Columbus: Filesystem Tree Introspection for Software Discovery

Shripad Nadgowda[1], Sastry Duri[1], Canturk Isci[1] and Vijay Mann[2]

[1]IBM T.J. Watson Research Center
[2]IBM Research-India

Abstract—**Software discovery is a key management function to ensure that systems are free of vulnerabilities, comply with licensing requirements, and support advanced search for systems containing given software. Today, software is predominantly discovered through querying package management tools, or using rules that check for file metadata or contents. These approaches are inadequate as not every software is installed through package managers, and agile development practices lead to frequent deployment of software. Other approaches to software discovery use machine learning methods requiring training phase, or require maintaining knowledge bases. *Columbus* uses the knowledge of the software packaging practices that evolved over time, and uses the information embedded in the file system impression created by a software package to discover it. *Columbus* is able to discover software in 92% of all official Docker images. Further, *Columbus* can be used in problem diagnosis and drift detection situations to compare two different systems, or to determine the evolution of a system overtime.**

I. Introduction

Software discovery is a critical function of platform management and operations. Especially for PaaS cloud providers, who are exposing their platforms to run customer applications, virtual machine (VM) or container images. It is important to know software contained in those images to ensure that software is free of known vulnerabilities, complies with licensing requirements. Today various cloud and software vendors provide images for various software and runtime distributions that are packaged ready-to-run on cloud. Some examples to these are Amazon Machine Images (AMIs) and Docker images among several other offerings. Despite the emergence of a diverse and prolific image economy for cloud (such as DockerHub[1], Amazon AWS Marketplace[2], IBM Bluemix[3]), the existing capabilities provided for defining and discovering the contents of these image distributions remain primitive and highly inadequate. In most cases, images come with bare minimum descriptions, which can be misleading, error-prone and allow very limited options for content search and discovery. Techniques for deep image content inspection and software discovery can greatly enhance the identification and usability of images in cloud.

Software is installed using package management tools, or built from downloaded sources, or by copying pre-built binaries. Figure 1 presents a taxonomy of popular package management tools. As you can see, each OS distribution, and sometimes language distribution has its own package managers. Often more than one package management tool is used to install software in a system, and in some cases a package can be installed using different package managers. For example, in an *Ubuntu* system, *tomcat* is installed using *apt*, while *BeautifulSoup* python package can be installed using either *pip* or *easy_install*. Having so many methods of packaging and installing software makes their discovery complex.

In *Columbus* we explore generic software discovery techniques that are agnostic to platform and software installation method. It is inspired by the most fundamental and deterministic system characteristic of a software: each software has a unique filesystem impressions, these impressions are typically organized across unique namespaces and more generally has their name embedded into these impressions. And we argue that this characteristic is consistent, standard and unique across different OS platforms (like Ubuntu, RHEL etc.) and installation method used (eg. agt-get install, dpkg -i, make/make install etc.). Thus, *Columbus* does not require any machine learning phase or knowledge database or fingerprint matching tools. It can discover name of the software right from a given filesystem impression. This is one of the important highlights of *Columbus* as it overcomes the disparities in environment or tools used to install that software and ensures that a single solution can be used for discovery.

II. Background and Related Work

One approach to software discovery involves querying repositories created by package management tools [4], [5]. However, this approach cannot detect software installed straight from sources. In another popular approach rules based on file attributes or contents are used to discover software [6], [7], [8], [9]. Main drawback of rule based systems is that rules require continous maintenance as software evolves and there is no standard governing how to write rules or how to package so that rule writing becomes easy [8]. Machine learning techniques for software discovery are explored in[10], [11]. Machine learning approaches can be considered as a generalized rule based approach in the sense that these approaches use most if not all file metadata added during software installation. However, techniques explored in [10], [11] work only file system changes caused by installation, not file system state.

In our view, Minersoft[12] comes closer to *Columbus*. It creates *Software Graph* from file system metadata that captures

