Praxi: Cloud Software Discovery That Learns From Practice

Anthony Byrne, *Student Member, IEEE,* Emre Ates, *Student Member, IEEE,* Ata Turk, *Member, IEEE,* Vladimir Pchelin, Sastry Duri, Shripad Nadgowda, Canturk Isci, and Ayse K. Coskun, *Senior Member, IEEE*

***Abstract*—With today's rapidly-evolving cloud landscape embracing continuous integration and delivery, users of cloud systems must monitor software running on their *containers* and *virtual machines (VMs)* to ensure compliance, security, and efficiency. Traditional solutions to this problem rely on manually-created *rules* that identify software installations and modifications, but these require expert authors and are often unmaintainable. Recently, automated techniques for software discovery have emerged. Some techniques use examples of software to train machine learning models to predict which software has been installed on a system. Others leverage knowledge of packaging practices to aid in discovery without requiring any pre-training, but these practice-based methods cannot provide precise-enough information to perform discovery by themselves. This paper introduces Praxi, a new software discovery method that builds upon the strengths of prior approaches by combining the accuracy of learning-based methods with the efficiency of practice-based methods. In tests using samples collected on real-world cloud systems, Praxi correctly classifies installations at least 97.6% of the time, while running 14.8 times faster and using 87% less disk space than a similar learning-based method. Using a diverse software dataset, this paper quantitatively compares Praxi to systematic rule-, learning-, and practice-based methods, and discusses the best uses for each.**

***Index Terms*—Solution monitoring, software discovery, machine learning, cloud analytics.**

I. Introduction

In today's rapidly changing business landscape, the speed at which software solutions can adapt to new business needs has become a critical factor in determining the success or failure of enterprises. To meet these market demands, developers are often given ample freedom to choose the technologies and components they use to build applications. As a consequence, in tandem with common software usage models shifting from vendor-supplied proprietary software to open-source software [1], there are numerous sources of software components for building applications. Other trends in the cloud, such as serverless computing [2], continuous deployment [3], and two-pizza teams[1] making independent decisions also cause typical software applications to end up with far more component diversity compared to that of software observed just a few years ago.

A. Byrne, E. Ates, A. Turk, V. Pchelin, and A. K. Coskun are with the Department of Electrical and Computer Engineering, Boston University, Boston, MA, 02215 USA. Contact e-mails: {abyrne19, ates, ataturk, vsemp, acoskun}@bu.edu.

S. Duri, S. Nadgowda, and C. Isci are with IBM T.J. Watson Research Center, Yorktown Heights, NY.

Manuscript received November 19, 2018; revised July, 2019.

[1]Amazon CEO Jeff Bezos introduced his famous two-pizza team rule, meaning that teams should not be larger than what two pizzas can feed [4].

Many of these changes have made organizations nimble and more responsive to market needs, but they also present new problems for businesses needing to maintain security or legal compliance. For example, the integration of third-party components often forces organizations to keep track of many different software licenses and security bulletins so that their systems remain in compliance and free of vulnerabilities. The big migration to the cloud by corporate IT departments [5] and the emergence of micro-service architectures have increased application diversity and caused a tremendous explosion in the size and scale of cloud deployments over the past decade. This increased diversity has forced both organizations and individual developers to abandon manual methods of server administration in search of more automated and robust means, including a way of searching for a specific piece of software among a large set of VMs or containers. With rapid development and deployment cycles often moving too fast for proper documentation, it is all too easy to lose track of what software is installed on a system, leading to inadvertent hosting of non-compliant or buggy applications. Awareness of these issues is a must for proper protection of high-value cloud deployments, and with constant iteration, one-time scans are not enough – continuous awareness is necessary for security assurance in the cloud [6]. Therefore, being able to continuously identify what software exists in a given system, i.e., *"software discovery,"* is becoming a core requirement for ensuring security and compliance.

Initial efforts for software discovery began with a basic need to determine whether a given system has any known vulnerabilities. To accomplish this, administrators often wrote rules detecting software that had been cataloged in knowledgebases like the National Vulnerability Database (NVD) [7]. A triggered rule would indicate a vulnerable system, which would then be quarantined and manually patched. Ideally, the next scan of the system would declare the system free of that particular vulnerability. However, this was not always the case: the rules themselves often required manual updating to ensure they recognized the applied patch and suppressed the vulnerability warning. This fragile, labor-intensive nature of rule-based methods became the motivation for more novel software detection systems.

Towards the goal of generating more comprehensive and robust software discovery methods in the cloud, our recent work [8] introduced a "learning by example" approach, which classifies unlabeled software installations using machine learning models trained on fingerprints generated from recorded examples of software installations. Fingerprints can be created

using simple hashes [8], or using more elaborate natural language processing (NLP)-inspired learning techniques [9]. Using this method, we observed software detection success in real-life cloud systems with over 90% accuracy [9]. We also studied how to classify system changes in near-real-time by building a "discovery service," *DeltaSherlock*, which continuously samples the system and analyzes events of interest at fixed intervals [10].

Instead of relying on a corpus-based machine learning mechanism of software detection, another approach known as *Columbus* leverages knowledge of software packaging practices (such as naming conventions) to aid in software discovery without need pre-training [11]. Such a method has the advantage of not having to spend time building a database of known software, but the disadvantage of requiring human involvement to interpret the loosely-defined output strings.

In this paper, we present a novel hybrid method of automated software discovery, *Praxi*, that combines aspects from both learning- and practice-based approaches. Praxi uses Columbus to generate informative tags without requiring any pre-training and then employs fast incremental learning methods to quickly update discovery models as new software packages become available. Praxi significantly reduces training time and the overhead associated with incremental training compared to existing learning-based approaches, while still providing highly accurate software discovery that requires no human involvement. More specifically, contributions of this paper can be listed as follows:

- We propose a novel hybrid discovery method called Praxi that can learn to perform automated software discovery using small numbers of training samples with over 99% accuracy in simulated production settings. Praxi quickly adjusts when the software corpus expands, as it supports incremental training and does not require any pre-training to extract system fingerprints.
- We provide a detailed comparison of Praxi's performance to that of state-of-the-art discovery approaches considering manually- and repository-installed packages. Considering systems that have observed single and multiple software installations, we report accuracy, runtime performance, and storage requirements of Praxi and contrast these with rule-, practice-, and learning-based approaches.
- We also provide an automated method of systematic rule-based software discovery to serve as a baseline for comparison to other discovery methods we evaluate.

The rest of the paper first provides background information about select previous approaches to software discovery in Section II, followed by an explanation of our proposed approach in Section III. We design and execute a set of experiments to evaluate the performance of Praxi in Sections IV and V. We discuss the results of those experiments and their potential applications in Section VI. Finally, we explore related work and conclude with an overview of future directions for our work in Sections VII and VIII.

II. Background

Several solutions to the software discovery problem have been proposed over the years. Some solutions, such as rule-based approaches, are straightforward and easy to implement, but suffer from poor accuracy due to the rigid, fragile nature of rules. Other solutions, such as some learning-based approaches, can achieve very high accuracy but require long, complex training processes. Still others, such as practice-based approaches, solve the rigidity and training problems by eschewing the need for a corpus, but have an output that is geared more towards manual human analysis than automated software discovery.

In this section, we briefly explain the workings of several of these kinds of methods, as they form the basis for our hybrid method, Praxi, as well as their individual advantages and disadvantages.

A. Rule-based approaches

Traditional approach to detect installed software is to manually define rules. System engineers design rules that verify the presence of specific files and predefined properties to detect misconfigurations and system changes [12], [13]. An example of such a rule may state that the existence of a file named `SIGFILESDKXA64.SYS2` with a file size of 100 bytes in a system indicates that `IBM SDK 5.0` is installed. This technique have been used for years to detect intrusions, viruses, and non-compliance with license agreements, among other uses. When well-written, a rule can achieve very high detection accuracy, even in "noisy" system environments where other methods have been shown to struggle.

