Automated System Change Discovery and Management in the Cloud

Hao Chen, Ata Turk, Sastry S. Duri, Canturk Isci and Ayse K. Coskun

Emerging cloud service platforms are hosting hundreds of thousands of virtual machine instances, each of which evolves differently from the time they are provisioned. As a result, cloud service operators are facing great challenges in continuously managing, monitoring and maintaining a large number of diversely evolving systems, and discovering potential resilience and vulnerability issues in a timely manner. This paper introduces an automated cloud analytics solution that is based on using machine learning for system change discovery and management. The learning-based approaches we introduce are widely used in multimedia and Web content analysis, but application of these to the cloud management context is a novel aspect of our work. We first propose multiple feature extraction methods to generate condensed "fingerprints" from the comprehensive system metadata recorded during the system changes. We then build up an adaptive knowledge base using all known fingerprint samples. We evaluate different machine learning algorithms as part of the proposed discovery and identification framework. Experimental results that are gathered from several real-life systems demonstrate that our approach is fast and accurate for system change discovery and management in emerging cloud services.

Introduction

Cloud computing promises the delivery of on-demand computing resources as a utility that can be used as needed. This promise has led to a revolution in IT technologies causing a rapid transfer of services to the cloud [1]. Regardless of whether a cloud operator uses bare metal computers, virtual machines, or containers to create computing facilities, basic questions remain the same: are these facilities free of any vulnerabilities, configured correctly, and can they avoid drifting from acceptable configuration states? New service automation and DevOps workflows have attempted to address the system drift problems by proposing the use of immutable architectures and tightly structuring software lifecycle into development, build, deployment and operations phases. However, current agile iteration principles that promote continuous development and improvement, and the fast pace of changes in underlying systems and software, counteract some of these benefits. Variability across systems in cloud environments remains a persistent problem. Therefore, discovering potential misconfiguration and vulnerability issues in a timely manner is elusive.

An effective solution to figure out system vulnerabilities and drifts is to monitor, check and analyze each change made to a system since it is booted. To understand what the system changes are about, one can dig out information from historical user or system logs. However, log data is usually too massive to be mined fast and accurately. It is also very inefficient to always keep a huge chunk of logs in storage. On the other hand, to determine if a system change includes software with known vulnerabilities, one can consult the package repository in the system and cross-check that information against, for example, National Vulnerability Database (NVD) [2]. However, a vendor could issue a fix pack that fixes a known vulnerability without changing package version. Sometimes vendors could back-port fixes into packages that have reached end of their life cycle. In

both cases, a single package name links to several different versions of packages: some of them are vulnerable while others are not. Further, users could install software from sources without using package managers. Simply using logs, package managers and repositories fails to discover vulnerabilities in all these scenarios.

Manually written rules that check for the existence of certain indicative features such as the existence of certain files, configuration parameters are used in addition to consulting package repositories in the system [3-5]. While these rules are sufficient to detect the presence of software for license purposes, they are not capable of discriminating between a vulnerable package, and one that includes a fix for it. Furthermore, approaches based on such rules are fragile and require constant maintenance, indicating a substantial amount of manual effort. A great amount of today's software gets released multiple times a week, and most of systems change everyday. Rule-based approaches have difficulties in keeping up with the pace of software and system changes.

Alternative methodologies that build inverted indexes of file tree structures to enable keyword-based searching for software discovery are mostly useful in scenarios where users have a deep understanding of the underlying file/process structures associated with the software they are searching for and can produce specific keywords to query [6]. However, as file names can be repetitive, uninformative, and misleading, the results of such systems are useful in narrowing down the search space but are not conclusive or comprehensive.

In this paper, we introduce an automated cloud analytics solution that generates *fingerprints* of changes in system state, and utilizes these fingerprints in a machine learning platform to perform system change discovery and management. We first propose multiple novel feature extraction methods to generate condensed fingerprints from the comprehensive metadata associated with the system change events. Our fingerprinting methodologies mostly focus on the file system features, and tend to represent changes in system state in a compact form. They can learn the hidden context behind filenames, and represent them with vectors utilizing the file tree structure and/or file co-location information to capture the semantic relationships of files. Using these fingerprints, we build an adaptive knowledge base that enables fast comparison of system state changes with previously labeled changes. More specifically, we learn the discovery model from the knowledge base with learning algorithms and then predict the new-coming system changes by the model. We then conduct experiments mainly based on system changes caused by software installation in this paper. Typical system changes include: software installations, updates, system reconfigurations and process executions. Among them, software installation is one of the most significant factors causing system changes [7]. Note that, our approach, however, is applicable for discovery of system changes caused by any of the above listed factors, as the procedure of the discovery remains essentially the same and is independent of the reasons of the changes. We evaluate several machine learning algorithms as part of the proposed discovery and identification framework on our knowledge base. We show that our mechanism can be utilized for fast (in a few milliseconds or seconds) and accurate (up to 98.75%) software and system change discovery.