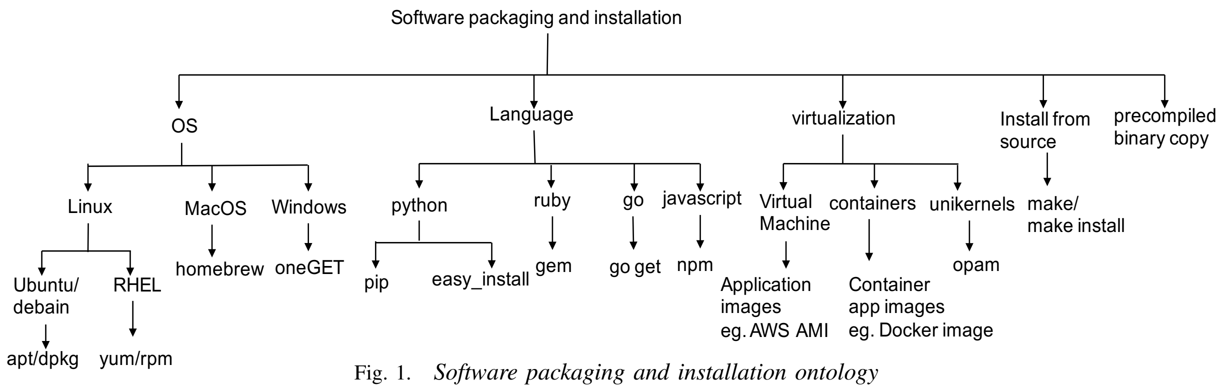


Fig. 1. *Software packaging and installation ontology*

structural as well as semantic relationships that exist between different files using heuristics and operating system specific tools like *yum, ldd, dpkg*. The Software Graph is used to create inverted index to facilitate search. *Columbus* uses file system metadata and knowledge of software packaging practices for discovery. Experiments show that *Columbus* discovers software frameworks which are not packages by themselves but consist of multiple packages, for example, *wordpress*, *owncloud*. Since no operating system specific tools are used this approach could work across different operating systems. Further, this approach lends itself for incremental discovery, that is, to discover what is installed since last scan.

III. System Design

In *Columbus* we have made a deliberate decision to split the software discovery objective into two incremental but independent functions viz. **software name discovery** and **software version discovery**. The number of software packages, their supported platforms, varied package managers keeps growing. In such situation, this separation of concerns helps us design general purpose solution for discovering software names and a specific solutions for discovering their versions. In this section we discuss overall system design for *Columbus*.

A. Filesystem tree introspection

Filesystem is the most fundamental computing paradigm that is used to organize, store and access data efficiently. And it is maintained as hierarchical tree with each node being a directory or file. Every subtree in this filesystem tree is a namespace identified by name of its root. Some standard shared namespaces on linux are log-namespace `/var/`**log**, executable binary namespace `/usr/`**bin**.

Typically software has different data files associated with it like configurations, libraries, binaries, log files etc. When it is installed, it stores these files in the filesystem tree by creating it own namspace or in a shared namespace. We identify these changes made by software as **filesystem impression** of software.

For example, when `rabbitmq` is installed, its filesystem impression includes:

```
(1)/etc/default/rabbitmq-server
(2)/usr/sbin/rabbitmq-server
(3)/usr/lib/rabbitmq/lib/
(4)/usr/lib/rabbitmq/bin/
(5)/usr/share/doc/rabbitmq-server/
(6)/usr/lib/rabbitmq/lib/
rabbitmq_server3.6.4/ebin
```

As we can see it creates its own namespace to store libraries `(3,6)`, documentation `(5)` and uses shared namespace to store configuration `(1)` and binary `(2)`.

Another important characteristic observed is that software often have their name embedded in filesystem impressions. In the above example, all namespaces created `[rabbitmq-server(1,2,5), rabbitmq_server3.6.4(6), rabbitmq(3,4,6)]` contain name of software `rabbitmq` with some variations in it.

B. Software name discovery as a pure function of filesystem meta-data

For brevity we like to define couple of terminologies used in the discussion below, namely $token$ and tag. $Token$ is essentially the raw name of every node in the filesystem tree i.e. directory or file. Following the `rabbitmq` example, $tokens$ generated for filepath (4) are `[usr, lib, rabbitmq, bin]` Thus, $tokens$ are the most fundamental filesystem metadata unit used by *Columbus*. $Tags$ on the other hand are discovered by *Columbus* after processing all $tokens$. $Tags$ are **most-frequent longest-common-prefix** amongst list of all tokens in the filesystem tree. For example, from list of tokens for all 6 impressions above 'rabbitmq' is discovered as a tag. *Columbus* uses two techniques to discover tags namely a) *namespace tagging* and b) *executable file tagging*. The only difference between the two approaches is - in former token list is created from complete **file paths** of every impression while in later case it is created only from the **executable file names** in the impression. All common system tokens e.g. `[usr, bin, etc]` are then removed from token list and remaining tokens are further processed and split with common name delimiters[−,:,_]. These tokens are finally aggregated based on their longest prefix match which is marked as tag and these $tags$ are then ranked based on their frequency count. In section IV, we present our new data structure called $frequency - trie$ that is used to optimally discover and rank $tags$ from potentially very large list of tokens.