A rule's effectiveness, however, is totally dependent on the author's ability to consider every possible scenario in which the target (the software being detected) could exist. Continuing the example from above, a patch could change the size of `SIGFILESDKXA64.SYS2` from 100 to 105 bytes, which would prevent the rule from being triggered and would therefore produce a false negative. To prevent this, the rule would either have to be updated by the author every time a patch is released, or be made general enough that it detects the `IBM SDK 5.0` regardless of version without creating additional false positives. Even if an expert rule writer could write such a rule, modern development practices (agile software development [14], continuous integration/continuous delivery [15], etc.) have accelerated today's software release schedules, and inevitably a manually-defined rule will have to be rewritten. When faced with a software catalog numbered in the hundreds of thousands, the human resources required to write rules to discover every version and variation of software that could potentially be installed becomes prohibitively costly.

Despite their disadvantages, the simplicity and ease of implementation of rule-based approaches make it likely that they will remain in use for many years. Thus, we introduce an automated method of rule generation. This approach attempts to locate file-tree segments (i.e., partial or absolute paths) that are unique to individual applications and uses them to build one or more rules that can detect that application. The rules check for the existence or absence of certain files or a combination of files. When enough rules are satisfied for a particular application by a filesystem sample, we conclude that the sample contains an installation of the said application.

TABLE I
DISTRIBUTION OF MYSQL SERVER FILE SYSTEM FOOTPRINT.

Namespace	File Count
/usr/share/man/man1	27
/usr/bin	26
/etc	24
/var/lib/dpkg/info	24
/usr/share/doc	7

The exact algorithm and implementation details are given in Appendix A.

Although this automated method does not require the extensive human effort involved with rule-making, it needs to be executed every time new software is introduced into the corpus, and its algorithmic runtime complexity ($O(m^3n^2)$ where m is the number of unique software labels, and n is the number of unique files across the entire corpus) makes it unsuitable for real-world deployments covering very large numbers of software applications. Nevertheless, we use this automated method whenever we discuss "training" in the context of rule-based methods, and it will serve as a baseline when analyzing experimental results as it is systematic and provides good accuracy.

B. Columbus: a practice-based approach

On the opposite end of the spectrum, with respect to the strict and definite rule-based discovery method, sits practice-based discovery. Unlike rule-based approaches that require rule regeneration after every corpus update, practice-based approaches require no prior analysis and can be applied on any system. However, practice-based approaches do not perform software discovery directly, and the output of this approach can only be used to aid in software discovery, not complete the task itself.

Practice-based discovery aims to exploit the generalized textual system state indicators by relying on the observation that, across most software installation mechanisms, a number of common naming and packaging practices exist. Specifically, most software has a unique filesystem footprint that is typically organized across unique namespaces and has the name of the software embedded into the footprint. By identifying these namespace clusters, it is possible to provide generic identifiers for installed software packages.

Broadly, a software package consists of a set of binaries, libraries, documentation, and configuration files. A typical software installation stores files in the filesystem under a namespace that is unique to that software. A standard practice is to name all the binaries with common software name prefix, such as `mysql`, `mysqld`, `mysqldump`, `mysqladmin` for the software MySQL, while another practice is to store configurations, libraries, logs under a directory named after the software. These naming conventions are no coincidence — they are efforts to ensure easy access to the user and simplify software management.

For example, the filesystem footprint for the `mysql-server` package installed on Ubuntu 16.04 contains 131 files. This footprint is distributed across the different namespaces as shown in Table I. Below are some sample entries in selected namespaces.

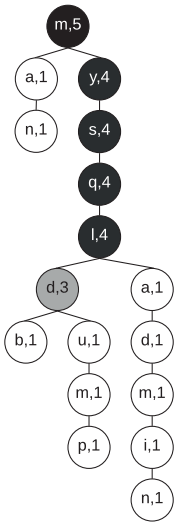


Fig. 1. A frequency trie for the inputs `[man, mysqld, mysqldb, mysqldump, mysqladmin]`. The *non-trivial* tag with the highest frequency is `mysql`, followed by `mysqld`.

```
/usr/share/man/man1/mysql.1.gz
/usr/bin/mysqldump
/usr/bin/mysqloptimize
/usr/bin/mysql
/etc/mysql/conf.d
/etc/mysql/mysql.cnf
/var/lib/dpkg/info/mysql-server-5.7.list
```

Inspired by this most fundamental and deterministic system characteristic of a software, we proposed Columbus [11], a generic software discovery technique that is agnostic to platform and software installation method. Columbus explores the filetree information provided in the form of a list of file paths to discover software installed in systems.

Each filepath is first tokenized to get a list of directory and file names. For example, `/etc/mysql/conf.d` is tokenized into the list `["etc", "mysql", "conf.d"]`. Common system tokens like `etc`, `usr`, etc. are removed from the list. Remaining tokens are indexed into a *frequency trie* or FT described in detail by Nadgowda et al [11], and an example FT is given in Fig. 1. Columbus uses FTs to discover *tags*, defined as the most-frequent longest-common-prefix among the list of all tokens in the filesystem tree. In an FT, when the frequency of any child node is found to be smaller than its parent node, then the trie path from the root to the parent node is concatenated and identified as a new tag with frequency of the parent tail node. Columbus builds two frequency tries: FT_{name} that stores all segments of a file path, and FT_{exec} that stores only the basename of executable files. Tags in each frequency trie are ranked based on their relative frequency. Then only *top k* tags are selected, where *k* is set heuristically. The results are then merged as described by Nadgowda et al [11].

Columbus aids in software discovery by identifying informative tags. Discovery is performed manually by administrators who use these tags to aid the discovery process. Considering the number of cloud instances operated by large organizations, this manual approach is not scalable.

C. DeltaSherlock: a learning-based approach

Our recent work [8], [9], [10] introduced a learning-based method of "software discovery by example" called DeltaSherlock. The main idea of this approach is to observe examples of system changes (mainly changes to the filesystem) made when known software installations take place, generalize these observations into a numerical vector representation, and detect software in different settings using machine learning models trained with these "generalizations." This is achieved by recording all filesystem changes (i.e., creation, modification, or deletion of a file) during the changes caused by software installation to a "changeset", generating condensed numerical "fingerprints" from those changesets using shallow neural networks, and then applying machine learning algorithms to train models based on the generated fingerprints for discovery. Fig. 2 provides a general overview of the DeltaSherlock method (top two rows), and Section III-A provides more details on the changeset recording process.

DeltaSherlock's changeset generalization or "fingerprinting" process is described in 3 parts: generation of an ASCII character histogram, analysis of the relationships between files in a directory, and the analysis of the tree structure of the filesystem.

The first step, generation of the ASCII histogram, places the ASCII numerical codes for all characters of the basenames of every changed file recorded in the changeset into the bins of a histogram and then normalizes, to produce the first 200 elements of the fingerprint vector.

The second and third parts of the fingerprinting process utilize techniques originating in the field of natural language processing (NLP) to capture the "meaning" of the names of these changed files. In these two steps, known as generation of the "filetree" and "neighbor" vector elements, changesets are analyzed using the shallow neural network training tool *word2vec* (w2v), which was originally designed to encode the syntactic and semantic relationships between words in sentences of human natural languages. When creating "filetree" fingerprint elements, the "sentences" fed to w2v are made up of the full absolute paths of each changed file in a changeset. Similarly, during the generation of "neighbor" fingerprint elements, w2v is fed "sentences" made of the basenames of each changed file and its neighbors (i.e files residing in the same directory). After analysis of all changesets in the training corpus has completed, w2v creates a "dictionary" mapping words (file and directory names) to vectors, and this dictionary is used to generate the remaining 400 elements of the fingerprint vector based on the words contained within the original changeset [9].

Concatenating and normalizing two or more of these "elemental" fingerprint types (histogram, neighbor, and filetree) results in a "combined fingerprint." We chose to primarily use the combination of histogram and filetree fingerprinting methods in our experiments for this paper because this combination typically performed better when compared to other fingerprinting methods.