Overview of System Change Discovery Framework

Our system change discovery framework is composed of three phases: (I) change set creation, (II) training, and (III) discovery. A *change set*, which contains all changes that happened to the system during a system event (e.g., a software installation), is crawled and recorded in the change set creation phase. **Figure 1** shows the change set creation flowchart. The training phase is composed of two stages: the fingerprint extraction and the model-learning. A *fingerprint*, a compact representation of each change set, is created in fingerprint extraction phase. In the model-learning phase, a knowledge base is first built up by change sets with known labels, and their corresponding fingerprints. The "label" here represents the name of the event that leads to the system changes. It can be a software package installation, e.g., "Apache Tomcat™ installation", update, e.g., "Tomcat update", or system configuration, e.g., "Tomcat configuration", etc. All fingerprints along with their labels in the knowledge base are then supplied to the learning algorithms to generate a machine learning *model*. Finally in the discovery phase, the learned model is utilized in the task of label prediction for new unidentified changes. Newly labeled change sets and their corresponding fingerprints are then stored into the knowledge base for future learning, which makes the knowledge base iteratively updated. In this way, the whole discovery system is automated and requires little to no human intervention in the long-term. Manually labeled training samples are only required at the beginning of the initialization of the knowledge base. After the initialization, human operators only need to verify or clarify samples that are labeled with low confidence, which only constitute a small set of whole samples. **Figure 2** provides an overall view of the training and discovery phases.

In the following sections, we first discuss the change set creation phase, in which we define what a change set is and how it is created. We then study the training phase. We discuss multiple fingerprinting methodologies to capture the extensive information stored in change sets in a compact form, followed by presenting various learning algorithms that we utilize for training the model. We then briefly discuss the system change discovery phase. Finally, we introduce the experimental methodology followed by an analysis of the performance of our discovery framework and discussions.

Phase I. Change Set Creation

A change set is the record of all changes that happened to the system during a system event, such as a software installation. It contains all entities that are created, modified or deleted during the event, e.g., files, packages, processes and configurations [8].

Change set creation process and an example change set is shown in **Figure 1** and **Listing 1** respectively. We create the change set by utilizing IBM's Origami service [9, 10]. As an example to change set creation, consider the installation of an application such as Apache Tomcat™, an open source Java Servlet software. A "snapshot" S_1 of the system is taken at T_1, followed by the installation of the subject software, in this scenario Tomcat, followed by a second "snapshot" S_2 of the system at T_2. The difference of two snapshots, i.e., $D = S_2 - S_1$, is a change set and we label it with the "Tomcat Installation" label to mark that this change set represents the system state changes observed due to an Apache Tomcat installation. Technically, a "snapshot" is taken as a text file consisted of metadata of the system, and the difference D is the output of a "text diff" applied on two snapshots.

Phase II. Training

Training has two stages, namely fingerprint creation and learning stages. In training, fingerprints are extracted from raw change set data, stored in a knowledge base, and a discovery model is then learned from data in knowledge base. Training process and its relationship with the discovery process is shown in the upper part of **Figure 2**.

Fingerprint Creation

Condensed key information is required to be extracted, either explicitly or implicitly, from change sets before they can be used to train the prediction models. The process of key information extraction is called "fingerprinting", and the extracted key information is defined as the "fingerprint", for each change set. In this section, we introduce multiple fingerprinting methodologies.

All fingerprinting techniques introduced here use file features in the change set, such as filenames and file paths. An example of file features can be seen in Listing 1. File features constitute the most significant part of change sets, and in most cases using only file features is sufficient in discovery and identifying system changes caused by software installation. It is also sufficient for other causes of system changes such as software updates and system configurations unless these operations do not cause a significant change in file features.

The most intuitive, straightforward, but storage-wise inefficient fingerprint is the *filename fingerprint*. A filename fingerprint is a list of filenames of all recorded files (added, modified, or deleted during the system change event) in a change set. Filename fingerprints are distinguishable because the combination of filenames of all changed files is mostly unique to system change.

A filename fingerprint can be quite inefficient especially when a change set contains thousands of file features. Hence, we propose a condensed numerical representation of these filenames, the *histogram fingerprint* [8]. The process of creating a histogram fingerprint from a filename fingerprint is as follows: (1) transform each filename in the filename fingerprint into a numerical value using some hash function, e.g., calculating the ASCII sum of all characters that the filename contains; (2) calculate histogram by grouping all the numerical values into a few bins, i.e., N_{bins}, and count the number of values in each bin as $C_i, i = 1, 2, 3 \dots N_{bins}$; (3) normalize histogram by: $C_i^{norm} = \frac{C_i}{\sum_{i=1}^{N_{bins}} C_i}, i = 1, 2, 3 \dots N_{bins}$, such that $\sum_{i=1}^{N_{bins}} C_i^{norm} = 1$. The histogram fingerprint is normalized so as to be independent of the total number of filenames in the change set. The length of histogram fingerprint is fixed at N_{bins}.

Both filename and histogram fingerprints utilize the file features as is, without trying to understand the "meaning" of the names of these files. However, it is now possible to capture the syntactic and semantic similarities and relationships between words in natural languages with no human supervision by providing significant amount of textual content to neural networks [11, 12]. Word2vec (w2v) is one such open source machine learning

(neural network) toolkit developed at Google for this specific purpose [11]. It has been shown to successfully capture the similarities among concepts in natural languages.