C. Software version discovery

Since *Columbus* works only from the filesystem metadata, it is able to discover software version if it is encoded in metadata. For example some numeric version identifier appended as suffix to the tokens. Thus for executable like $apache2$, $php5$ or namespces like $/usr/lib/rabbitmq/librabbitmq\_server-2.7.1$; $/usr/lib/python2.7$ it can discover version number and associate with discovered tag. But, as we observed many software have their precise version number either stored in VERSION, README files in their namespace or in their executable binary itself. *Columbus* currently does not access file contents and is not able to discover version information for such software.

IV. Implementation Details

This section discusses implementation details of *Columbus*. It requires two inputs from user. First, a filesystem tree representing either full filesystem or specific impression of given software. And second input is token blacklist, which is used to filter common system tokens like [`usr`, `lib`, `var`, `local`] etc. to improve precision and optimize performance. The token blacklist we used in our experiments was computed by running *Columbus* on vanilla platforms of Ubuntu 14.04 LTS, Centos 7, Oraclelinux, Windows 7 and their results for *top* 100 tags are combined and blacklisted.

Figure 2 shows architecture of *Columbus*. Primarily there are three major components namely $Filesystem\ Metadata\ scanner$, which generates fileset, $Token\ processor$ which tokenizes file paths from fileset and filters *blacklisted* tokens, and $Tag\ discovery$ which identifies potential software tags.

A. File Metadata scanner

File metadata scanner traverses a given filesystem tree and creates a $fileset$. A $fileset$ consists of a set of files and directories and their metadata —absolute path, and permissions. Linked files (symlink, hardlink) are included in $fileset$, but links are not followed to the originals because during a typical software installation links are used to make shared libraries on the system available in software's own namespace. Since in *Columbus* our emphasis is on namespace analysis for discovery, not including linked files from different namesapce into $fileset$ helps maintain accurate and consistent reflection of filesystem impression and avoids duplicates. Sample $fileset$ for $nginx$ software impression would include:

```
dir:/etc/nginx:666
file:/etc/nginx/nginx.conf:600
file:/usr/sbin/nginx:776
```

B. Token Processor

Token processor consumes the $fileset$ generated by scanner, and produces two Trie instances, one for namespace and another for executable files. In **Tokenizer** module filepath is split into tokens based on (system specific) path-separator and stored into a list. It generates $tuple : (token-list, isExecutable)$ for every member of $fileset$. If a file has executable permission for any user, $isExecutable$ flag is set to $true$ otherwise it is set to $false$. Tokens containing any name delimiters (:, $-$) are further split into multiple tokens. For the nginx example $fileset$, this output will be as follows:

```
(["etc", "nginx"], false)
(["etc", "nginx", "nginx.conf"], false)
(["usr","sbin", "nginx"], true)
```

These tuples are then forwarded to **Filter** module which removes blacklisted tokens from $token-list$. In our example, it will remove `etc`, `usr`, `sbin` from token-list in each tuple and forward them to indexer.

Indexer module reads each tuple, traverses the token list and indexes every token to FT_{name} or *frequency trie for namespace*. And for tuple which has $isExecutable$ parameter set to $true$, it indexes $only$ the last tag from list (i.e. base name of the executable file) to FT_{exec} or *frequency trie for executable*. Following our example, all 4 $nginx*$ tokens get indexed into FT_{name}, but only one ["nginx"] from third tuple gets indexed into FT_{exec}.

C. Tag discovery

In *Columbus* two primary goals are: 1) discover all longest prefix matches a.k.a. $tags$ from the list of tokens 2) order and rank $tags$ based on their frequency. To this end we designed our new data structure called $frequency\ trie$ or FT, by extending basic trie data structure and its operation. Trie[13] is a popular information retrieval data structure used in applications like partial match, auto complete, longest prefix match etc.

Frequency Trie Node: A standard trie node contains data storage for one character of the key, a flag representing if the current node is end of key and array of pointers. Each pointer in the array represents possible next character of keys. In *Columbus*, this basic data structure is extended with additional fields. Each node in FT is structured as:

```
typedef struct TrieNode{
    char data;
    bool isLeaf;
    int frequency;
    bool isTag;
    Node* children[POSIX_CHARSET_SIZE];
}Node;
```

We allow every POSIX file name character in FT node. Further we store two additional parameters with each node, namely $frequency$ count and boolean $isTag$ indicating whether there is a tag terminating at this character node. Utility of these new fields is discussed below.

Frequency Trie Insert: Unlike traditional applications, we do not use FT for search but only operation we ever perform is $insert$. We have extended insert operation with two additional steps namely *frequency count update* and *tag discovery*.