Recently, we extended these fingerprinting and machine learning methods from single software installation discovery to multiple system event discovery, and proposed a DeltaSherlock system daemon that can capture system states on-demand and detect multiple system changes between them [10]. This prior work demonstrated DeltaSherlock's real-world usefulness by testing the system with "noisy" changesets that included multiple application installations randomly spaced throughout one recording interval, which introduced background noise from things like system caching and log rotations. DeltaSherlock was then further upgraded to include additional features like a client/server architecture that enabled distributed changeset collection and processing, enhanced backend database management (for efficient storage of changesets and fingerprints), and a web-based user interface.

Despite its positive performance, DeltaSherlock presents overhead issues with regards to its time and storage requirements. In previous work using DeltaSherlock [10], training an SVM-RBF model with 13,650 changesets took approximately 5 hours and 50GB of disk space. This does not include the time and disk space required to generate and store the changesets themselves, and DeltaSherlock requires constant storage of all training changesets so that the w2v dictionaries and corresponding fingerprints can be regenerated, as incremental training is not supported. In our most recent experiments, although we were able to eliminate some of the overhead associated with DeltaSherlock by eschewing the "neighbor" elements of the fingerprint, we hypothesized that a hybrid method could further reduce overhead and increase scalability while maintaining high accuracy. Thus, resolving these overhead concerns became one of our primary motivations for developing our new hybrid method: Praxi.

III. Praxi: A New Hybrid Approach for Software Discovery in the Cloud

We propose a new hybrid discovery method called Praxi that combines the strong points of practice- and learning-based approaches while alleviating their weaknesses. Our proposed approach replaces the fingerprint generation phase of DeltaSherlock, which requires training systems dictionaries via word2vec, with Columbus, which does not require prior training. It also replaces the machine learning engine used in DeltaSherlock, which does not support working with free-form text as features, with Vowpal Wabbit (VW), which not only supports free form text as features, but also incremental training (enabling minimal-cost model updates when new packages are released) and faster overall training and testing.

This new approach, shown in Fig. 2 (bottom two rows), starts in much the same way as the other approaches covered in this work: by taking in as an input a set of filesystem changes observed during some time period, known as a *changeset*. The changeset is analyzed using the practice-based method, Columbus, which reduces the changeset down to the set of strings (*tagset*) that appear most frequently in the filepaths contained within it. We then use the tagset (labeled with the name of system event that occurred) as features to train a machine learning model in the training phase. Finally, during the evaluation phase, we present unlabeled tagsets to the trained model for classification.

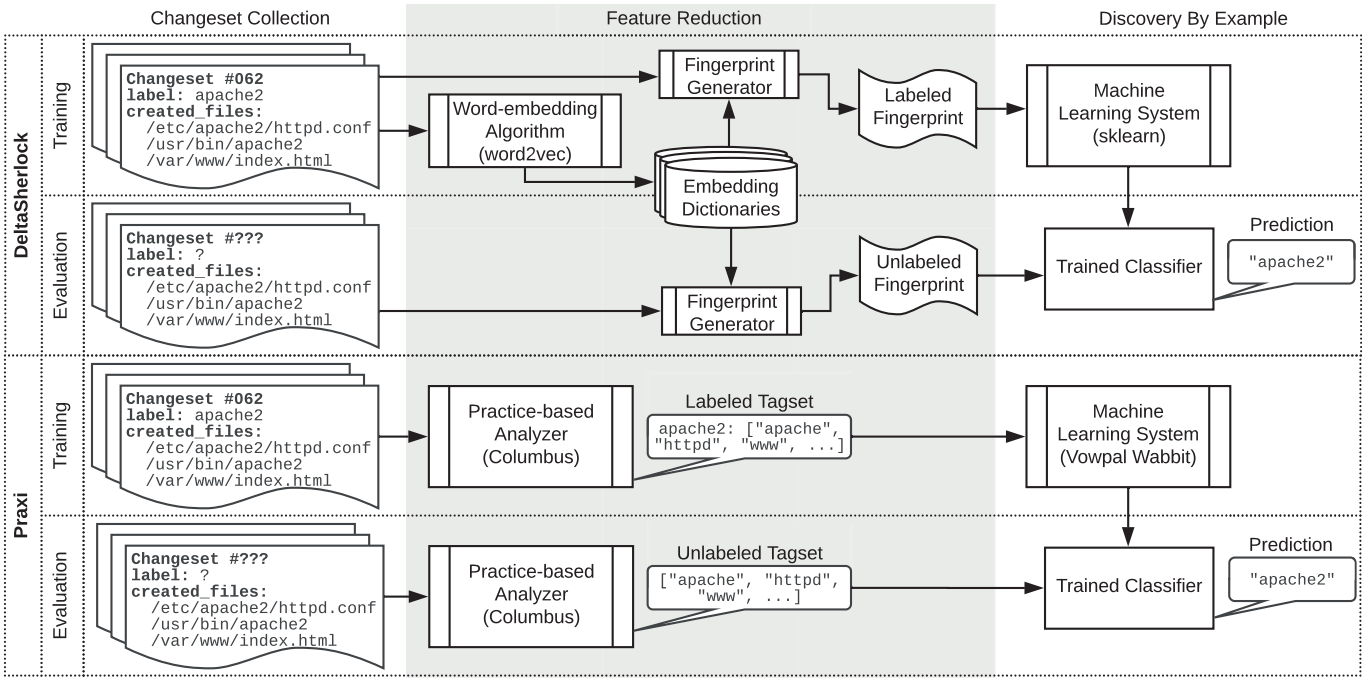


Fig. 2. A comparison of the overall training and evaluation processes for DeltaSherlock and Praxi. Praxi requires fewer steps and less disk space during its training process, and does not require the periodic dictionary regeneration and classifier retraining required by DeltaSherlock.

A. System Change Recording

Similiarly to DeltaSherlock, Praxi operates on changesets — collections of filesystem changes observed within a closed time interval, in which one or many system events, such as an application installation, update, or removal, may have occurred [10]. In our implementation, these changes are collected using the Linux kernel's *inotify* feature, which notifies a change recording daemon running in the background on the target system of the creation, modification, or deletion of any file or directory that has an inotify "watch" placed on it. A graphical overview of this process is shown in Fig. 3. We found that this method introduces the least amount of overhead into the system while ensuring that all changes are captured, but other mechanisms of system change collection could also be used to suit other system configurations, such as the `FindFirstChangeNotification` function present in the Windows API or the periodic filesystem crawl approach used by Turk et al [10].

Upon notification of a filesystem change, the daemon records several attributes to a *changeset record*: the file's absolute path and UNIX permissions octal, the kind of change that occurred (i.e., creation, modification, or deletion), and the UNIX timestamp at which it occurred. The changeset record is then appended to the currently open changeset. In our implementation, this process occurs entirely in-memory in order to reduce disk I/O overhead.

After a certain amount of time or upon some user-configurable event, the recording daemon "closes" the changeset by sorting its changeset records by time of occurrence, removing any duplicate entries, setting the `close_time` attribute, and either saving the changeset to a file on-disk or uploading the changeset to a remote server for processing. The daemon then "opens" a new empty changeset for writing.

B. Practice-based Analysis & Changeset Reduction

After a changeset has been closed, it is analyzed using a practice-based method called Columbus. As detailed in Section II-B, Columbus uses a frequency trie to discover a set of tags made up of the most frequent longest-common-prefix string tokens found in a filesystem tree, relying on the organizational conventions in place among today's software developers and package maintainers. Instead of using Columbus to scan an entire filesystem tree (as it was used by Nadgowda et al. [11]), our approach uses Columbus to scan the "tree" of file-change records within a changeset.

The resulting tagset is therefore representative only of the system changes that occurred while that changeset was open and being written to, eliminating the possibility of noise produced by events occurring before the recording period. Conversely, because Columbus places only the tags that occur more than once in the resulting tagset, noise arising from irrelevant stimuli (such as log file rotations, caching, etc.) during the recording period is also filtered out.

Beyond noise reduction, the practice-based analysis step also offers several practical benefits. By condensing changesets (which vary in size from a few kilobytes to tens of megabytes, depending on number of changes captured) down to tagsets (which are typically less than a kilobyte), Praxi requires much less storage space than other methods. The simple data structure of tagsets (basic space-separated-value strings) also makes them easy to store in both "flat" text file datastores and fully-featured database systems. Further advantages of this analysis and reduction technique are discussed in Section VI.

