

ImageElves: Rapid and Reliable System Updates in the Cloud

Deepak Jeswani, Akshat Verma, Praveen Jayachandran, Kamal Bhattacharya
IBM Research - India.

Abstract—Virtualization has significantly reduced the cost of creating a new virtual machine and cheap storage allows VMs to be turned down when unused. This has led to a rapid proliferation of virtual machine images, both active and dormant, in the data center. System management technologies have not been able to keep pace with this growth and the management cost of keeping all virtual machines images, active as well as dormant, updated is significant. In this work, we present *ImageElves*, a system to rapidly, reliably and automatically propagate updates (e.g., patches, software installs, compliance checks) in a data center. *ImageElves* analyses all target images and creates reliable image patches using a very small number of online updates. Traditionally, updates are applied by taking the application offline, applying updates, and then restoring the application, a process that is unreliable and has an unpredictable downtime. With *ImageElves*, we propose a two phase process. In the first phase, images are analyzed to create an update signature and update manifest. In the second phase, downtime is taken and the manifest is applied *offline* on virtual images in a parallel, reliable and automated manner. This has two main advantages, (i) spontaneously apply updates to already dormant VMs, and (ii) all updates following this process are guaranteed to work reliably leading to reduced and predictable downtimes. *ImageElves* uses three key ideas: (i) a novel per-update profiling mechanism to divide VMs into equivalence classes, (ii) a background logging mechanism to convert updates on live instances into patches for dormant images, and (iii) a cross-difference mechanism to filter system-specific or random information (e.g., host name, IP address), while creating equivalence classes. We evaluated the ability of *ImageElves* to speed up mix of popular system management activities and observed upto 80% smaller update times for active instances and upto 90% reduction in update times for dormant instances.

I. INTRODUCTION

Virtualization has made it increasingly easy and less expensive to create new virtual machine instances. There is evidence that instead of simply consolidating existing workloads into fewer servers, virtualization has led to an increase in the number of systems, now running as VMs [10]. There is a proliferation of barely used VMs, as developers forget to return (or intentionally postpone, anticipating reuse) the VMs they do not use to the resource pool at the end of a project. Unfortunately, system management technologies have been unable to keep pace with this rapid proliferation, and the management cost of keeping all virtual machine images updated, both dormant and active, is significant. In a recent study, it was noted that more than 58% of virtual machine images in a university data center had not been used for over 1.5 months. Nearly a quarter of the images in the university's data center as well as in EC2 were not updated for 2-3 months [21]. This can pose a significant security threat to the data center, as well as increasing the time for updating

these dormant images and getting them 'ready' when they are needed again.

Due to resource sharing and I/O indirection, virtualization has also exacerbated the problem of system management, making it more expensive and time consuming to update systems reliably. Anti-virus updates and operating system patches can result in increases in CPU utilization of the order of 40% on a virtual machine – changes that would have a negligible impact on a physical system [10]. As any system update is deemed unreliable, in order to deal with unforeseen problems, traditionally, any changes or updates to a running system are applied by taking the application offline for a large enough change window, applying the updates while system administrators are on stand-by, and then restoring the application. Often, the actual change window is significantly larger than the time required for the update [16]. In this work, we propose *ImageElves* to rapidly, reliably, and automatically propagate system updates (e.g., patches, software installs, compliance checks) to dormant as well as live virtual machine images in a data center.

There are several tools available today that automate application of patches and compliance checks for online virtual machines [6], [15], [8], [7], [16]. While these tools address the scalability challenge, the patches themselves remain unreliable, and administrators need to fix any failed updates manually. Once a failed update is fixed manually, the administrator has to then ensure that the patch itself is fixed or create yet another patch that can then be fed to the tools for automatic application on the VMs. Hence, these tools do not help reduce the total application downtime by much, as a large chunk of the downtime is needed to manually fix any unforeseen problems during the update process. Furthermore, these tools do not patch dormant VM images. Recently, a novel tool called Nüwa was presented in [21] to patch dormant VMs. The tool automatically rewrites a patch that is intended to be applied on online VMs, to exclude or replace statements that require a running VM. This modified patch is then applied on the dormant VMs. Apart from being unable to handle online VMs and the additional cost of rewriting patches, this tool has additional limitations in that the rewritten patches are not guaranteed to succeed.

Contribution: In this paper, we design a system called *ImageElves* that can work for (i) both active and dormant instances, (ii) guarantees the reliability and time taken for an update, (iii) can handle all kinds of updates (without source code), and (iv) scales with new types of updates. *ImageElves* works by first identifying equivalent images and then applying offline update manifests on equivalent images in

a reliable fashion. *ImageElves* uses three key ideas: (i) a novel per-update profiling mechanism that partitions the VMs into equivalence classes, (ii) a background light-weight logging mechanism to convert updates on live instances into update manifests for dormant instances, and (iii) a cross-difference mechanism to identify random or system-specific information across VMs (such as hostname or IP address) while creating equivalence classes. Instances that belong to a common equivalence class are guaranteed to perform identically for the specified update. This reliability allows us to perform updates fully automatically and in parallel on all equivalent instances, leading to significant reduction in update time and labor cost. Further, it avoids expensive operating system and application testing after update application, leading to shorter change windows. We implemented *ImageElves* on a target system with 37 virtual machines and observed upto 80% reduction in update times for active instances and upto 90% reduction in update times for dormant instances.