During each token insertion, for every character in the token, frequency count of existing trie node is incremented and for new node count is initialized to 1. In Fig.3(a), for first token (`mysql`), every trie node is set to frequency count 1, and when

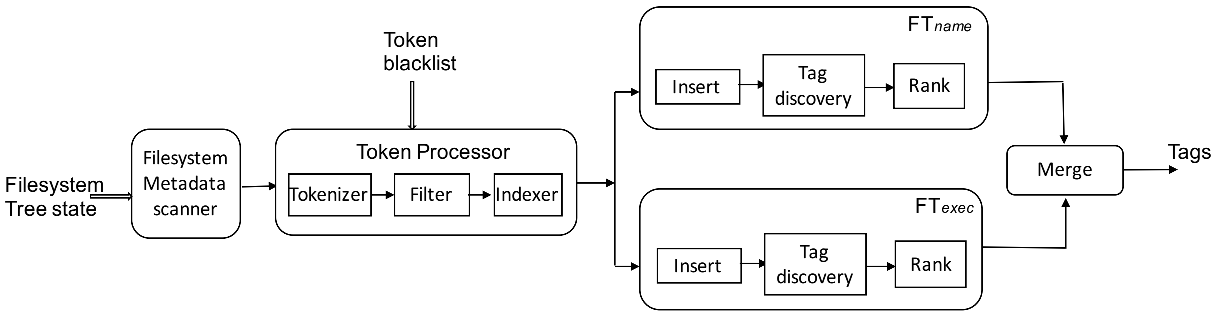


Fig. 2. *Columbus Architecture*

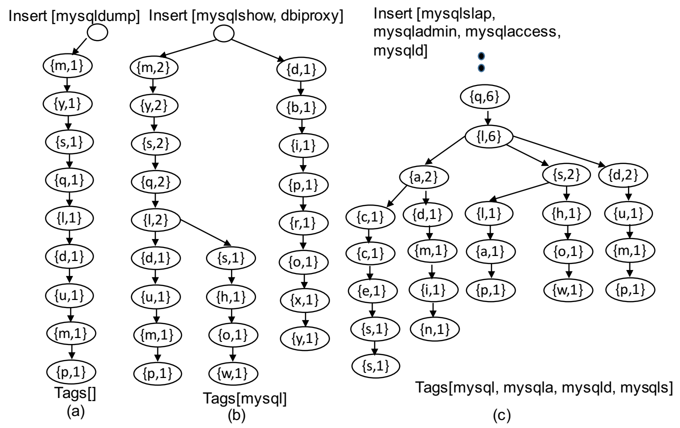

Insert [mysqldump]
Insert [mysqlshow, dbiproxy]
Insert [mysqlslap, mysqladmin, mysqlaccess, mysqld]
Tags[]
(a)
Tags[mysql]
(b)
Tags[mysql, mysqla, mysqld, mysqls]
(c)

Fig. 3. *Frequency Trie data structure. For each node only (data, frequency) is shown.*

second token (`mysqlshow`) is inserted(Fig.3(b)), frequency count for first five common nodes {m,y,s,q,l} is incremented to 2. One important thing to note is in *Columbus*, there is no ordering enforced for token insertion. Thus if the same set of tokens are inserted in different order, the intermediate states of frequency trie could be different. But final state of the $trie$ and frequency count of respective nodes are always same.

Tag discovery is performed by comparing the frequency count of current trie node (after update) to it's immediate parent node. If $frequency_{parent} > frequency_{current}$, then starting from root node, concatenation of characters from every node along the path till parent node is identified as a tag. And frequency count of tag is same as the frequency count of this parent node. Parent trie node is updated to set $isTag$ flag to $true$. And newly discovered tag is added to a map along with $frequency\ count$.

As shown in In Fig.3(b), when new token (`mysqlshow`) is inserted into frequency trie, at it's sixth node (`mysql"s"how`), since it's frequency count {s,1} is smaller than it's parent node {l,2}, concatenation of characters till parent (`"mysql"`) is discovered as a tag with frequency count of 2.

Fig. 3(c) shows progression of this sample FT after inserting more tokens. As we can see there are multiple $tags$ discovered along the same insert path e.g [`mysql, mysqla, mysqld, mysqls`]. To mitigate this we enforce certain policy rules for $tag\ discovery$. For example, (a) minimum $length$ (b) minimum $frequency$ count. Default policy used is $length >= 2$ and $freqeuncy >= 3$. Only when a node satisfies these policies, their $isTag$ flag is set to $true$ and a valid tag is discovered. Under these policies no tag will be discovered in Fig.3(b) and in Fig.3(c), single `mysql` tag will be discovered with count 6.