We propose that w2v can also be used for gleaning the meaning behind filenames. Just as concepts that tend to appear in the same sentence in a specific order have a special relationship, we argue that filenames that appear in the same file tree branch or in the same folder (hence neighbors in locality) have a special relationship, and we propose two fingerprinting methodologies that utilize these two separate sources of information. We feed the file features and their "neighbors" - the set of files that reside in the same folder - as sentences to v2w and create a vector representation for each filename that we call "neighbor vector" of a filename. For each change set, we sum the "neighbor vectors" of the changed files in the change set by performing a simple vector addition. Then we normalize the summation vector to a unit vector to obtain a *neighbor fingerprint*.

Similarly, by feeding the filename of a changed file in the change set together with the folder names that are in the same file tree branch as a sentence to v2w, we create another vector representation for each filename, called as the "file-tree vector" of a filename. For each change set, by adding the file-tree vector representations of the changed files and then normalizing the summation vector to a unit vector, we obtain a *file-tree fingerprint*.

When provided with sufficient amount of folder and file tree information, we observe that w2v can easily identify the semantic relationship between files. In **Figure 3** we display two-dimensional vectors created by w2v for a set of filenames when file tree information is supplied to it. As shown via dashed circles in the figure, even when the vector dimensions are as low as two, w2v manages to retain a sense of the semantic relationship among the software objects represented by filenames and it is even possible to roughly group the filename vectors based on these semantic relationships. As an example, it is possible to observe from the figure that the 2D vectors for Emacs – the popular Linux editor – and Lisp – the programming language used for implementing most of the editing functionality built into Emacs – are very close. Please recall that these vectors are not fingerprints themselves but they are informative inputs to the fingerprinting algorithm. However, using w2v supplied vectors of changed filenames for fingerprinting enables the fingerprinting algorithm to retain a semantic sense of the installed program. When vector dimensions are increased to 200 or more, w2v starts to display much more accurate results. We should also note that w2v supplied vectors also retain a sense of relative relationship between files. As an example, when using neighbor vectors, we observed in our data that the relationship between "apache-commons-dbutils.jar" and "apache-commons-dbutils.xml" is akin to the relationship between "ivy.jar" and "ivy.xml".

Learning Using Fingerprints

Having defined our proposed fingerprinting methodologies to represent the change sets observed after system change events in a compact form, we now describe how we use these fingerprints in various learning frameworks to train models that can perform system change discovery. The set of machine learning algorithms we consider for system change discovery include nearest neighbor, logistic regression, support vector machines (SVM), decision trees and random forests. Below we briefly introduce these widely used machine learning algorithms.

Nearest Neighbor (NN) [13, 14] is a classification technique that labels a given sample using the closest (or most similar) samples within a given previously labeled dataset. Closeness is defined by a similarity or distance function, e.g., Euclidean distance, Manhattan distance, cosine similarity, etc. A generalization of this is the k-nearest-neighbor (kNN) algorithm, which utilizes the "k" closest samples. In this paper, we consider the one-nearest-neighbor algorithm with the Euclidean distance. For a pair of fingerprints (f_i, f_j) introduced before (they are both vectors, no matter what type), the Euclidean distance is calculated as $\|f_i - f_j\|$, i.e., the L_2-norm. The smaller the distance is, the more similar two fingerprints are.

Unlike other learning algorithms that have to go over a training phase to provide a learning model of coefficients, support vectors, or decision rules, the NN algorithm requires no training. It simply keeps the set of all training samples, and operates on these samples during the discovery phase to find the nearest neighbor (or k nearest neighbors) of the new-coming samples based on the given distance or similarity function, and reports the corresponding label(s) and their distances as the discovery result.

Logistic Regression (LR) [15] is a classification algorithm that typically deals with binary outputs. The basic idea of the logistic regression is to train a coefficient vector of the feature from a training data set by minimizing a defined cost function using programming methods. It is a generalization from linear regression by applying a logistic function. Logistic regression method can be further generalized to predict the probabilities of more than two possible outputs, i.e., the multi-class logistic regression, with applying the one-vs-all algorithm. In this work, we apply multi-class logistic regression with the L_2-regularization in our problem to avoid over-fitting. The weights on the cost of regression error and the regularization are trained through cross-validation on the training dataset.

Support Vector Machine (SVM) [16] attempts to find an optimal set of hyper-planes in high-dimensional space that divides the samples into classes with largest margins. An SVM model is learned from training samples, which maps the samples as points in space, and divides classes by clear gaps (hyper-planes). Samples are then predicted to classes based on the side of the gap that they fall on. Samples on the margins are called support vectors. We apply one-vs-one algorithm to extend a binary SVM to a multi-class SVM, i.e., N(N-1)/2 classifiers are constructed if we have N classes.

SVM applies kernel functions to map the original space to a higher-dimensional space. The most widely used kernel functions are the linear kernel and the radial basis function (RBF) kernel [17], which are both tested in our experiment. In SVM, a soft margin is typically applied, which chooses a hyper-plane that splits examples as cleanly as possible, though makes a more complex decision hype-plane. The trade-off parameter and other parameters related to different kernels are learned by cross-validation on the training dataset in our experiment.