II. BACKGROUND AND MOTIVATION

We first present a background on how servers are updated in production data centers.

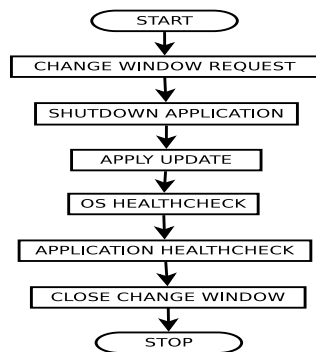


Fig. 1. Update Flow in Production Data Centers

A. Update Process on Production Servers

Any update in a production environment is a change with the potential to break the application in the environment. Hence, updates are dealt with very carefully, using a change management process. Huang *et al.* present the change management process for patching in [6]. Even though the exact change management process differs across data centers as well as with the nature of updates, the essential steps remain the same.

We abstract a generic change management process for various kinds of updates in Fig. 1. One may note the structural similarity of this process with various documented change management processes (e.g., [6]). The process starts with a request for a change window to perform the change. Once the change window is granted, the application is shutdown at the start of the window. The update is then applied on the target server. Once the update completes, testing is performed to ensure that the update did not break the application. Testing consists of operating system health checks followed by health checks for the application. If testing succeeds, the change window is closed. Otherwise, manual remediation is performed followed by re-testing.

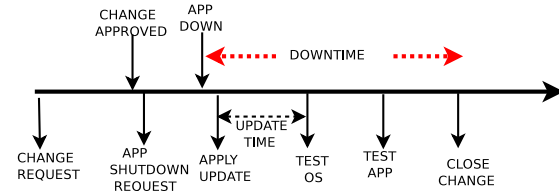


Fig. 2. Timeline of an update

We capture the steps on a timeline in Fig. 2. The most important thing to observe here is that, even if the actual time taken for the update to be applied is small, the overall time taken from the request for a change window to a change close is fairly large. Approval for a change window is manual and takes time. More importantly, the actual change window is often orders of magnitude larger than the update time [16]. This is because of the inherent unreliability of this change process requiring expensive operating system and application tests. Coupled with a buffer time kept for manual remediation, the actual planned downtime is typically a few hours, even if the actual update may take less than 10 mins. Further, every update has huge associated labour cost due to the testing performed after the update is completed.

The above process captures how an update is applied on one system. Frequently, updates need to be applied on almost all instances in a data center (e.g., an OS security patch) or on very large number of instances (e.g., database or web server updates). Due to inherent risk in the update process, administrators typically update only a few systems to start with. Once the first few updates succeed, they try to update more systems. Updating all relevant systems in the data center often happen months after the update was available [4].

In this work, we focus on system updates, which are common across a data center. Examples of such updates are security patches, compliance updates, installation and configuration of system management software, upgrades of operating systems and common middleware. The common denominator for all these updates are that they are applied on a large number of systems. The core idea behind our work is to profile these updates on very few instances and use that information to update majority of the instances in a reliable and low-cost manner. Application-specific updates like code upgrades, which touch few instances only, are outside the scope of this work and should be handled using their regular update process.

B. VM Sprawl

Server virtualization leads to a reduction in the hardware footprint of the data center by consolidating multiple workloads as virtual machines on a shared physical server. This leads to a reduction in data center infrastructure cost. However, system administration cost is proportional to the number of distinct managed servers (physical or virtual) in a data center. Every virtual server runs an operating system and an application stack, which need to be administered. In fact, it has been observed that virtualization leads to an increase in the number of running systems in a data center [10].

An increase in the number of managed VMs in a virtualized data center is fairly easy to explain. Virtualization allows virtual machines to be provisioned automatically in the order

of a few minutes. In comparison, provisioning a physical server was a time-consuming activity, requiring multiple levels of approval in a traditional data center. Further, since virtual machines can be created with fairly small sizes, users request virtual machines indiscriminately. Finally, one can shutdown virtual machines and free up any CPU or memory resources used by the virtual machine. Since storage is fairly inexpensive, this has led to users creating virtual machines indiscriminately and shutting them down, when not required. In a recent survey, it was observed that more than 58% of the VMs in a cloud were not used for the last 1.5 months [21].