Another important policy implemented specifically for FT_{name} is that only one tag can be discovered along any given file path. For example consider this $fileset$:

/usr/share/java/tomcat-jasper.jar
/usr/share/java/servlet-api-2.5.jar
/usr/share/java/tomcat-jdbc-7.0.56.jar
/usr/local/tomcat/webapp

After tokenizing and insertion of first three file paths, *Columbus* only discovers single tag [`java`]. 'tomcat' is not discovered yet, since it is nested in the other parent namespace. Although, frequency count of every trie node of 'tomcat' is kept updated. When fourth file path is inserted, since now 'tomcat' appears in its own namespace, it is identified as a tag. This optimization helps reduce candidate tags.

Each frequency trie maintains two counts separately namely $token_{count}$ and tag_{count}. $token_{count}$ is total number of tokens inserted into that trie. And tag_{count} is the frequency count of the last trie node of a tag. E.g. tag_{count} for 'mysql' tag would be the frequency count of trie node mysq"l". Then a tag_{score} is computed as:

$$tag_{score} = \frac{tag_{count}}{token_{count}} \qquad (1)$$

Output of each frequency trie is a list of $tags$ ordered by their tag_{score}. Finally, $topK$ results from FT_{name} and FT_{exec} trie are merged. During merge first common tags are merged-sorted and then remaining unique tags are merged-sorted to produce final $topK$ output of *Columbus*.

Other applications of frequency trie: Frequency trie can be used to obtain other operational insights into cloud. For example, we can compute and compare frequency trie of two similar systems to identify differences between them for fault detection. Likewise, we can compute and compare frequency trie of same system at different times to identify changes in its namespace.

Elk	**Name-space**	node (12.23)	kibana (4.99)	logstash (4.1)	jruby (2.65)	gems (3.95)	babel (0.98)	ruby (0.6)	python (0.58)	aws (0.43)	elastic (0.39)
	Executable	index (4.19)	python3 (1.15)	jquery (0.8)	babel (0.48)	java (0.38)	elastic-search (0.38)	acorn (0.38)	uglify (0.33)	node (0.28)	logstash (0.24)
owncloud	**Name-space**	owncloud (8.66)	apps (4.07)	perl (1.21)	punic (0.8)	apache (0.53)	aws (0.48)	swift (0.27)	symphony (0.26)	rackspace (0.25)	memcache (0.14)
	Executable	apache2 (1.29)	perl (0.9)	check (0.6)	-	-	-	-	-	-	-

TABLE I
Columbus scorecard. Discovered $tags$ along with their tag_{score} in % are shown

.

V. Experimental Evaluation

We performed discovery experiments on images in Docker Hub[1], a public repository accessible to all. Docker Hub contains public images in excess of 100k. A Docker image consists of several layers, and is built using Dockerfile. Each line in the Dockerfile encapsulates filesystem changes caused by commands in that line in a separate read-only layer. Thus we naturally get filesystem impression by accessing individual layers of an image. These images are constructed using various OS distributions, with alpine, debian, Ubuntu and Centos being more popular. We also observed that often software is installed using standard package managers like apt, yum, gem, npm, and pip. In some, software is built from sources using tools like make/make install. In other cases, software is installed by simply copying pre-built binaries. We predominantly used the images pulled from DockerHub. In some cases, however, we built docker images using modified versions of Dockerfile to create layers with desired characteristics to evaluate *Columbus*. We ran these experiments on an Ubuntu 14.04 LTS VM with 16 GB memory and and a 300GB. We used docker client/server version 1.10.2 with btrfs as storage backend.

A. Evaluation criteria

For comparing and validating the result of *Columbus* we needed ground truth about what software packages are installed in selected images. We used `docker history`, and Dockerfile to construct ground truth. Docker history, however, does not contain information about software packages that are installed as dependencies. The goal for *Columbus* is to discover atleast as many software packages as possible using docker history alone. For discovered tag to be a success it should match exactly with the expected software package name. Thus, when *Columbus* returned $couch$ as a tag, we consider it a failure as the software package is named $couchDB$.

We conducted three sets of experiments to evaluate *Columbus* namely $Micro\ Experiments$, $Macro\ Experiments$ and $Case\ Study$. For each set of experiments we ran *Columbus* to discover top 10 software $tags$.