C. Model Training & Classification

In our evaluation implementation, we chose to use the Vowpal Wabbit (VW) tool [16] to train our classifiers. This tool

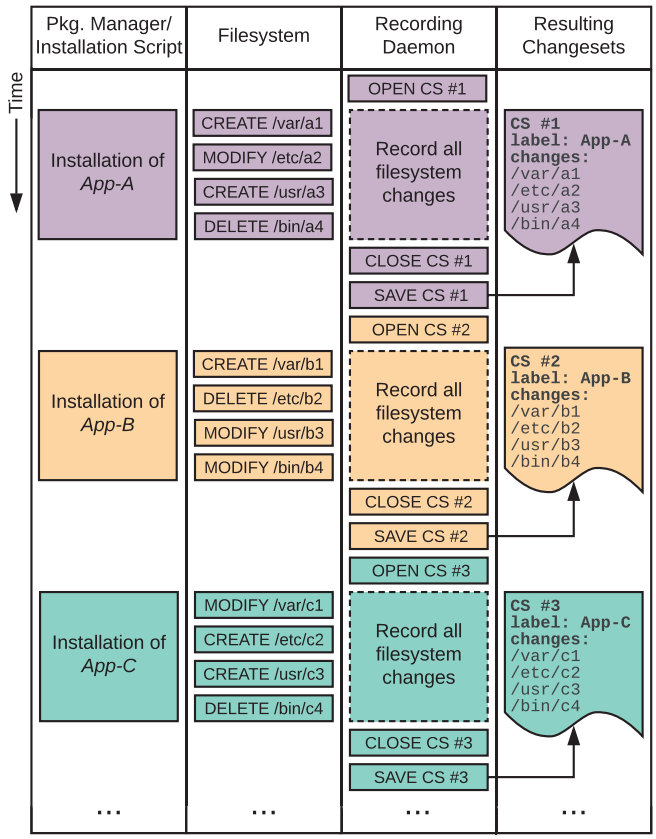


Fig. 3. A general overview of the system change recording process used in our experiments.

was chosen instead of other machine learning engines because of three main advantages: flexibility, speed, and incremental learning capability. In terms of flexibility, the VW tool offers input feature hashing using the *Murmurhash v3* [17] algorithm, allowing the use of variable-length sets of plain-text strings (i.e., tagsets) as input features.

After a tagset has been produced, it is used as a feature to train a machine learning model using sparse gradient descent on a hinge loss function. The tagset is treated as if it was a "bag of words," much in the same way that human-readable sentences are processed by natural language classification models. We use the Vowpal Wabbit tool to generate a single multi-class classifier for our experiments with single-label changesets. For experiments involving multi-label changesets, we use Vowpal Wabbit's cost-sensitive one-against-all (CSOAA) method. Internally, VW's CSOAA classifier evaluates each sample against n regression functions based on sparse gradient descent on a hinge loss function, where n is the number of unique labels (application names) present in the corpus. Each regression function returns a "cost" representing the likelihood of a particular label being present in the sample. A high cost (1) is assigned if the package is not present in the sample, whereas a low cost (0) is assigned if the sample contains that package.

In terms of both model training and sample classification speed, VW vastly outperforms other machine learning engines, as shown in Sec. V-C. Finally, unlike many traditional learning engines, VW supports incremental or "online" learning, where trained machine learning models can be updated with new data without requiring a full retraining. The ability to update models incrementally instead of training from scratch is very critical, considering there are hundreds of thousands of known software packages and with many new packages becoming available on any given day.

In addition to its input format flexibility, the Vowpal Wabbit tool allows for flexible parameterization and performance tuning. To optimize the learning model for the task of classifying tagsets, we parameterized our models based on discrete parameter sweeps and prior experience with machine learning tools. Although we found Vowpal Wabbit to be the most suitable tool for our implementation, any machine learning system that supports variable-length string-based input features could be used in its place.

IV. Experimental Methodology

In order to perform a comprehensive evaluation of Praxi, we have designed and executed a set of experiments demonstrating the method's accuracy, runtime performance, and storage complexity. Several of these experiments offer comparisons against DeltaSherlock, a learning-based software discovery method, while others demonstrate benefits unique to Praxi. In all of these experiments, we use datasets based on popular software installed via source code compilation, manual installation scripts, and pre-built repository packages.

A. System Setup

All of our experiments use the following process (or some slight variation of it, where specified) to prepare the environments in which they will run. To begin, a standard OpenStack `m1.small` VM instance with 1 vCPU, 2GB RAM, and 20GB persistent disk is spun up on the Massachusetts Open Cloud [18] using the latest "cloud image" of our target platform: Ubuntu 16.04 Xenial. The images are updated so that all pre-installed packages are at their latest versions, and then their package managers are configured to be "frozen," meaning that the versions of installed packages will remain consistent throughout our experiments. Our custom experiment controller daemon is then installed and configured to run on boot, allowing us to use the Linux kernel's *inotify* feature to monitor the filesystem for changes with little overhead (as explained in Section III-A).

B. Dataset Generation

We collect changesets using two different experimental configurations: one designed to produce "clean" changesets and another designed to produce "dirty" ones. Clean changesets represent installations performed under conditions ideal for collecting the "purest" possible filesystem footprint of an application. Before clean changesets are collected, the experiment controller executes a "pre-run" where all applications are installed and then removed, leaving their dependencies pre-installed on the system so that the observation process only captures the software application(s) in question.

Dirty changesets, on the other hand, represent installations performed under conditions designed to model realistic filesystem noise and installation practices. Unlike clean changesets, the preparation for the collection of dirty changesets does not include a pre-run step, so many application dependencies are not pre-installed. As a result, dependency installations occur within the changeset recording period, and vary based on the order of the application list used in a particular run.[2]

Finally, we generate changesets that contain multiple application installations by combining two or more single-application changesets.

a) Collecting "clean" changesets: After the experimental environment has been prepared (including the pre-installation of dependencies) and the VM rebooted, the controller places inotify "watches" on the files and directories in the filesystem, excluding special and device files like those typically found under `/proc/` or `/dev/`. The controller then begins recording any file creations, modifications, or deletions reported by the kernel to a changeset.

As soon as recording begins, the first application is installed using the APT package manager (if the application is a repository package) or a vendor-recommended installation script (if the application is a manual installation). As soon as the installation (or installations, if collecting multi-label changesets) completes, the changeset is closed and "ejected" from the controller, meaning that no more changes can be recorded to that particular changeset. The changeset, which now contains a list of all files changed during the installation period, is labeled with the name of the software application(s) installed and sent back to a central server for processing. The controller replaces the ejected changeset with a fresh, empty one, restarts recording, and begins installation of the next package(s) on the list.

This process is repeated throughout the run, which ends when the software application list is exhausted. After a run ends, all applications are uninstalled, the application list is randomly shuffled, and a new run begins. Runs are performed until at least 150 samples of each label are obtained.

b) Collecting "dirty" changesets: The dirty changeset collection process proceeds in much the same way as that of clean changeset, except dependencies are not installed in a "pre-run", nor they are removed between application installations. As mentioned above, this results in dependency installations being captured within an applications changeset. Secondly, random waiting periods between 10 and 30 seconds are inserted before and after a package is installed (during which the controller still records changes), meaning that more random system noise (log rotations, caching, etc.) is captured. Other than these differences, the "dirty" data collection is identical to the "clean" data collection process detailed before.

c) Generating multi-application changesets: Changesets containing two or more application installations are required for evaluating the multi-label classification performance of our method. Instead of creating these changesets "organically" by allowing the controller daemon to observe many installations within a single recording period, we "synthesized" multi-application changesets by concatenating several single-application changesets together in order to save time. We found that these "synthetic" changesets were almost identical to those created "organically" with the same applications.

Each multi-label changeset used in our experiments was synthesized from between 2 and 5 single-application changesets randomly chosen (without replacement) from the overall "dirty" corpus (including both repository packages and manual installations). Controls were put in place to ensure that two or more changesets containing the same application were not included in the same multi-application changeset. A total of 3,000 multi-application changesets were synthesized.