Decision Tree (DT) [13] is a tree-like graph in which each (non-leaf) node and each branch represents a test on an attribute and the outcome of the test, respectively. Leaf nodes represent classes, into which samples are finally classified after passing through

tests on all attributes. A decision tree is most commonly learned in a top-down induction method, i.e., repeatedly splitting training sets into subsets in a recursive manner based on tests of attributes until splitting no longer improves the prediction performance. Comparing with other learning algorithms, an additional benefit of a decision tree is that the decision rules that are learned from a training data set can be usually visualized in a human-readable manner.

Random Forests (RF) [18] is an ensemble learning method based on decision tree. It constructs multiple decision trees in training and uses the mean or mode of the prediction of individual trees as the final output. Random forest is mainly used to solve the "over-fitting" issue of decision tree.

Phase III. Discovery

In the discovery phase, the models trained on the knowledge base that contains application labels and corresponding fingerprints are utilized for performing prediction over new fingerprints extracted from unobserved change sets. More specifically, the fingerprint of a new coming unobserved change set is generated, input into the model, and the identification (i.e., the label) of the change set is returned. Discovery process and its relationship with training are displayed in the lower part of **Figure 2**.

Experimental Methodology

The datasets used in experimentation are generated as follows: We randomly select 160 software packages from the Linux yum repository and install these packages on two different operating systems in two different cloud environments, namely the Fedora-19 on Amazon Web Service (AWS) EC2 micro instances, and the Fedora-21 on Massachusetts Open Cloud (MOC) [19] medium instances. Note that the approach also applies to other software systems, such as APT-like repositories, manual installation from binaries, etc. We have briefly tested them and observed similar results. In addition, the approach is independent to the location of installation, as we either only use the relative path or not use the path information at all in fingerprint design. In that way, we make sure that the same software installed in different folders can still be discovered. We record the system change set for each installation. We select software package installations as the system change trigger events because software installations are one of the most significant events that can lead to notable system changes. However, the proposed discovery technique is not limited to application installations and can be applied to a variety of system change events, such as security patches, system configurations and process execution, etc.

A change set not only includes records of changes caused by the software installation, but also contains other "background noise", such as temporary files created automatically by the system and changes made by other user operations or unrelated running activities in parallel, etc. Therefore, change sets consist of variations and vary from installation to installation. Even installing the same software on the same instance multiple times leads to different change sets. Moreover, dependency packages are resolved and installed during software installation. Some popular dependencies are shared by multiple software packages, and as a result, during the batch installation of 160 packages, dependencies of

some later installed software packages may have already been installed during installations of prior software. Hence different orders of installations in the batch installation among these 160 software packages lead to differences in change sets. Thus, in order to capture variations in change sets, we batch install 160 software packages multiple times in random order. We install each software package 3 times on different AWS instances and 4 times on different MOC instances to create a training knowledge base. Overall, the training dataset consists of 160 software installation classes with each class containing 7 change set samples. This dataset is also used to generate the w2v dictionaries for neighbor and file-tree fingerprints.

Our testing dataset is generated as follows: we randomly select 80 software packages out of the 160 classes, and install each of them once on a separate AWS instance with Fedora-19. Then we randomly select another 80 software packages and install each of them once on a separate MOC instance with Fedora-21. The change set samples obtained from these installations are used as our discovery test cases. Therefore, our test dataset contains 160 tests in total, with 80 from AWS Fedora-19 installation and 80 from MOC Fedora-21 installations. The test data set is generated in this way so as to capture the experimental varieties of different OSs and platforms. The accuracy of discovery is defined as the number of cases that are correctly identified among these 160 test cases, divided by 160. We test discovery accuracy of all combinations of different fingerprints methodologies and learning algorithms discussed previously.

Experimental Results

Figure 4 shows the discovery accuracy of various combinations of the fingerprinting methodologies and the learning algorithms. We test the performance of the one nearest neighbor (NN), logistic regression with regularization (LR), SVM with linear and RBF kernel (SVM-linear and SVM-RBF), decision tree (DT), and the random forest (RF) machine learning algorithms. In LR, SVM-linear and SVM-RBF, parameters are tuned with cross-validation on the training data set. Either one-vs-one or one-vs-all method is used in each learning algorithm for multiclass discovery, as discussed previously. Since there exist some variations in model generation in DT and RF, the discovery results vary corresponding to different models. We calculate average performance of DT and RF across 20 test runs.

The fingerprints in our experiment include: the histogram fingerprint with different number of bins ($N_{bins} = 20$ and $N_{bins} = 200$), the neighbor fingerprint, and the file-tree fingerprint. The lengths of both the neighbor and the file-tree fingerprints are 200. We also test the accuracy of utilizing combinations of histogram ($N_{bins} = 200$), neighbor and file-tree fingerprints as feature sets. As an example, the histogram + neighbor fingerprint has 400 dimensions, with first 200 dimensions coming from the histogram fingerprint and the last 200 dimensions coming from the neighbor fingerprint. Similarly, the length of the histogram + file-tree fingerprint is 400, and the length of the histogram + file-tree + neighbor fingerprint is 600.

As seen in **Figure 4**, the highest discovery accuracy is as high as 98.75%, and is achieved by using logistic regression on the combination of histogram, neighbor and file-tree fingerprints. All learning algorithms with the exception of the decision tree algorithm,

which may suffer from over-fitting, achieve the best performance when some combinations of fingerprints are used. Histogram fingerprint with 200 bins has consistently better performance than with 20 bins for all algorithms. In our experimental tests we also observed that further increasing the number of bins of the histogram to 1000 or larger counts in fact decreases accuracy, as it leads to highly sparse fingerprints.