B. Micro Experiments

In this set of experiments our objective is to evaluate discovery scope and system properties of *Columbus*. We selected two software suites viz. ELK(elasticsearch, losgatsh, kibana) stack and owncloud(online collaboration and file sharing framework). And instead of individual layers, we ran *Columbus* on whole filesystem tree of these images. The results for top 10 $tags$ discovered by *Columbus* are shown in Table I. Few important observations here are:

(a) *Columbus* discovered all expected software $tags$. In addition, it discovered $tags$ for dependent packages and third party tools. In ELK image, along with elasticsearch, logstash and kibana, it also discovered java, jquery, and node. In owncloud image, along with apache, swift, and memcache, it also discovered client libraries for aws, and rackspace.

(b) *Columbus* can discover software suites such as *owncloud or wordpress* which are not technically software packages. Such discovery is not possible through standard package manager queries. For example, $owncloud$ is a collective name for the complete software suite rather than a single software package. Since this suite creates its own filesystem namespace for organizing its files *Columbus* is able to discover it in FT_{name}. Further, as there's no runnable for it, it's not discovered in FT_{exec}.

(c) Relevant tag discovery is feasible even in the case of large filesystem impressions. In this experiment, *Columbus* analyzed complete filesystem which often is larger than single docker layer. For example, for ELK image, the size of filesystem was $900MB$ containing $85K$ files and directories. with around $4K$ executable files. For this case *Columbus*, generated total of $761K$ tokens and after filtering inserted $574K$ & $2K$ tokens into FT_{name} and FT_{exec} respectively. Figure 4 shows distribution tag_{score} for top 200 & 100 $tags$ for FT_{name} and FT_{exec} respectively. We observed same pattern in owncloud case. As we can see this is a long-tail distribution and the number of tokens contributing to tag discovery are typically small. Also in Table I we can observe even for discovered top ranked $tags$ their score is low. For example in case of ELK image, `elasticsearch` is discovered in FT_{exec} even with a low score of 0.38%. This essentially proves the fundamental and most generic hypothesis for *Columbus*.

(d) Table II shows memory overhead of our Frequency Trie data structure. FT_{exec} has relatively low memory footprint as compared to FT_{name}. The increase in the memory footprint is not linear with number of tokens, primarily because large number of trie nodes gets shared and we only increase their frequency count in such scenarios.

C. Macro Experiments

In this set of experiments, our objective was to establish usefulness of *Columbus* as a general purpose solution. We considered two software namely `mysql` and `haproxy`. These

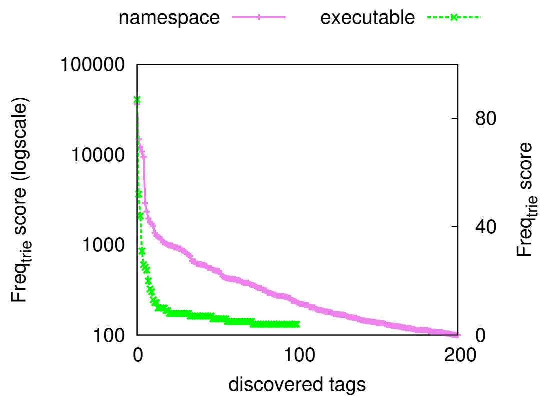


Fig. 4. *Micro Experiment*

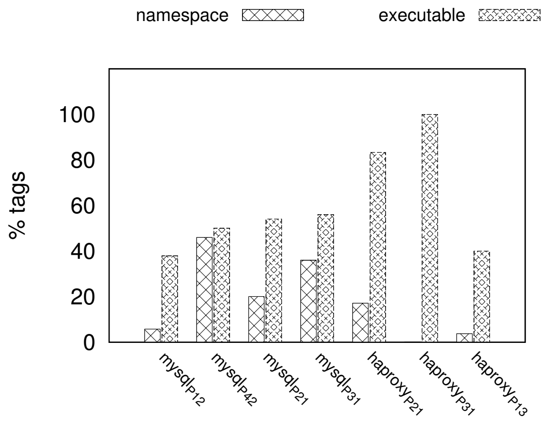


Fig. 5. *Macro Experiment*

	# tokens		Size(MB)	
	namespace	runnable	namespace	runnable
ELK	574K	2154	286	14
owncloud	234K	1146	208	9

TABLE II
Frequency Trie Memory Overhead

software are installed using four different installation methods on three different platforms as shown in Table III. Then, we ran *Columbus* discovery and validated that it is able to discover $tags$ for `mysql & haproxy` as top ranked $tags$. Their individual report is shown in Fig. 5.