The explosion of virtual machines (both active and dormant) in a virtualized data center has led to the problem of *VM Sprawl*. VM sprawl poses significant challenge for system administrators in a data center. Firstly, it increases the labour cost of managing and updating the software components due to an increase in the number of managed instances. Secondly, since a large number of instances are dormant at a given point in time, scheduled updates miss the dormant instances. Since the resources consumed by dormant instances are released to other active instances, it is not even possible to bring up all dormant instances and update them periodically. This combined with the inherent unreliability of system updates exacerbates the situation for system administrators.

In this work, we pursue the idea that VM sprawl can also be leveraged to increase the inherent reliability of system updates. The ease of cloning instances and the use of golden masters to provision instances in a virtualized data center often leads to a large number of virtual machines with the same system and middleware footprint (with possibly different applications and data). Given an update, our key idea is to automatically profile the update and identify all instances that are semantically equivalent for the update. Equivalent instances are guaranteed to respond identically to an update, allowing updates to be applied reliably, automatically, and rapidly.

III. *ImageElves* DESIGN

We first present the goals of *ImageElves* and then describe the techniques that help us achieve these goals.

A. Design Goals

- **Reliable Update Process:** One of the primary issues in the current system update process is the lack of reliability. Since it is not clear if an update will succeed, many system management activities are still performed under manual supervision. In case an update fails, the system administrator decides to roll back the update cleanly or perform remediation actions and retry the update. A reliable update process is a pre-requisite for automation.
- **Deterministic Update Duration:** The lack of reliability of the system update process also reflects in long change windows. Even if an update completes in 5 mins in most cases, the change window requested by a system administrator is proportional to the worst case update duration. Change windows in production systems often involve application downtime or reduced application availability (e.g., reduced cluster size). Hence, an important goal

for *ImageElves* is to ensure deterministic update times, leading to short change windows.

- **Reduced Update Duration:** *ImageElves* also attempts to reduce the update duration, leading to even shorter change windows.
- **Update both Dormant and Active Instances:** Virtualized data centers have a large number of dormant VMs along with the active instances. *ImageElves* attempts to ensure that both active and dormant instances are updated.
- **Reduced Labour Cost:** One of the most important business goals of *ImageElves* is to reduce the overall labour cost for managing systems in large data centers.
- **Content-Oblivious Updates:** Automation of the update process should be oblivious to the specific commands executed by the updates so as to be generic and applicable to all updates.

B. Design Overview

ImageElves introduces the following novel ideas to ensure a reliable, automated and low cost system update process in data centers.

- **Per-Update Equivalence Class Identification:** *ImageElves* creates a signature for each update and partitions all relevant instances (online and offline) in a data center into equivalence classes based on the signature. Signature consists of all files which may have a dependency on the update. A successful update on one member of an equivalence class guarantees successful update on other members, ensuring that the update process is reliable.
- **Cross-Difference Signature Filter:** Signature files may typically contain environmental parameters (e.g., IP addresses, timestamps) leading to semantically identical images being classified as not equivalent. We employ a cross-difference signature filter to eliminate environmental or random noise from signature files allowing for creation of larger equivalence classes.
- **Offline Manifest Creation:** We log all file system changes during an update to create an offline manifest that can be applied on other equivalent images with deterministic update durations. Further, the offline manifest is created in a content-oblivious manner, i.e., oblivious to the actual actions performed by the update or any human remediation actions performed for the update to work. Offline manifest also allows us to meet the design goal of handling dormant instances.
- **Parallel Automated Offline Updates:** Once the offline manifests are created, the reliability and content-oblivious nature of manifests allow us to automatically update multiple systems in parallel, leading to significant reduction in update time.

C. Identifying Equivalent Images

One of the central ideas behind *ImageElves* is to identify a set of VM images, which are equivalent. A pair of images can be considered as equivalent if the result of any update applied on the two images would be identical. It is obvious that there would be very few images, which are universally equivalent (i.e., they would behave identically for any set of updates).

Clearly, only clones are universally equivalent. We conjecture that for a particular update, there may be many images which are equivalent to each other with respect to that update. We first define a few notations that help us formalize the notion of equivalence.

Definition 1: Signature: A signature $S_{i,j}$ for an update U_i and an image I_j is defined as the set of all image files that impact the application of U_i on image I_j . It includes the list of all files which are read by the update U_i , along with their attributes and hash value of the content of files after cross difference filter.

Definition 2: Manifest: A manifest $M_{i,j}$ for an Update U_i and an instance I_j is defined as all files and properties that are modified or created by the update U_i on image I_j . A manifest captures all filesystem changes and applying the manifest $M_{i,j}$ offline is equivalent to applying the update U_i on instance I_j .

Definition 3: Equivalent Images: Two instances I_j and I_k are said to be equivalent for a given update U_i , i.e., $I_j \equiv_{U_i} I_k$, iff their manifests are identical $M_{i,j} = M_{i,k}$.

