple events with high accuracy (up to 96.8% accuracy) by just investigating the changes observed by the file system. We also present our initial design for providing DeltaSherlock as a service. Our ongoing efforts include running the DeltaSherlock service on a public cloud and expanding the capabilities of discovery by utilizing other system features.

References

[1] K. Weins, "Cloud computing trends: 2016 state of the cloud survey." http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey#enterpriseworkloads, 2016.

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

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

[4] H. Chen, S. S. Duri, V. Bala, N. T. Bila, C. Isci, and A. K. Coskun, "Detecting and identifying system changes in the cloud via discovery by example," in *2014 IEEE International Conference on Big Data (Big Data)*, pp. 90–99, 2014.

[5] H. Chen, A. Turk, S. S. Duri, C. Isci, and A. K. Coskun, "Automated system change discovery and management in the cloud," *IBM Journal of Research and Development*, vol. 60, no. 2-3, pp. 2:1–2:10, 2016.

[6] IBM, "Endpoint manager relevance language guide [online]." http://pic.dhe.ibm.com/infocenter/tivihelp/v26r1/topic/com.ibm.tem.doc_8.2/Relevance_Guide_PDF.pdf, 2012.

[7] "Openioc [online]." http://www.openioc.org/.

[8] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma, "Automated known problem diagnosis with event traces," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 375–388, Apr. 2006.

[9] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 687–700, 2014.

[10] M. D. Dikaiakos, A. Katsifodimos, and G. Pallis, "Minersoft: Software retrieval in grid and cloud computing infrastructures," *ACM Transactions on Internet Technology (TOIT)*, vol. 12, no. 1, p. 2, 2012.

[11] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proceedings of the 5th European conference on Computer systems*, pp. 111–124, ACM, 2010.

[12] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," *SIGOPS Oper. Syst. Rev.*, vol. 39, pp. 105–118, Oct. 2005.

[13] C. Isci and V. Bala, "ASPLOS 2014 tutorial - origami: Systems as data." https://sites.google.com/site/origamisystemsasdata/asplos2014, 2014.

[14] G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang, "Virtual machine images as structured data: the mirage image library," *Proceedings of the USENIX HotCloud*, 2011.

[15] S. A. Bohner, "Impact analysis in the software change process: A year 2000 perspective," in *Software Maintenance 1996, Proceedings., International Conference on*, pp. 42–51, IEEE, 1996.

[16] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, Mar. 2003.

[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[18] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *Dept. of Informatics, Aristotle University of Thessaloniki, Greece*, 2006.

[19] M.-L. Zhang and Z.-H. Zhou, "A review on multi-label learning algorithms," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 8, pp. 1819–1837, 2014.

[20] G. Madjarov, D. Kocev, D. Gjorgjevikj, and S. Džeroski, "An extensive experimental comparison of methods for multi-label learning," *Pattern Recognition*, vol. 45, no. 9, pp. 3084–3104, 2012.

[21] E. A. Cherman, M. C. Monard, and J. Metz, "Multi-label problem transformation methods: a case study," *CLEI Electronic Journal*, vol. 14, no. 1, pp. 4–4, 2011.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.