	Ubuntu	Oraclelinux	Centos
yum-install	-	P12	P13
apt-install	P21	-	-
make-install	P31	P32	P33
source-copy	P41	P42	P43

TABLE III
Platform/Install Path Matrix

Few important observations here are: (a) when software is installed with package managers such as yum, apt, its filesystem impression includes namespaces for its dependent packages and executable scripts for installation of every package (e.g. .preinst, .postinst, .prerm, .postrm for apt). As a result tag_{score} in FT_{name} is relatively very low when compared with tag scores in corresponding FT_{exec}. For example, namespace trie scores for : $mysql_{P12}$ is 6% , $mysql_{P21}$ is 20%, and 4% for $haproxy_{P13}$. But at the same time, corresponding scores in executable trie are 37% for $mysql_{P12}$, 54% for $mysql_{P21}$ and 83% for $haproxy_{P21}$. Thus, scores from both the tries compliments each others and for high confidence tag discovery, we take both scores into account.

(b) when software is installed from source, the filesystem impression it creates contains files specific to software: header files, source files, and binaries. Thus we observe high tag_{score} for discovered $tags$. For example, for $haproxy_{P31}$, it creates a single $haproxy$ binary, thus tag_{score} is 100% in FT_{exec} but low 1% in FT_{name}. On the other hand for $mysql_{P31}$ filesystem impression holds man pages, test-cases, libraries and executable binaries. As a result, we see high score for both FT_{exec} and FT_{name}. Similarly for $mysql_{P42}$.

Thus, although we observe variations in the filesystem impressions for same software across platforms and installation paths, they still exhibit namespace localization and with *Columbus* we are able to parse and discover precise $tags$ for them.

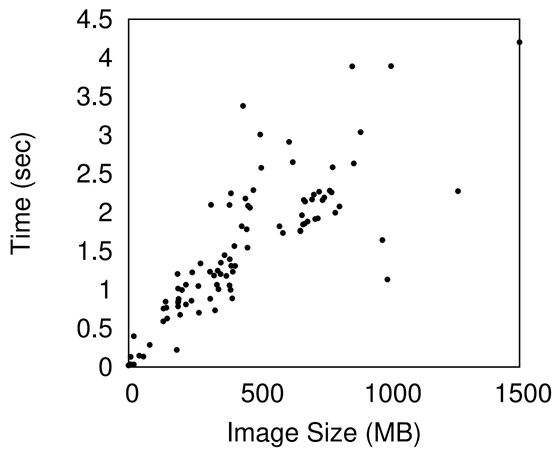
D. Case study

In this case study, we evaluate *Columbus* on all 115 docker official images to assess how it performs in real-world situations. We observed that these images were built using 13 different OS distributions with debian 8 (66%) and ubuntu 14.04(7%) being the popular ones. Size of the images varied from 7MB to 1.5GB with median being 400MB and average 452MB. These images are built leveraging various installation methods including `apt-install, dpkg -i, yum install, npm install, pip install, gem install, make install, binary copy etc.`

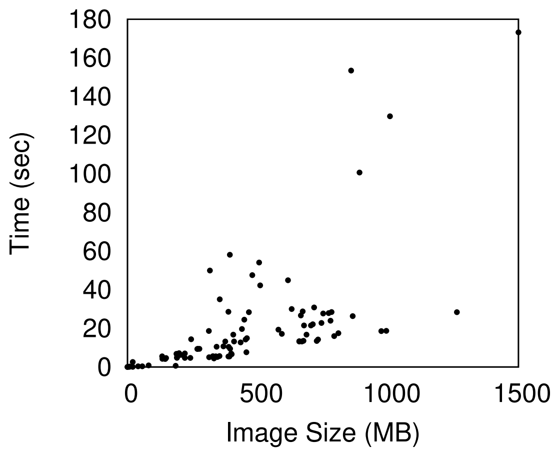
Then for every image, we ran *Columbus* for every layer of that image and validated that discovered $tags$ are matched with software name. We were able to discover $tags$ for 92% of the images. Discovery failed primarily for basic platform images like `clearlinux, vmware photon etc.` and ones which just contains 2-3 files with application binaries and their config files like `thrift, value, traefik`. For other images *Columbus* also discovered $tags$ for dependent software or third-party libraries. In some cases, we found that discovered $tags$ are prefixes of real software names, for example, tags `couch, mongo` correspond to software packages `couchdb, mongodb`, respectively. Although these $tags$ are representative enough, but in our future work we are trying to make discovered $tags$ match with real software name.