We observe from **Figure 4** that utilizing the file neighborhood and file-tree information in fingerprint creation process causes notable improvements in performance. In some algorithms (i.e., NN and DT), simply using the neighbor information leads to the highest accuracy. Involving other information such as histogram or file-tree may blur the model and predication boundary. Considering that the file-tree fingerprint depends on the paths of installation that are sometimes modified by users, neighbor information can be more reliable and general in broader use cases.

In addition to the discovery accuracy, the time for model training and testing are other significant aspects that should be taken into account, especially in some real-time monitoring scenarios, in which discovery results must be returned as soon as possible. From our results, all the combinations of learning algorithms and fingerprint methodologies can finish all 160 tests in less than 0.1s. Notice that this number is almost independent with size of knowledge base in all studied algorithms except for NN. We should note that the test time of NN could increase with increasing labeled sample sizes.

For training on a knowledge base containing 160 classes with 7 samples each, logistic regression has the longest training time, which is around 10-20 seconds depending on the types of fingerprints used. Decision tree with combined fingerprints has training time around 5 seconds. All the other combinations finish training in less than 1 second. Notice that there is no training time issue in nearest neighbor algorithm, as there is no model to be trained. In practice, a discovery system can be designed as a combination of an online training phase and an offline training phase. Algorithms that are able to train and update the model fast, though with slightly lower accuracy can be applied in the online training phase to update the prediction model frequently, while algorithms with longer training time but higher accuracy can be applied as an offline training method, to update the model less frequently with some fixed periods, e.g., once a week.

Related Work

Standard system management and system change discovery mechanisms employed industrially today are mainly rule-based solutions that utilize large sets of manually written rules to check the existence of certain indicative properties, such as the existence of certain files. OpenIOC [5] is one such open framework that uses rules to examine registry, file content and metadata information to determine security vulnerabilities. BigFix [3] is a commercial offering that uses rules to scan systems and applies fixes automatically based on scan results. Rule-based approaches, however, are labor intensive as each new system and software requires a new set of rules, require frequent edits and updates due to updates on systems and/or software packages, and require domain expertise over a variety of systems and applications to prepare the rules, which is hard to come by.

As a complementary solution to manually written rules, a few studies investigate automated learning methods in system performance diagnosis [20, 21]. These studies mainly rely on system performance metrics to detect the performance drift on either hardware or firmware layer, and mostly do not deal with problems in software and system layer. EnCore [22] is a tool developed that learns configuration rules from a given set of sample configurations, and automatically detects software misconfigurations. Though it effectively solves types of misconfiguration problems, it does not target to general software and system changes.

Recently, some work studies the opportunities and challenges to interactively search across virtual machine (VM) images at a high semantic level, and sketches the outline of an implementation by a discard-based search [23, 24]. Alternative system change and software discovery methodologies based on indexing methodologies and information retrieval techniques are proposed. Minersoft [6] indexes file system information to build a keyword-based query processing systems that enables searching for software existence on indexed systems. Similarly, Mirage [25] is an image library that stores cloud images such that their file system structure is indexed in a way that enables scanning, searching and comparison of VM instances. However, indexing-based approaches require maintenance of large indexes per target VM that get constantly updated as the VM evolves. Besides, indexed file names and processes can have repetitive string representations, which can be uninformative and misleading thus results in inconclusive or incomprehensive result sets.

In contrast, our approach (1) is fully automated requiring little to no human intervention; (2) can adapt to changes and updates by learning from the new examples and updating models; (3) significantly reduces the amount of maintenance required due to changes on instances by creating compact representations of changes occurring in system states, and (4) can provide highly accurate and comprehensive results to system change discovery queries.

Furthermore, we note that due to fast development cycles observed in state-of-the-art system implementation practices, many system change discovery use cases require the capability of querying with examples, such as “listing the set of VMs that have made a given type of an installation/configuration”, perhaps to identify systems that pertain a certain type of misconfiguration or bug observed in one VM. We should note that, unlike rule-based or indexing-based approaches, our proposed framework performs nicely in these kinds of “query by example” scenarios as well.

Conclusion and Future Work

As cloud computing technologies continue to mature and keep gaining attractions in many industries, the demand for intelligent analytics solutions that ease the management of cloud environments increases. In this study we have introduced an automated cloud analytics solution that caters to one of such demand, namely system change discovery and management. Our solution achieves efficient discovery by recording system changes in *change sets*, generating compact *fingerprints* of system state changes and utilizing these fingerprints in a machine learning platform. We have shown that with understanding the hidden context and the semantic relationships among filenames in

change sets, automated, fast (in a few milliseconds or seconds) and accurate (up to 98.75%) system change discovery is achievable by our technique.

As an immediate follow-up of this work, we plan to test the accuracy and efficiency of proposed system on more additional cloud environments such as the Google Cloud Engine and Microsoft Azure as well as other popular operating systems such as CentOS, Ubuntu, and RHEL, and prove the scalability of our solution. Besides, investigation of configuration discovery in popular cloud applications such as Hadoop, Spark, RabbitMQ, Cassandra, etc. is a natural extension of the proposed work.