It is easy to understand that if an update is applied successfully on an instance I_j , the update will also be successful on any instances that are equivalent to I_j for the given update. Consider also the case where an update U_i was not automatically successful on an instance. However, a system administrator performed additional operations on the instance to make the update successful. Consider a new update U'_i , which includes the update U_i as well as the additional manual remediation actions performed to make the update successful. Any instances that are equivalent to I_j for this modified update U'_i can also be successfully updated by applying the update U_i along with the manual actions performed in U'_i .

The notion of per-update equivalence is very powerful and we leverage it to ensure reliability. Given a set S of instances on which an update U_i needs to be applied, we only need to identify the distinct equivalence classes for U_i in S . Then, if the update is successful for any one image in an equivalence class, it will necessarily be successful on all other images in that class. Further, if an update failed on one image in an equivalence class, we can reliably identify that the update will fail on all other images in that class, without having to apply the update on any of them. In order to identify equivalence classes for an update U_i , we use the files that constitute the signature of the update. Given a successful update of U_i on an instance I_j , we identify the files in the update's signature $S_{i,j}$. All images that have an identical content in the signature files are marked as equivalent. We use SHA1 technique to test whether the content is identical. The details of this test is discussed in subsequent section.

It is important to note that images with identical content of the files in signature set are guaranteed to be equivalent. However, images with different signatures can also have the same manifest and may be equivalent. Hence, our approximation may create more equivalent classes than may actually exist in a set of instances. The key property of this approximation is that it is safe, i.e., *if we identify an update as reliable, it is indeed reliable.*

D. Cross-Difference Mechanism

We investigated files that constitute the signature for many updates. We observed that often files in the signature are configuration files for operating system, OS components, or application packages. Updates often look for the presence or configuration of various components or applications they depend on and access the configuration files associated with these applications. Configuration files often contain environmental parameters or *Points of Variability (PoV)*, e.g., IP addresses, hostnames. These PoVs differ across instances and as a result, even if a configuration file in two different instances are semantically equivalent, a simple *diff* between the two files will show differences. Hence, *ImageElves* would not be able to classify two such images as equivalent (even if they indeed are equivalent for an update).

We also observed another common issue with such files. Many applications often use XML files for configuration and tag XML nodes with randomly generated numbers. These random numbers show up as differences between two files, which are identical in all other respects. Since *ImageElves* is designed to be content-oblivious (i.e., does not use expert knowledge about the actual actions performed by updates), discarding the PoVs or random attributes while classifying files is challenging. *ImageElves* instead uses a slightly different approach to identify noise or environmental parameters.

ImageElves uses clones of one instance to create the signature and manifest. Since the two instances are clones, their signature and manifest should be identical modulo any noise or environment parameters. Any difference in content between the signature of the two instances is annotated as noise. This cross-difference information is passed to a file difference method, which ignores any differences at locations annotated as noise between two files it compares. We observed that this cross-difference mechanism was able to dramatically increase the number of images in an equivalent class and thereby decrease the number of equivalence classes identified.

E. Offline Update Manifest Creation

ImageElves allows updates to be applied offline on a dormant VM instead of a running instance. This allows us to update dormant VMs without any need to bring them online. Dormant VMs often do not have compute or memory resources assigned and bringing them online for updation is not feasible. Further, dormant images often lie in backup application instances or in libraries, which are brought up only when the primary instance has failed. In such cases, updating the dormant instances may delay recovery, leading to extensive downtimes. Offline updates can also lead to faster updation in many cases, as we will show experimentally. Further, a large number of images can be updated in parallel in an offline fashion.

An offline update to an image is nothing else but applying the update manifest on equivalent images. It is important to note that a manifest created using one instance can not be applied on another instance, unless the two instances are guaranteed to be equivalent. The manifest is created on one online instance for each equivalence class in a straightforward manner. Before the update is applied, a snapshot is taken of the

instance. Once the update completes successfully, an image *diff* is performed between the snapshot and the updated instance. For every file that is different, a patch is created to transform the file in the snapshot to the file in the updated instance. The collection of all these patches constitute the manifest for the update, which can subsequently be used to update the remaining instances in the equivalence class offline.

F. ImageElves Automated System Update Process

We now describe the overall update process enabled by *ImageElves*. Given a new update U_i that needs to be applied on a target set of instances S , *ImageElves* follows the two-phase iterative process described in Fig. 3.

```

Input: Set  $S$ , Update  $U_i$ 
While  $S$  is not empty
  Phase 1: Select a target instance  $I_j$  at random
  Apply the update  $U_i$  on  $I_j$ 
  Create the Signature  $S_{i,j}$  and manifest  $M_{i,j}$ 
  Update  $S_{i,j}$  and  $M_{i,j}$  using cross-difference
  Phase 2 (Parallel): Compute  $S_{i,k}$  for all images  $I_k$ 
  and create equivalent classes
  For all  $I_k \equiv_{U_i} I_j$ , (Parallel Step)
    If  $U_i$  was successful on  $I_j$ 
      Apply  $M_{i,j}$  or  $U_i$  on  $I_k$ 
    Else
      Mark  $I_k$  as failed
    End-If
  Remove  $I_k$  from  $S$ 
End-For
End-While