Software for which $tags$ were discovered successfully, we further processed them for software version identification. Only 16% of the software had their version number embedded in their namespace. For example,

mariadb:
/usr/share/doc/mariadbserver10.1/mysqld.sym.gz
/usr/share/doc/mariadbserver10.1/INFO_BIN



(a) FT_{exec} runtime



(b) FT_{name} runtime

Fig. 6. Case Study - Performance Evaluation

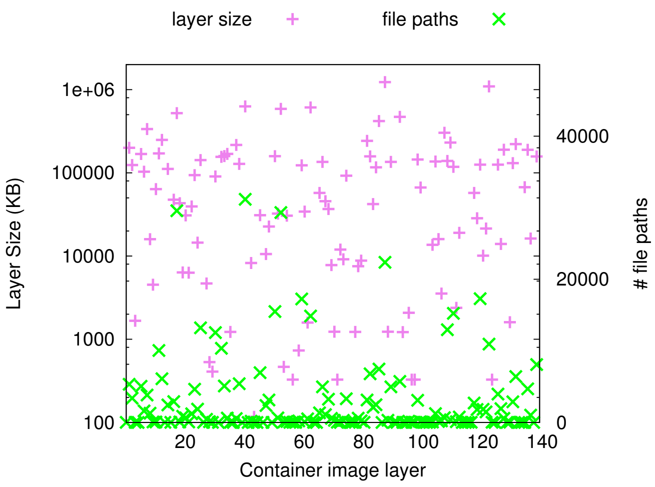


Fig. 7. *Case Study: Scale Evaluation*

python:

/usr/local/lib/python3.5/sitepackages/..

/usr/local/include/python3.5m/..

We identified other two common heuristics to discover software version namely $readme$ and $version$ files. Software more generally tends to encode their version information either in a file named "version" or "readme". So these file contents can be read and parsed to potentially extract the software version information. So for each software, we searched for these files in the namespace identified by their corresponding tag. Our analysis showed 22% software had $version$ file while 25% had $readme$. In cloud environment users may not be comfortable to let other tools read contents of their files. For this reason, accessing file content should be done with explicit user permission. In the current design, *Columbus* works only from the filesystem metadata and do not access any file contents. Thus, *Columbus* is not able to discover software version from these file contents. Although, we are exploring to enable this capability in *Columbus* along with a policy manager, wherein user can selectively enable file contents parsing to improve the software version discovery.

Figure 7 summarizes *Columbus* result across all layers of all images. This graph shows statistics for the image layers in which we successfully discovered $tags$. We observed that there were 54% layers with size $0B$. These empty layers represent no filesystem changes during image build process. These layers are ignored in the experiments. Here are important observations:

(a) $valid$ software $tags$ discovered 48% of layers. Thus there's an opportunity to optimize our filtering techniques even further.

(b) $tags$ were discovered from layers with size varied from less than $10KB$ to greater than $1.2GB$.

(c) $tags$ were discovered for filesystem impression size in number of files from as low as 4 to as high as $31K$.

(d) Figure 6 shows discovery time for *Columbus* across all images. Since there are limited number of executable files in an image across all sizes, they are processed quickly and FT_{exec} discovery time (Fig. 6 (a)) is typically less than $10seconds$. On the other hand for FT_{name}, since every file is processed discovery time is relatively higher (Fig. 6 (b)).

Thus, regardless of the size of filesystem impression or number of files accounted in impression, *Columbus* is effectively able to discover software $tags$. And attributed to our new frequency trie data structure, *Columbus* exhibits an optimal performance discovering these $tags$.

VI. Conclusion

In this work we presented design and implementation of *Columbus* as a general purpose software discovery technique that does not require any package manager query, knowledge database or fingerprint matching tools. Experimental evaluation show that it can discover software for around 92% of official Docker images. *Columbus* can even detect software suites like *wordpress, owncloud* and compositional software like ELK . In future work, we plan to extend *Columbus* as a general purpose cloud operational insights solution for fault-isolation, tag-correlation and system behavioral analysis.

References

[1] "Docker Hub," *https://hub.docker.com/*.

[2] "Amazon AWS Marketplace," *https://aws.amazon.com/marketplace*.

[3] "IBM Bluemix," *www.ibm.com/cloud-computing/bluemix/*.

[4] "Clair," *https://github.com/coreos/clair*.

[5] "Agentless system crawler," *https://github.com/cloudviz/agentless-system-crawler*.

[6] "Open source software discovery," *http://ossdiscovery.sourceforge.net*.

[7] "Endpoint manager relevance language guide," *https://github.com/bigfix*.

[8] "Rule Writing for OSS Discovery: Guide to identifying open source software," *http://ossdiscovery.sourceforge.net/WritingProjectRules.pdf*.

[9] "OpenIOC," *http://www.openioc.org/*.

[10] 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*, pp. 90–99, IEEE, 2014.

[11] 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.

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

[13] "Trie," *https://en.wikipedia.org/wiki/Trie*.