Since most of the machine learning algorithms we investigate can provide a prediction confidence level along with their predictions, confidence threshold setting mechanisms can be investigated in future work to discover new applications that are not in the current knowledge base, as well as reduce the error of miss labeling by filtering out low confidence predictions. Another related possible research avenue is the investigation of prediction accuracy on highly noisy and/or insufficient/partial data. This task can be achieved by applying the confidence threshold setting mechanisms to determine when to make a prediction and when to wait for more input.

References

1. L. Wei, H. Zhu, Z. Cao, X. Dong, W. Jia, Y. Chen, and A. V. Vasilakos, "Security and privacy for storage and computation in cloud computing," *Information Sciences,* vol.258: 371-386, 2014.
2. National vulnerability database. [Online]. Available: http://nvd.nist.gov.
3. Endpoint manager relevance language guide. [Online]. Available: http://pic.dhe.ibm.com/infocenter/tivihelp/v26r1/topic/com.ibm.tem.doc_8.2/Relevance Guide PDF.pdf.
4. Open source software discovery. [Online]. Available: http://ossdiscovery.sourceforge.net.
5. Openioc. [Online]. Available: http://www.openioc.org.
6. 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: 2, 2012.
7. S. Bohner, "Impact analysis in the software change process: A year 2000 perspective," in *Proc. of IEEE International Conference on Software Maintenance*, pp. 42-51, 1996.
8. 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 *Proc. of IEEE International Conference on Big Data (Big Data),* pp. 90-99, 2014.
9. Origami: Systems as Data. [Online]. Available: https://sites.google.com/site/origamisystemsasdata.
10. D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala, "Opening black boxes: using semantic information to combat virtual machine image sprawl," in *Proc. of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE),* pp. 111-120, ACM, 2008.
11. T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. of Workshop at International Conference on Learning Representations (ICLR)*, 2013.

12. T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 3111-3119, 2013.
13. B. Clarke, E. Fokoue, and H. H. Zhang, *Principles and theory for data mining and machine learning*. Springer Science & Business Media, 2009.
14. T. Cover and P. Hart, "Nearest neighbor pattern classification," *Information Theory, IEEE Transactions on,* vol.13, no.1: 21-27, 1967.
15. H. Jr, D. W., and S. Lemeshow. *Applied logistic regression*. John Wiley & Sons, 2004.
16. C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol.20, no.3: 273-297, 1995.
17. C.-W. Hsu, C.-C. Chang, and C.-J. Lin, "A practical guide to support vector classification," Technical Report, Department of Computer Science, National Taiwan University, 2003. [Online]. Available: http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf.
18. L. Breiman, "Random forests," *Machine learning,* vol.45, no.1: 5-32, 2001.
19. A. Bestavros and O. Krieger, "Toward an open cloud marketplace: Vision and first steps," *Internet Computing, IEEE,* vol.18, no.1: 72-77, 2014.
20. P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proc. of the 5th European conference on Computer systems*, pages 111–124, ACM, 2010.
21. P. Xiong, C. Pu, X. Zhu, and R. Griffith, "vPerfGuard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments," in *Proc. of the 4th ACM/SPEC International Conference on Performance Engineering*, pp. 271-282, ACM, 2013.
22. 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," in *Proc. of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS),* pp. 687-700, ACM, 2014.
23. M. Satyanarayanan, W. Richter, G. Ammons, J. Harkes, and A. Goode, "The Case for Content Search of VM Clouds," *The First IEEE International Workshop on Emerging Applications for Cloud Computing (CloudApp '10),* 2010.
24. L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki, "Diamond: A Storage Architecture for Early Discard in Interactive Search," in *Proc. of USENIX Conference on File and Storage Technologies (FAST),* vol.4, pp. 73-86. 2004.
25. G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang, "Virtual machine images as structured data: the mirage image library," in *Proc. of the 3rd USENIX conference on Hot topics in cloud computing (HotCloud'11),* 2011.

Bios

Hao Chen *Boston University ECE Department, Boston, MA 02215 USA (haoc@bu.edu*). Mr. Chen is a Ph.D. student in the Electrical Computer Engineering Department of Boston University (BU). He received a B.S. degree with distinction in Information

Science and Electronic Engineering from Zhejiang University, China in 2010. His research interests include data center energy management and demand response, smart grid, intelligent analytics in cloud system management, big data and machine learning.

Ata Turk *Boston University ECE Department, Boston, MA 02215 USA (ataturk@bu.edu)*. Dr. Turk is a Postdoctoral Researcher in the Electrical Computer Engineering Department of Boston University (BU). He received his B.Sc. and Ph.D. degrees from the Computer Engineering Department of Bilkent University, Turkey. His research interests include information retrieval, data analytics, and combinatorial optimization for performance, energy, and cost improvements in cloud computing applications. Previous to joining BU, he was a postdoctoral researcher at Yahoo Labs.

Sastry S. Duri *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (sastry@us.ibm.com)*. Dr. Duri is a senior software engineer at the IBM T. J. Watson Research Center in Yorktown Heights, New York. He earned a Ph.D. degree in computer science from the University of Illinois at Chicago. His professional interests include distributed and computing systems, mobile commerce applications, radio-frequency identification (RFID) based supply chains, and sensor and actuator applications. In his spare time, he coaches teams for FIRST Robotics competitions. In the past, he represented IBM in the industry standard group EPCglobal Application-Level Events (ALE) Working Group, a subsidiary of the Uniform Code Council (UCC), and in Open Location Services Interface Standard (OpenLS) workgroup.