```

Fig. 3. *ImageElves* Update Flow

The update process takes an update along with possible target instances S as input. It first finds a leader in the set S . On the leader, a regular update process is applied, which involves taking a conservative downtime. The leader can be elected randomly or based on some criterion; possibly an instance, which is not running any critical application. The update process may succeed quickly, may succeed with human intervention or fail. In any case, a signature of the final update is created, which is then used to identify equivalence classes. All instances that are equivalent to the leader are either marked as failed (in case the original update had failed) or updated reliably (the update may either be applied offline using the manifest or online using the original update).

The actual update can be applied either online or offline. If the update is applied using the offline manifest, all relevant images are updated in parallel. We recommend the following rules of thumb for selecting between online and offline mechanism for applying an update to a equivalence class. For dormant instances, the offline manifest is likely to be the preferred mechanism. If the update required human intervention on the leader, the offline manifest is again likely the preferred mechanism, since the offline manifest captures both automated and manual actions and can be replicated on the remaining VMs in the equivalence class without any human intervention. For error-free updates on active instances,

the choice between offline and online may depend on the duration of the online update and the number of equivalent images in the class. If the online update took a long time or required reboots, the offline manifest may speed up the process deterministically. If the online update was short and applicable to only a few images, it may make sense to apply the update online. The entire process can be optimized further by pipe lining manifest creation and update application across multiple equivalent classes. Our current implementation of *ImageElves* is captured in Fig. 3 and optimizations to further speed up the process are left as future work.

IV. IMPLEMENTATION DETAILS

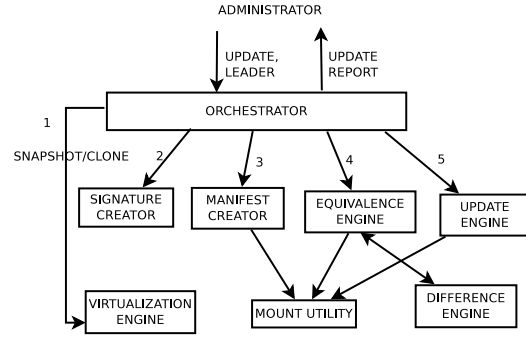


Fig. 4. *ImageElves* Component Diagram

We have implemented *ImageElves* in python to update virtual machine images. Our current implementation works only for Linux and Unix systems. It does not handle changes to the Windows registry, but we are planning to extend our implementation for Windows systems as well. Figure 4 captures the key modules implemented in *ImageElves*.

An *Orchestrator* component drives the overall update process. It takes from the administrator an *Update* and a *Leader* VM image to apply the update. It invokes a *Virtualization Engine* to take a snapshot and then asks the administrator to perform the update using the *Signature Creator* module. Once the update is completed, the *Signature Creator* module returns the signature $S_{i,j}$ associated with the update U_i on instance I_j . The *Orchestrator* then invokes the *Manifest Creator*, which leverages the snapshot to create a *Manifest* for the update. The *Orchestrator* passes the signature of the update to the *Equivalence Engine*. The *Equivalence Engine* leverages a *File Difference Engine*, which implements the cross-difference mechanism, and returns the equivalence classes formed. Both the *Equivalence Engine* and the *Manifest Creator* use a *Mount Utility* to mount images during equivalence class and manifest creation. We next describe details of each individual module.

A. Mount Utility

The proposed system currently uses VMware VDDK to mount images. Since multiple images can have the same volume attributes, the Logical Volume ID, Physical Volume ID and Volume Group ID of the images are changed temporarily. The renaming is necessary to avoid namespace conflicts on the system. The IDs are restored to their original values when images are unmounted.

B. Signature Creator

As mentioned earlier, the current implementation of *ImageElves* works for UNIX based operating systems. To create the signature for an update, we make use of UNIX's strace tool [11]. Strace is a diagnostic, instructional, and debugging tool, which intercepts and records all system calls invoked by a process. The administrator applies the update on the leader within the operational context of strace. Thus, when the update runs, all files touched by it or its child threads are captured. A conservative signature is created by *ImageElves* that includes all files read by the update on the leader. We filter out few files containing low-level system specific information from this conservative list. The files which are filtered out in our current implementation are */etc/mtab*, */etc/ftab* (both contain text information of all mounted filesystems), */etc/resolv.conf* (contains DNS info) and */etc/passwd*.