[2]As an example, say application *A* shares a set of dependency packages *D* with application *B*. In one run, *A* is installed before *B*, so the installation of packages in *D* are captured by *A*'s changeset. In a subsequent run, the application list might be shuffled such that *B* is intalled before *A*, so the filesystem changes caused by *D* are instead captured within *B*'s changeset.

TABLE II
NUMBER OF UNIQUE APPLICATIONS IN OVERALL CORPUS

	Apps	Clean C.Sets	Dirty C.Sets
Repository Packages	73	10,950	10,950
Manual Installations	10	1,500	1,500

C. Application Sources

In order to better reflect the wide variety of software installation methods in use today, we selected 83 applications from various sources. These applications can be roughly split into two categories: repository packages and manual installations. The total number of collected changesets for different software categories are presented in Table II.

a) Repository Packages: The repository package list consists of 73 packages that can be found the Ubuntu official default repositories (full list of packages used can be found in Appendix B). The packages in this list are diverse in terms of category, popularity, and installed size. 150 clean and 150 dirty changesets are generated for each target in this list using the standard installation process, resulting in 21,900 total changesets.

b) Manual Installations: The manual installation list consists of 10 applications that aim to simulate the occasions when a user needs to use software that is not included in their distribution's package repository (full list of applications used can also be found in Appendix B). These applications are installed via source compilation, a self-installation script, or some combination of the two. Seven of the applications in the list involve some sort of source compilation step in their installation process. Once again, 150 clean and 150 dirty changesets are generated for each application in this list using the standard installation process, resulting in 30,000 total changesets.

D. Performance Metrics

To measure performance, all experiments are evaluated using the standard measures of precision, recall, and F1 score. To account for variations in number of samples per unique label (class imbalance) in our cross-validated experiments, we use support-weighted macro-averaged F1 scores instead of the

standard micro-averaged F1 measure. That is, we calculate the weighted F1 score for an application `app` as follows:

$$F_1^{app} = 2 \cdot \frac{P^{app} \cdot R^{app}}{P^{app} + R^{app}} \cdot \frac{S^{app}}{T}, \quad (1)$$

where P^{app} and R^{app} are (respectively) the per-label precision and recall for `app`, S^{app} is the ground-truth number of samples of `app` present in the testing set, and T is the total number of samples (of all labels) in the testing set.

To then calculate the average support-weighted F1 score for all labels, we take the simple arithmetic mean of the results of Eqn. 1 for all applications in the testing set as follows:

$$F1 = \frac{\sum_{app \in TestSet} F_1^{app}}{|TestSet|}. \quad (2)$$

V. Results

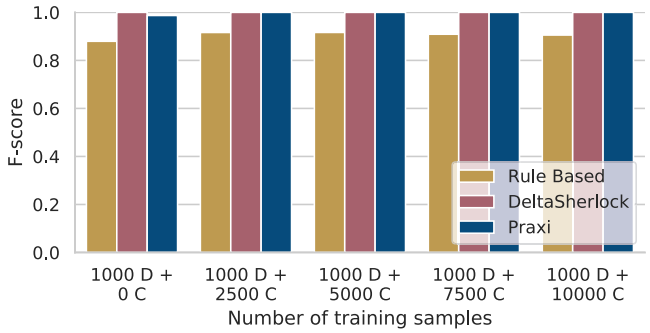
To evaluate the performance of Praxi, we have designed and executed a set of experiments measuring the accuracy, runtime, and overhead of the approach in different scenarios. In all experiments, we compare Praxi's results to that of other recent software discovery methods to provide points of reference for the reader. Unless otherwise noted, all experiments utilize the system setup, dataset generation techniques, and performance metrics detailed in Section IV.

A. Single-label Classification

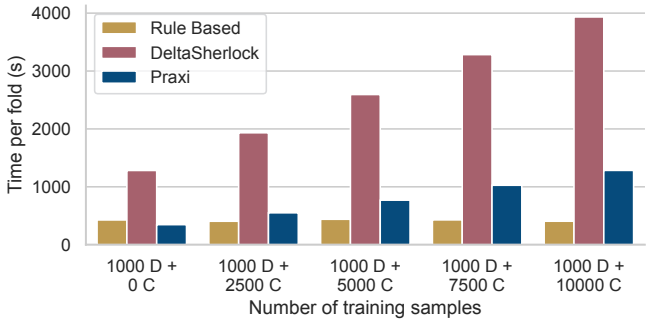
We begin with a comparison of Praxi's single-label classification accuracy to that of two other methods: *DeltaSherlock* and the automated rule-based method detailed in Appendix A. In these experiments, each changeset is guaranteed to contain a single application installation, and all three methods are trained and tested on the same disjoint datasets.

The training set consists of 1,000 dirty changesets and n clean changesets, where n varies between 0 and 10,000 (inclusive) in increments of 2,500. The testing set consists of 2,000 dirty changesets. We perform a modified form of 3-fold cross validation by swapping out which 2,000 of the 3,000 total dirty changesets are used for testing and averaging the results. This methodology was designed to demonstrate each discovery method's performance when the number of training samples per label is relatively low. All changesets were chosen randomly from our overall corpus and contain varying quantities of samples of every repository package and manually installed application. The experiment was repeated 15 times per discovery method (3 folds of dirty changesets $\times$ 5 increments of clean changesets).

Fig. 4(a) displays an F1 score comparison of Rule-based, DeltaSherlock and Praxi for different training dataset sizes. As seen in the figure, Praxi was able to correctly identify the vast majority (98.7%) of changesets without any additional clean training samples. Accuracy improves to 100% after adding just 2,500 samples to help the VW model learn to classify outlier changesets, and stays perfect throughout further additions. In comparison, DeltaSherlock was able to identify all samples accurately in all trials, but as shown in Fig. 4(b), at the cost of significantly longer runtime (more than double of Praxi).



(a) Accuracy of methods.



(b) Runtime of methods.

Fig. 4. The results for single label classification. 'D' refers to the number of dirty changesets, and 'C' refers to the number of clean changesets.

The automated rule-based method achieved a consistently low runtime, but could only reach a peak accuracy of 91% after adding 5,000 additional samples. Interestingly, the rule-based method became less accurate as additional samples were added beyond 5,000. This is due to the fact that the rules generated by the rule-based method become more and more prone to over-fitting as more training data becomes available.

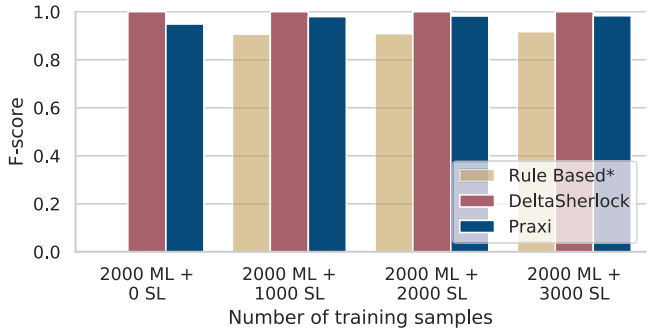
In order to simulate an even more realistic server environment than could be modeled by the dirty changesets detailed in Sec. IV-B, we recorded additional system noise from a variety of sources, including a live web server, an active MongoDB database server, a user's web browser, and a random filesystem noise generation script. We then randomly overlayed samples of this noise atop the original dirty changesets and repeated our experiments, increasing the size of each dirty changeset by 8.8 kilobytes on average. After repeating our single-label experiment with these "dirtier" changesets, we found that Praxi experienced a slight drop in accuracy, with F1 scores ranging from 96.3% without any clean training samples to 99.6% with the addition of 10,000 clean training samples. By comparison, DeltaSherlock and Rule-based experienced almost no changes in accuracy, with still-perfect classification in the case of the former and a peak accuracy of 92% with the same bell-curve-like accuracy trend in the case of the latter.