Canturk Isci *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (canturk@us.ibm.com)*. Dr. Isci is a Research Staff Member in the Cloud Computing Division at the IBM T. J. Watson Research Center, where he leads the Scalable Data Center Analytics team in Research and the Operational Analytics Squad within IBM Cloud Services. His research interests are cloud computing, operational and devops analytics, novel monitoring techniques based on virtualization and containerization, and energy-efficient computing techniques at various computing abstractions, from microarchitectures to data centers. He received a B.S. degree in electrical engineering from Bilkent University, an M.Sc. degree with distinction in VLSI System Design from University of Westminster, and a Ph.D. degree in computer engineering from Princeton University.

Ayse K. Coskun *Boston University ECE Department, Boston, MA 02215 USA (acoskun@bu.edu*). Dr. Coskun is an associate professor in the Electrical and Computer Engineering Department at Boston University. Her research interests include energy-efficient computing, new computer architectures, and management of data centers. She has a Ph.D. in computer science and engineering from the University of California, San Diego. She is a member of IEEE and the ACM.

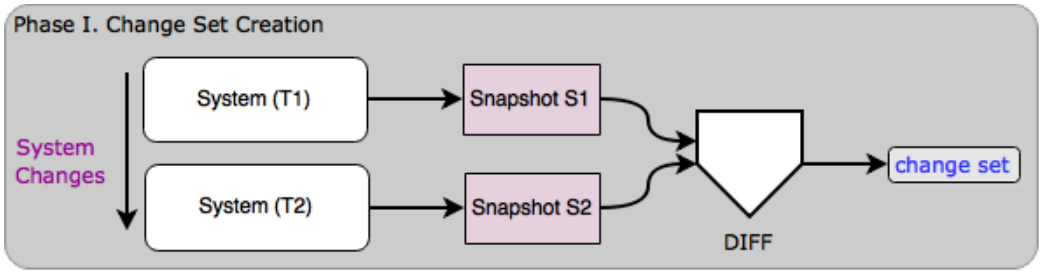
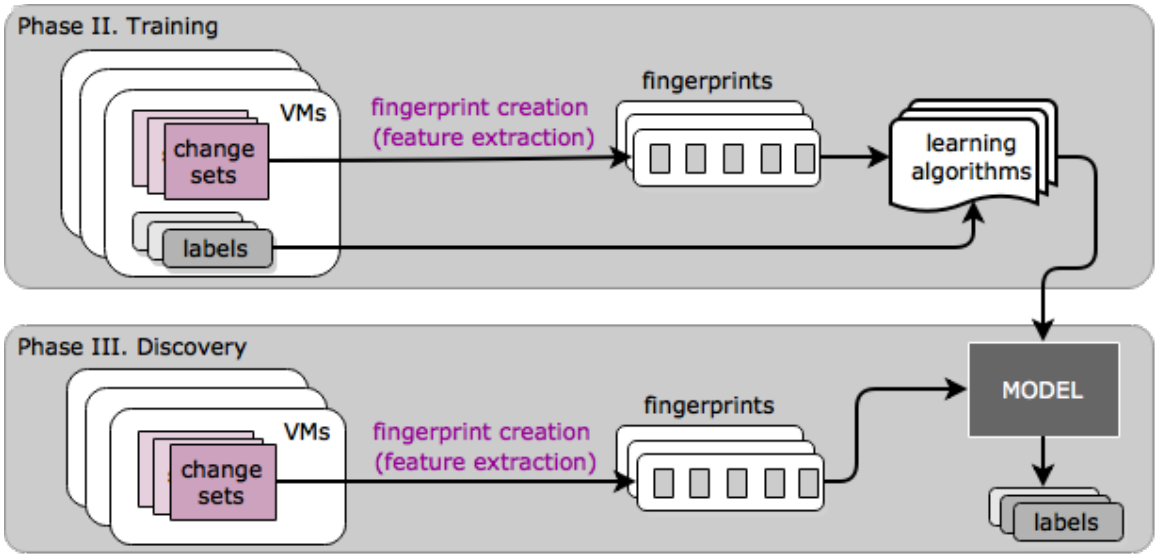


Figure 1 Flowchart of change set creation. Snapshots of the system are captured before and after the system change event. Then a diff operation is calculated on these two snapshots, and the change set is generated.



§

Figure 2 Training and discovery phases of the system change discovery framework. Labels and extracted fingerprints from change sets are input into learning algorithms to train the model in the training phase. The learned model is then used to discover and label the new-coming unidentified changes during discovery.