The *Signature Creator* module keeps a record of all file names and the operations performed on them as part of the signature of the update. The operations that are recorded by strace consist of open (read/write/modify), create, delete (rename, unlink) and permission modification operation (chmod, lstat, statfs). Once this signature is ready, we leverage the *Difference Engine* to apply the cross-difference operation between the leader and its clone to prune out random and environmental parameters from these files. The environmental parameters can be identified using a Points-of-Variability (PoV) mining system (e.g., [3], [9]). The PoVs and random values identified are removed from the signature files and a SHA1 digest of each file is created. This digest is used as the file signature to create equivalence classes.

C. Manifest Creator

The *Manifest Creator* module mounts the leader containing the update and its snapshot of state before the update to find out the changes created in the file system by the update. It looks for changes only within files captured as part of the signature as identified by the *Signature Creator* module. The manifest for the update contains all files that were modified or created by the update, their meta information (such as path, permissions, and ownership), as well as instructions regarding the nature of operations performed on each file. This information describing the contents of the manifest is stored in a meta manifest file. The instructions are different for different types of operations performed by the update. For a file create operation, the *Manifest Creator* makes a copy of the file (data) after removing PoVs. It adds the location of the copied file in the meta manifest file, along with instructions to copy and replace PoV values when applying the manifest on another image. For files which are modified, the *Manifest Creator* module uses the Linux diff utility to create the patch. It adds the location of the patch along with instructions to apply the patch in the meta manifest file. Similarly, for files for which permissions are modified, the manifest contains the new permission as data, and instructions to change permission of the file to the new value. Files that were only read are excluded from the manifest, and for files that are marked for deletion, a delete instruction is added in the manifest.

D. Equivalence Engine

Given an update signature for a leader, the *Equivalence Engine* identifies other images in the data center, which are equivalent to the leader. The process to create signatures for other images is the same as the signature creation process in the *Signature Creator* module. The files corresponding to the signature are fetched from the instances and the random values and PoVs are filtered out. A SHA1 hash is computed for the each file and is used for equivalence check. For each target image, the hash of files in the signature are compared against the corresponding hash for the leader. If the hash matches for all files, the target image is tagged as equivalent to the leader and can be updated automatically. The above process is completely parallelizable and is performed for all images in parallel. It is also possible to use the above computation to create equivalent classes for images that are not equivalent to the leader. Those classes can be reused during equivalence class creation for the next leader. However, we have not implemented this optimization currently.

E. Limitations of ImageElves

We now discuss certain known limitations of our system. First and foremost, as with any patching or software upgrade process, it is possible that upgrading or patching certain components of a system, may cause higher level applications to fail. While *ImageElves* guarantees that the patch or upgrade applied is successful, it provides no guarantees on the correct functioning of higher level applications. The implications of applying a system upgrade has to be considered before using *ImageElves* to apply the upgrade. Second, the cross-difference mechanism used by *ImageElves* does not work for binary files, and any PoVs hidden within binary files will be missed. Nevertheless, using *ImageElves* to apply upgrades that have binary files in their signatures is still reliable, except that we may create more equivalence classes than actually necessary. Finally, the current implementation of *ImageElves* does not have any failure recovery capabilities in the event of a failure of the system running *ImageElves*. But, traditional recovery techniques such as using snapshots can be used to restore VMs that were being upgraded back to known stable states.

V. EXPERIMENTAL EVALUATION

In this section, we present results from various experiments conducted to study the effectiveness of *ImageElves*. We compared *ImageElves* with a baseline update process, where a system administrator performs the updates online using patch executables, installers, and customized scripts invoked using ssh. Our experiments were designed to investigate the following for the *ImageElves* update process.

- Ability to update dormant images
- Increased reliability of the update process
- Reduction in downtime
- Overhead of the process

A. Experimental Setup

The evaluation of the proposed approach was performed on 37 virtual machines. In future we plan to experiment on large

number of VMs in a data centre. Few VMs under evaluation were running enterprise applications like Hadoop and Trade6 accessing clusters of WAS machines and DB2 machines. Others were used by researchers for their personal work. The VMs were KVM images running on RHEL 5.6 operating system, with an average size of about 12 GB. The images were stored in a 6 TB NFS volume. The *ImageElves* operations were performed from a blade server having 10 GB memory and two quad-core 3.0 Ghz Intel Xeon processors which were able to serve 8 workers to perform heavy parallelizable tasks. *ImageElves* was implemented in Python v2.7. We chose 10 updates for the purpose of evaluation, which can be broadly classified into the following three categories:

- **Compliance:** These are updates applied on a VM to check and enforce compliance with standardized specifications for the data center. The updates considered were to set audit flag, set audit flag for all sysadmin actions, to check and set password length, and to check and set password age.
- **Software install and configuration:** The updates in this category were installation of TEM client (Tivoli Endpoint Manager) [6], TAD4D, Epel repository, and configuration of yum with a local IBM repository.
- **Patches:** These are the most frequent kind of updates which are used in data centers and clouds to keep systems updated and secure. Logwatch security and DHCP security patches were evaluated with *ImageElves*.