B. Multi-label Classification

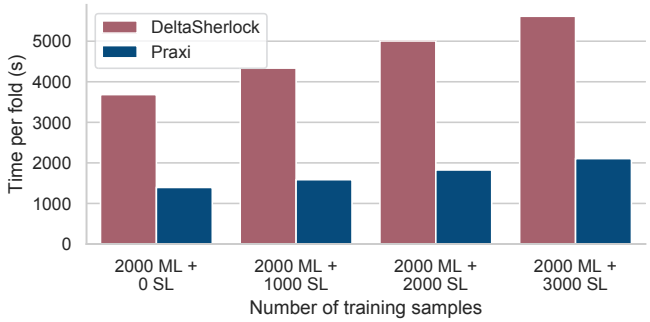
In a typical real-world use-case, a software discovery method will often encounter more than one application installation within an observation period. To evaluate and compare Praxi's performance in this setting, we use synthesized dirty

TABLE III
COMPARISON OF OVERALL OVERHEAD FOR MULTI-LABEL CLASSIFICATION

	Praxi			DeltaSherlock		
Phase	**Operation**	**Time (min)**	**Disk (MB)**	**Operation**	**Time (min)**	**Disk (MB)**
Feature Reduction	Columbus Tag Extraction	3.7	55	Dictionary Generation	13.1	370
				Fingerprinting	55	24
Discovery	VW Model Training	1.5	59	RBF Model Training	11	489
	VW Model Evaluation	0.2	-	RBF Model Evaluation	0.7	-
	Overall	**5.4**	**114**	**Overall**	**79.8**	**883**



(a) Accuracy of methods.



(b) Runtime of methods.

Fig. 5. The results for multi-label classification. The rule-based method cannot be trained using multi-label data; therefore it is only trained using the single-label training samples. 'ML' refers to the number of multi-label changesets, and 'SL' refers to the number of single-label changesets.

multi-application changesets (as defined in Section IV-B) to build the disjoint training and testing datasets.

The training set consists of 2,000 multi-application changesets and n dirty single-application changesets, where n varies between 0 and 3,000 (inclusive) in increments of 1,000. The testing set consists of 1,000 multi-application changesets. We perform 3-fold cross validation by swapping out which 1,000 of the 3,000 total multi-application changesets are used for testing and averaging the results.[3] This methodology was designed to explore the effectiveness of single-label training samples in a multi-label setting.

All changesets were chosen randomly from our overall corpus, and all multi-application changesets contain between 2 and 5 application installations. The ground-truth number of applications in each changeset is provided to each discovery method during testing; i.e., each discovery method is forced to provide the correct number of applications. This part of the methodology accounts for the lack of continuous timestamps in synthesized multi-application changesets, which breaks DeltaSherlock's algorithm for inferring the number of applications present in a changeset by counting local maxima in the number of filesystem changes over time. For real-world applications of these methods, prior work has shown that this quantity prediction algorithm can be used on timestamped changesets containing up to 10 applications with less than 1.6% error, and that overall classification accuracy degrades much more slowly per additional application when quantity data is provided [10]. All in all, we repeated the experiment 12 times per discovery method (3 folds of multi-application changesets $\times$ 4 increments of single-application changesets).

While the general trend of the results, shown in Fig. 5(a), largely matched that of the single-label classification experiments, several interesting exceptions exist. Praxi was able to identify 95% of samples accurately without any single-label examples being present in the training set. Upon adding just 1,000 single-label training samples, however, Praxi's accuracy jumped to 98%, while adding additional single-label samples beyond the first 1,000 had essentially no effect. In comparison, DeltaSherlock again traded significant runtime for perfect accuracy, as shown in Fig. 5(b). The rule-based method does not support multi-label training samples, so it could not be used for the first trial involving no single-label training samples. Once single-label samples are introduced, however, its performance becomes similar to what was observed in Section V-A, with an average accuracy of 91%.

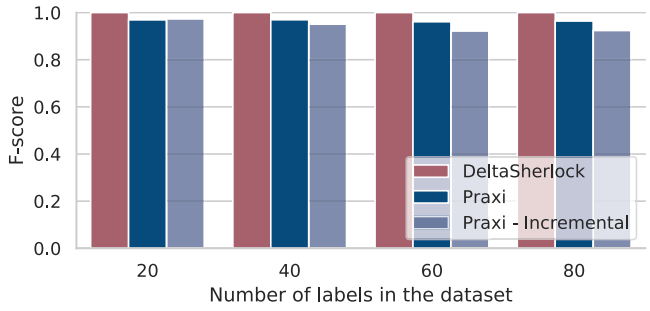
C. Runtime and Disk Usage

In order to promote adoption and scale to thousands of applications and cloud instances, a software discovery method must minimize its associated overhead. Table III compares the runtime and disk usage of Praxi and DeltaSherlock while performing the multi-label classification experiment described in Section V-B on an m1.xlarge virtual machine with 8 vCPUs and 16GB RAM. As shown in the table, Praxi is able to run the overall experiment 14.8 times faster[4] than DeltaSherlock while using 87% less disk space. A large portion of these savings is due to elimination of a dictionary generation step in the feature reduction phase. Trained Vowpal Wabbit models are also significantly smaller than RBF models, leading to additional savings in the training step.

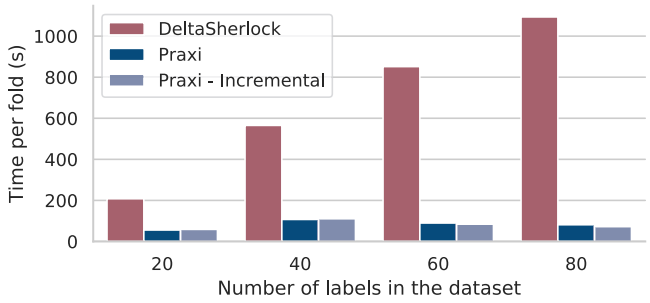
It is also worth noting that it's not necessary to run both the Columbus tag extraction and the w2v dictionary generation

[3]Note that this cross-validation process is different from the one used in Section V-A, where the larger fraction of the corpus was used for testing instead of training

[4]The runtime shown in Table III for the "VW Model Training" operation does not include time spent loading Columbus tags from disk. Columbus tags are small enough that they could reasonably be kept in memory between the tag extraction and model training operations, negating the need for disk-caching in a production implementation of Praxi.



(a) Accuracy of methods.



(b) Runtime of methods.

Fig. 6. The results for incremental classification. Both methods have good accuracy as new labels are introduced but Praxi runs significantly faster.

every time models are to be retrained. In a production environment, w2v dictionaries would be regenerated upon addition of applications into the corpus that touch files that have yet to be touched by any other application in the corpus. Columbus tagsets, on the other hand, can be generated individually as changesets are collected, since there is no interdependence between two tagsets from different changesets. In other words, Columbus tagsets *never* have to be regenerated. They are only iteratively generated as new changesets are added, making the cost of adding new applications to the corpus significantly lower than that of DeltaSherlock and the rule-based method.

D. Scalability Analysis

As mentioned in Section III-C, Praxi was designed with the ever-accelerating speed of software development in mind. Today's software discovery methods must be able to quickly incorporate newly-written and updated software into its corpus. Praxi handles these cases using incremental training, and to evaluate Praxi's performance in this area, we designed a set of experiments that increases the number of applications in the corpus from 20 to 80 in increments of 20. For each increment, we add 20 dirty single-label changeset samples of each application into the training set, and 10 of the same into the testing set. At every increment, we measure the accuracy and overhead of three approaches: Praxi trained incrementally, Praxi trained from scratch (i.e., the VW model is completely retrained, with no online training), and DeltaSherlock (which did not support incremental training at time of writing) trained from scratch. We cross-validated our results three-fold by rotating which 10 changesets were used for training, running the experiment three times, averaging results across the 12 runs. Results are shown in Fig. 6.

Fig. 6(a) shows that incrementally-trained learning-based discovery is possible at the expense of a slight decrease in accuracy. Adding the first 20 new labels to the dataset caused a 3 percentage point drop in Praxi Incremental's accuracy, and successive iterations caused further drops. Overall though, Praxi Incremental's accuracy never drops below 92%. On the other hand, the accuracy of standard Praxi (with full retraining) and DeltaSherlock remained largely constant throughout the experiment. As seen in Fig. 6(b), Praxi runs significantly faster and scales better as more labels become available.