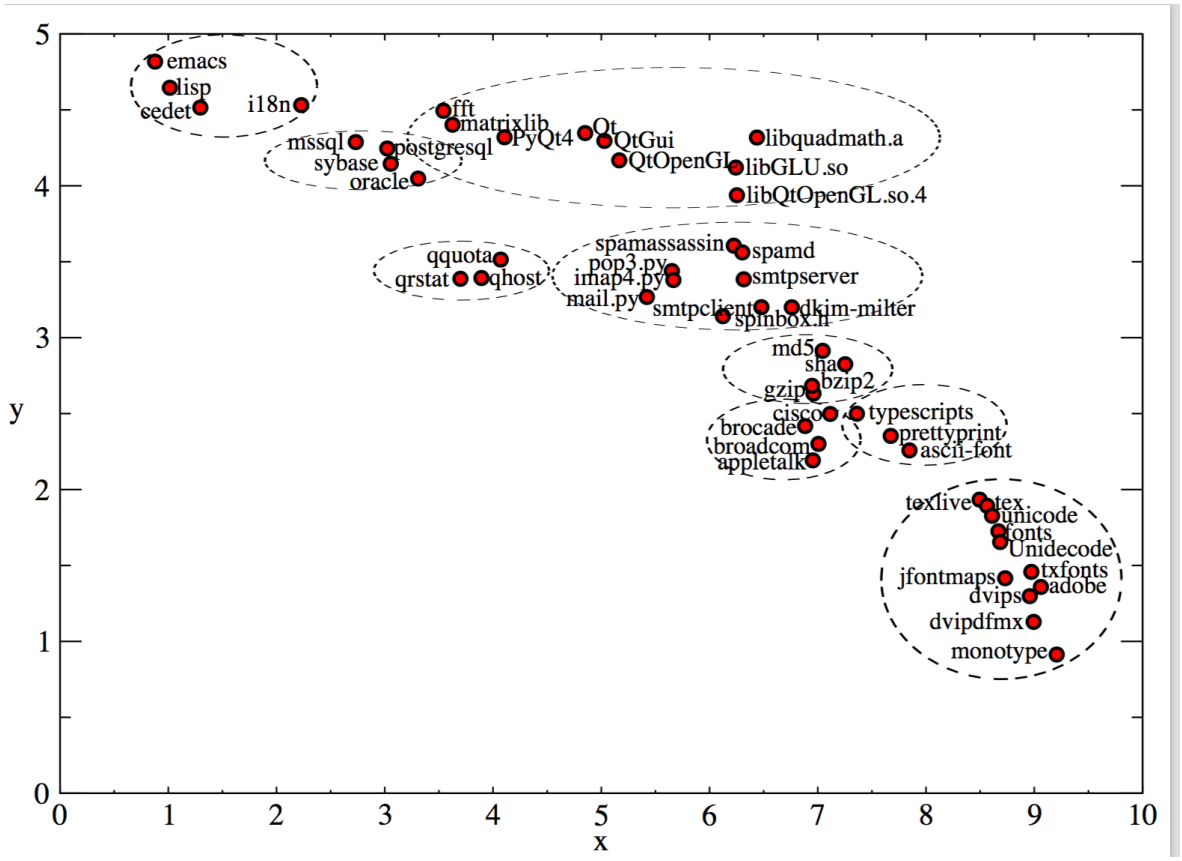


Figure 3. 2D vectors created by w2v for a set of filenames when file tree information is supplied to it. Created vectors retain the semantic relationship among the software objects they represent.

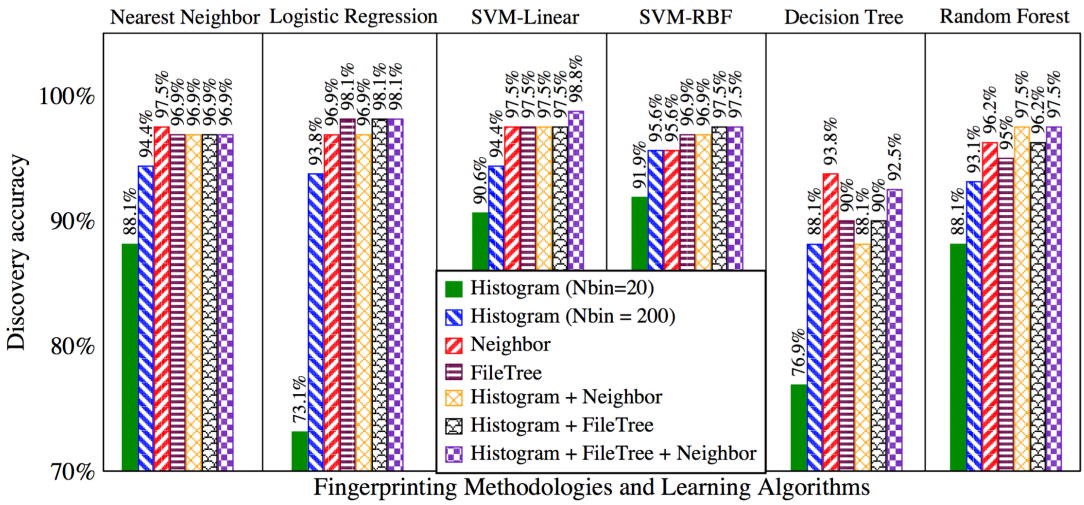


Figure 4. Discovery accuracy for multiple fingerprinting methodologies and learning algorithms. Results are grouped by learning algorithms.

{
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" > ...
}
}

Listing 1: A sample change set. It contains all entities that are created, modified or deleted during the system change event, e.g., OS, files, packages, processes and configurations.