For experiments on dormant VMs, all VMs were dormant and powered off to begin with. For experiments on active VMs, all VMs were powered on and running to begin with. In reality, a data center may have dormant as well as running VMs, and *ImageElves* is capable of leveraging this state information in deciding which VMs should be picked as leaders for update profiling, logging, and manifest creation. In our experiments, the leader VM for each equivalence class was picked randomly.

B. Update Process

The process to apply each update on dormant VMs using *ImageElves* was as follows. We first booted up one VM, applied the update online and created the signature and manifest. We then parallelly mounted all other VM images in the pool, estimated if they differed in the signature files for the update, and created equivalence classes. If an equivalence class had fewer than 4 VMs, we applied the patch online on each of them sequentially one after another. If an equivalence class had 4 or more VMs, we picked a leader VM from amongst them randomly, powered it on, applied the patch online, and created the manifest for other VMs in the equivalence class. Once the manifests for every equivalence class was created, the manifests were applied parallelly on all the remaining dormant VMs offline. Parallelizing application of the manifest on offline VMs is feasible as the process is guaranteed to succeed. We noticed that increasing the parallelism by 16 VMs caused the running time for this step to increase by only 10%, similar to the numbers reported in [19] for offline updates. In addition to the above steps, the process to apply the updates on online VMs using *ImageElves* involved shutting down the

Adjustment	Direct manifest	Diff class same manifest	Number of classes
audit (comp)	17	16	3
PA (comp)	15	10	6
pwLen (comp)	17	17	4
pwAge (comp)	16	18	4
TEM (s/w)	28	5	4
TAD4D (s/w)	19	13	5
epelRepo (s/w)	14	13	9
yumRepo (s/w)	36	0	1
logwatch security (patch)	15	14	8
DHCP security (patch)	13	13	8

TABLE I
NUMBER OF IMAGES IN EACH EQUIVALENCE CLASS CREATED BY *ImageElves* FOR EACH UPDATE

VMs to apply the manifest offline and booting up the VMs once the process was completed.

The baseline process we compared against involved applying the patches on running VMs. This process applied on dormant VMs involved booting up the VM, copying necessary files over, running the update installer or script over an ssh connection, and then shutting down VM, similar to the Microsoft Virtual Machine Servicing Tool [14]. These steps are applied sequentially on the VMs one after another (unless otherwise mentioned). The process applied on active VMs involved shutting down the application, copying files over, executing the update and then restoring the application.

C. Results

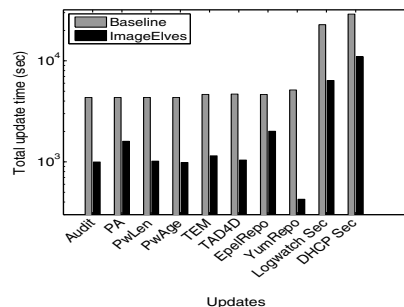


Fig. 5. Total time taken for applying updates on 37 dormant VMs

We first present the total time taken for various updates to be applied on a pool of 37 dormant VMs using *ImageElves* as well as the baseline process in Figure 5. Table I provides details of the number of equivalence classes and the number of VMs in the two most dominant classes created by *ImageElves* for each update. It can be observed that only a few equivalence classes are created for each update, and hence most VMs are updated offline leading to 75% or higher reduction in the total time needed for 7 out of 10 updates. For the yum update, we see a 90% reduction in update time establishing the effectiveness of *ImageElves* to reduce update time.

The total time taken for various updates on 37 active VMs using *ImageElves* and the baseline process is presented in Figure 6. For active VMs, *ImageElves* has the additional overhead of shutting down VMs and turning them on, and the speed up over the baseline process varies between 30% and 80%.

We next investigate the reliability of the *ImageElves* process. We define *precision* of *ImageElves* as the fraction of VMs on which the update was successfully applied. We define *recall* of *ImageElves* as the ratio of the number of VMs on which