VI. Discussion

Our goal in this paper is to compare and contrast different classes of software discovery methods, identify the best aspects of each (depending on the problem or use case) and combine these best aspects into our "hybrid method" Praxi. Table IV summarizes our observations of this comparison, which we detail in this section.

Practice-based methods such as Columbus do not require training and can discover applications that have never been "seen" before. They use packaging and naming conventions to discover software. In our experiments, Columbus discovered tags such as `mail` and `onecloud`. These are not applications in traditional sense (i.e., not distributed together as a package), but rather the name given to a collection or category of applications installed together. That Columbus manages to discover them shows its semantic discovery capability. However, the output of practice-based methods is not well-defined (i.e., a unstructured set of strings). For this reason, practice-based software discovery cannot be fully automated, as human intervention is required to interpret the output.

Rule-based methods are straightforward and easy to understand. In the past, rules had to be written by human domain experts, making it impractical to write and maintain rules for a large number of frequently-updated applications. In this paper, we provided an automated mechanism for rule generation, which removes this constraint. The main drawback of rule-based approaches, even automated ones, is that they cannot "learn" generalized definitions, only rigid heuristics. We observed that this could lead to over-fitting on larger datasets, often in the form of rules looking for unreliably-present cache or log files, decreasing overall accuracy.

Learning-based methods, especially those that use intelligent fingerprinting techniques to learn system context in a concise manner, are great for generalizing observed characteristics of software packages and identifying them in different contexts in a fully-automated manner. They resolve the over-fitting concerns present in rule-based methods, and the manual effort requirements of practice-based methods. However, pre-existing learning-based methods required regeneration of all fingerprints and full retraining when new applications were introduced to the corpus. This, when combined with long training times, made these methods difficult to maintain.

All of the discovery methods mentioned share a common feature: they operate on changesets, or filesystem deltas with

TABLE IV
HOLISTIC RELATIVE COMPARISON OF AUTOMATED DISCOVERY METHODS

	Praxi	DeltaSherlock	Rule-Based
Classification Accuracy	High	Highest	Fair
Model Training Time	Low	High	Lowest
Overall Disk Usage	Low	High	Low
Can Iteratively Train?	Yes	No	No

known discrete start and ending times. In the experiments detailed in this paper, the start and end times were controlled so as to not fall in the middle of an application installation and create "partial changesets" that would only contain some of the filesystem changes caused by an application. Prior work has found that most discovery methods perform poorly when used to classify partial changesets, often because an installation event can be split across changeset bounds in such a way that neither the preceding nor the following changeset contains enough information to uniquely identify the application [10]. Of course, in most real-world applications, the precise timing of system changes is not known, but can be inferred using the same algorithm used for quantity prediction discussed in Section V-B. That is, if a peak in filesystem change frequency is encountered at the end of one changeset, the system could infer that a similar peak at the beginning of the next changeset would necessitate merging and analyzing the two changesets as one. Alternatively, the software agent responsible for recording filesystem changes to changesets could monitor change frequency and only stop recording after a set period of inactivity.

This issue aside, having noticed the strengths and deficiencies of existing discovery methods, we set out to design a fully-automated method that would combine the high accuracy of learning-based methods with a practice-based fingerprinting approach to greatly reduce overhead and improve performance over existing methods. Furthermore, we found that utilizing incremental-training-ready machine learning methods allowed us to avoid the time-consuming process of retraining the models from scratch every time a new application is discovered. That being said, incremental training does not come without a cost: as detailed in Section V-D, each successive incremental training led to a slight drop in accuracy when compared to a "cleanly retrained" Praxi model. The largest drop in accuracy occurred after the first incremental training, which doubled the number of labels in the corpus. Each successive incremental training caused a smaller and smaller accuracy loss, which leads us to believe that the drop in accuracy is directly related to the percent increase in the number of labels in the corpus caused by each training increment.

In most software discovery applications that would utilize incremental training, the number of labels added by each increment would usually be quite small compared to the number of labels already present in the corpus, just as the number of new packages added to a software repository each day is usually quite small compared to the number of packages already present in the repository. Thus, it is reasonable to assume that in practice, the drop in accuracy caused by each incremental training would be quite small. Nevertheless, we acknowledge that even small successive drops in accuracy may eventually compound into an undesirably large error over time. As such, we would advise users of Praxi to occasionally perform a full retraining after multiple successive incremental trainings (e.g., one weekly full retraining would be recommended if a user was performing daily incremental trainings).

In summary, we believe Praxi achieves our goals by trading a slight decrease in accuracy (compared to learning-based methods) for significantly faster runtimes, lower overall disk usage, and the ability to incrementally train models when new applications are discovered.

VII. RELATED WORK

The global software development ecosystem is quickly expanding as developers focus on using pre-built, often open source components in their products and services instead of solely relying on in-house or specialized proprietary solutions. This increase in "software diversity" creates problems for system administrators who are in charge of keeping all their software components secure and well-performing. Researchers have been exploring several automated approaches for discovering and classifying software systems. In this section we explore previous work broadly in software and systems analytics in the cloud as it relates to our work.

Of the approaches we consider in this paper, rule-based approaches are the usual choices for discovering and identifying system changes in the cloud [13], [19], [20]. Traditionally, system experts manually 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. Installation logs and caches can be used as hints to a package's existence on a system, and the file change manifests provided by some package managers can help designers find files unique to a single package.

As the maintenance of rules has to be performed by system experts with specific knowledge of systems and packages, maintaining rules under the aforementioned breakneck pace of release cycles and large-scale system deployments is expensive and impractical. 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, a phenomenon that can occur multiple times per week for some packages. In this paper, we propose and demonstrate an automated rule generation approach that attempts to mitigate many of these issues by reducing the effort needed to create a new rule. However, the inflexibility of rules and the over-fitting nature of rule-based approaches still stands and we showcase that Praxi significantly outperforms rule-based approaches in discovery accuracy.

There is a separate body of work that focuses on detecting which applications are executing in the system [21], [22], [23]. While these methods may be used for discovering executing software, our approach can detect any installed software and does not need the software to execute first.

System discovery problems have been addressed in contexts other than software discovery as well. Several prior

studies [24] investigate the system anomaly detection and diagnosis problem by performing comparisons with existing examples of a "healthy" or "base" system. In these approaches, registry entries and system event logs have been used in troubleshooting methods that identify problems on a given system. Instead of focusing on discovering software directly, other works [25] focus on data provenance, or discovering the process by which data is formed. While these approaches are appropriate for some applications, not all systems produce or accept large amounts of data, making it less effective to track data instead of the producing software directly.

Several tools exist to monitor the runtime behavior of cloud applications. One example known as CLAMBS [26] monitors several runtime attributes such as CPU & memory utilization, uptime, and running services to monitor the health of applications in a cloud-platform-agnostic manner. CLAMBS, like most tools, will only provide insight into applications it is configured to monitor, and does not discover new software without additional configuration.

While Praxi's primary goal is to discover newly-installed software, it could conceivably assist in the task of cloud forensic data analysis, since it provides a method for identifying filesystem activity over time. Tools such as ForenVisor [27] serve a similar purpose, recording much more detailed forensic data from more sources (including hardware I/O modules and raw memory) at the cost of needing to add a hypervisor layer to the system stack.

EnCore [28] learns configuration rules from a given set of sample configurations, and then automatically detects deviations from these "norms" as software misconfigurations.

Somewhat similarly to our practice-based methods, Minersoft builds an inverted index file-tree structure using file metadata, and uses the file-tree to discover software [29], 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 [30], [31].

Another system change discovery approach we compare against is a recently-introduced learning-based method, also known as "discovery by example" [8], [32], [33]. The state-of-the-art discovery by example system is called DeltaSherlock [10]. 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. Furthermore, since this approach automatically extracts abstracted semantics using the whole changed metadata set, it avoids sticking to any piece of a specific feature, which makes this approach more robust to slight tweaks than rule-based approaches.

The learning models in DeltaSherlock can be updated with newly collected data to automatically keep pace with software and systems updates, but these updates require full regeneration of fingerprints and retraining of models, a slow and storage-intensive process. While DeltaSherlock and its predecessor learning-based systems offer fully automated discovery, their inability to update themselves with new data and their long training time requirements render them unfit for today's fast paced application development environments. Praxi on the other hand, offers incremental training, significantly outperforms DeltaSherlock in terms of runtime, and requires significantly less storage.

VIII. Conclusion

We proposed Praxi, a framework for scalable and fast software discovery in cloud instances. Praxi combines the strong points of previously proposed practice- and learning-based approaches while alleviating their deficiencies to offer a fully automated discovery solution that does not require extensive learning for feature reduction and prediction. Furthermore, unlike any of the existing solutions, Praxi supports incremental learning. This enables fast, accurate, incremental updating of the discovery framework and avoids the need for re-training from scratch when new applications are discovered.

Our experimental analysis revealed that Praxi can provide over 97.6% accurate discovery with 14.8 times lower runtime and 87% lower storage overhead than its counterparts. In future work, we hope to explore to possibility of Praxi detecting and differentiating between individual versions of software, as well as further performance improvements in "noisy" environments where multiple changes occur at the same time.

Acknowledgment

This project has been partially funded by the IBM T.J. Watson Research Center. The authors thank Dr. Hao Chen for his innovative work on applying "learning by example" to the software discovery problem and for the many discussions that led to this paper. The authors also thank Sadie Allen for her contributions to the experiments described in this paper.

References

[1] S. Sijbrandij, "How Open Source Became The Default Business Model For Software," https://www.forbes.com/sites/forbestechcouncil/2018/07/16/how-open-source-became-the-default-business-model-for-software/, 2018.

[2] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[3] G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software technology*, vol. 57, pp. 21–31, 2015.

[4] A. Atlas, "Accidental adoption: The story of scrum at amazon. com," in *Agile Conference, 2009. AGILE'09.* IEEE, 2009, pp. 135–140.

[5] Rightscale, "Cloud Computing Trends: 2018 State of the Cloud Survey," https://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2018-state-cloud-survey, 2018.

[6] S. Lins, S. Schneider, and A. Sunyaev, "Trust is good, control is better: Creating secure clouds by continuous auditing," *IEEE Transactions on Cloud Computing*, 2016.

[7] "National vulnerability database," *http://nvd.nist.gov/.*

[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 *Proceedings of IEEE International Conference on Big Data*. IEEE, 2014, pp. 90–99.

[9] 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.

[10] A. Turk, H. Chen, A. Byrne, J. Knollmeyer, S. S. Duri, C. Isci, and A. K. Coskun, "Deltasherlock: Identifying changes in the cloud," in *IEEE International Conference on Big Data*, 2016, pp. 763–772.
[11] S. Nadgowda, S. Duri, C. Isci, and V. Mann, "Columbus: Filesystem tree introspection for software discovery," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, April 2017, pp. 67–74.
[12] "Endpoint manager relevance language guide," *https://github.com/bigfix*.
[13] "OpenIOC," *http://www.openioc.org/*.
[14] A. Cockburn, *Agile software development*. Addison-Wesley Boston, 2002, vol. 177.
[15] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Boston, 2011.
[16] J. Langford, L. Li, and A. Strehl, "Vowpal wabbit: Online learning project," 2007. [Online]. Available: http://hunch.net/~vw/
[17] A. Appleby, "Murmurhash3 hash function," 2017. [Online]. Available: https://github.com/aappleby/smhasher
[18] A. Bestavros and O. Krieger, "Toward an open cloud marketplace: Vision and first steps," *IEEE Internet Computing*, vol. 18, no. 1, pp. 72–77, Jan 2014.
[19] "Open source software discovery," http://ossdiscovery.sourceforge.net.
[20] "OSS discovery developer guide," http://ossdiscovery.sourceforge.net/WritingProjectRules.pdf.
[21] S. Zander, T. Nguyen, and G. Armitage, "Automated traffic classification and application identification using machine learning," in *The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05)l*, Nov 2005, pp. 250–257.
[22] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu, "Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices," in *USENIX Workshop on Health Information Technologies*. USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/healthtech13/workshop-program/presentation/Clark
[23] E. Ates, O. Tuncer, A. Turk, V. J. Leung, J. Brandt, M. Egele, and A. K. Coskun, "Taxonomist: Application detection through rich monitoring data," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 92–105.
[24] 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, no. 4, pp. 375–388, Apr. 2006. [Online]. Available: http://doi.acm.org/10.1145/1218063.1217972
[25] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proc. of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 405–418.
[26] K. Alhamazani, R. Ranjan, P. P. Jayaraman, K. Mitra, F. Rabhi, D. Georgakopoulos, and L. Wang, "Cross-layer multi-cloud real-time application QoS monitoring and benchmarking as-a-service framework," *IEEE Transactions on Cloud Computing*, 2015.
[27] Z. Qi, C. Xiang, R. Ma, J. Li, H. Guan, and D. S. Wei, "Forenvisor: A tool for acquiring and preserving reliable data in cloud live forensics," *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 443–456, 2017.
[28] 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," *SIGPLAN Not.*, vol. 49, no. 4, pp. 687–700, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2644865.2541983
[29] M. D. Dikaiakos, A. Katsifodimos, and G. Pallis, "Minersoft: Software retrieval in grid and cloud computing infrastructures," *ACM Trans. Internet Technol.*, vol. 12, no. 1, pp. 2:1–2:34, Jul. 2012. [Online]. Available: http://doi.acm.org/10.1145/2220352.2220354
[30] 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*. ACM, 2010, pp. 111–124.
[31] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 105–118.
[32] C. Isci and V. Bala, "Origami: Systems as data," 2014, tutorial presented at ASPLOS. [Online]. Available: https://sites.google.com/site/origamisystemsasdata/asplos2014
[33] G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang, "Virtual machine images as structured data: The mirage image library," in *Proc. of USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'11. USENIX Association, 2011, pp. 22–22.

Anthony Byrne is an undergraduate student with the Department of Electrical & Computer Engineering at Boston University. His current research interests include operating systems and applications of machine learning in cloud systems.

Emre Ates is a Ph.D. candidate with the Department of Electrical & Computer Engineering at Boston University. He received his B.Sc. in electrical and electronics engineering from the Middle East Technical University, Turkey. His current research interests include automated analytics methods for large-scale computing systems.

Ata Turk is a Research Scientist with the Electrical & Computer Engineering Department at Boston University. His research interests include big data analytics and combinatorial optimization for performance, energy, and cost improvements in cloud computing applications.

Vladimir Pchelin is a Ph.D. candidate in the Department of Electrical & Computer Engineering at Boston University. He earned his M.S. in applied mathematics at the University of California, Davis, and his B.S. in physics at Saint-Petersburg State University, Russia. His research interests include machine learning and distributed systems.

Sastry Duri is a Senior Software Engineer at the IBM T.J. Watson Research Center in Yorktown Heights, NY. He earned a Ph.D. in computer science from the University of Illinois, Chicago. His professional interests include distributed and computing systems, mobile commerce, and sensor and actuator applications.

Shripad Nadgowda is a Senior Software Engineer at the IBM T.J. Watson Research Center in Yorktown Heights, NY. He earned his M.S. in computer science from the Stony Brook University, New York. His research interests include storage management, distributed systems, and virtualization.

Canturk Isci is a Senior Software Engineer at the IBM T.J. Watson Research Center in Yorktown Heights, NY. He received his M.S. in VLSI system design from the University of Westminster, and his Ph.D. in computer engineering from Princeton University. His research interests include virtualization, data center energy management, and microarchitectural and system-level techniques for workload-adaptive and energy-efficient computing.

Ayse K. Coskun is an Associate Professor with the Department of Electrical & Computer Engineering at Boston University. She received her M.S. and Ph.D. in computer science and engineering from the University of California, San Diego. Her research interests include energy-efficient computing, management of cloud and data centers, HPC system analytics, and computer architecture.