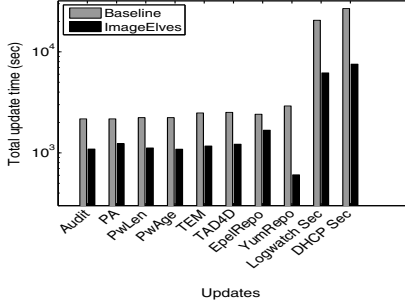


Fig. 6. Total time taken for applying updates on 37 active VMs

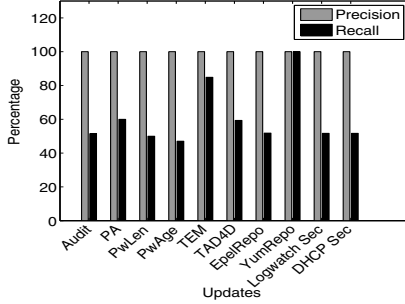


Fig. 7. Reliability and accuracy of the updates using *ImageElves*

ImageElves was able to identify that the VM belongs to the same equivalence class as the leader VM to the number of VMs that actually had the same manifest as the leader VM, expressed as a percentage. The numerator is the number of VMs for which the signature files matched that of the leader VM and *ImageElves* was able to guarantee that they belong to the same equivalence class. This number is shown as column ‘Direct manifest’ in Table I. The column ‘Diff class same manifest’ in Table I shows the number of VMs missed by *ImageElves* due to minor inconsequential differences in the signature files in these VMs. This column together with the column ‘Direct manifest’ is the denominator for estimating recall, which gives the number of VMs for which the manifest was observed to be the same as the leader VM. The precision and recall for *ImageElves* across the 37 VM images is plotted in Figure 7. *ImageElves* has a 100% precision and a recall of over 50% for all the updates. While there is potential for improving the method for equivalence class creation, these numbers nevertheless demonstrate the reliability of the process.

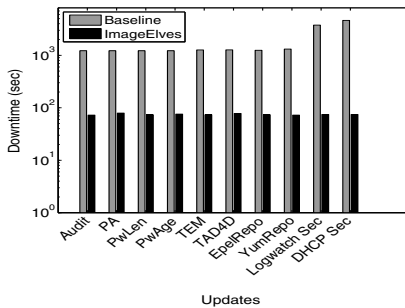


Fig. 8. Total downtime needed for updates using *ImageElves* and the baseline process

Figure 8 presents estimates on the total downtime needed for *ImageElves* as well as the baseline process. For the baseline process, we assume that the downtime needed is 5 times the total update application time, together with 5 minutes each for OS and application health checks. For *ImageElves*, such a large downtime is needed only for one leader VM in each equivalence class. Once a manifest has been created for each equivalence class, for the remainder of the VMs the update is guaranteed to succeed. So, the downtime can be much shorter, which we fix as 1.1 times the update application time using *ImageElves*. These numbers are shown in Figure 8 (for *ImageElves* we do not show the downtime needed for the leader VMs, but only for the VMs on which the updates are applied offline). *ImageElves* can reduce the downtimes for updates on VMs by an order of magnitude (about 1/15 in our results).

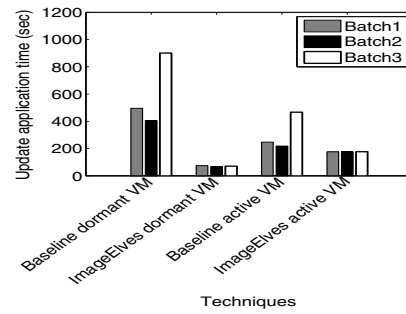


Fig. 9. Update times for batched updates on a dormant and active VM

Typically, administrators batch updates together to be applied during scheduled downtimes. We recreate this scenario by grouping updates together into three batches. The first batch consists of the compliance updates audit, PA, pwLen, and pwAge. The second batch consists of the installation and configuration of software packages TEM, yum, and epel. The third batch applies all the updates together. Figure 9 depicts the update application times for these three batches of updates on a single dormant or active VM (assuming that the manifest has already been created for the VM). Even for a single VM, when updates are batched together *ImageElves* is able to reduce the update application time compared to applying the batch of updates online, for both dormant as well as active VMs.

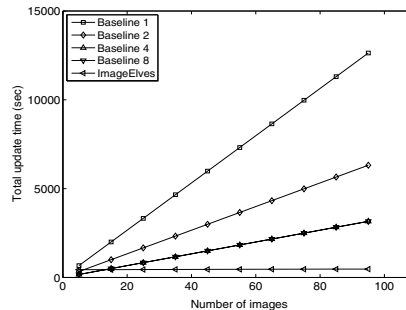


Fig. 10. Total update times for different number of dormant VM images

We next measure the total time needed for updates (for the end-to-end process) for increasing number of VMs for both *ImageElves* and the baseline process. This is reported

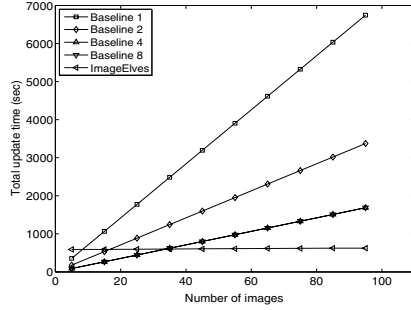


Fig. 11. Total update times for different number of active VM images

in Figure 10 for dormant VMs and in Figure 11 for active VMs. The number of images is increased from 5 to 95 in steps of 10. For the baseline process, we consider different cases where the system administrators are assisted by a tool such as [19], that allows them to parallelize the application of updates on multiple VMs. We consider 4 cases where the administrator can apply updates on 1, 2, 4, or 8 VMs parallelly, depicted as ‘baseline dormant/active 1/2/4/8’. As the updates are not guaranteed to succeed, it would be impossible for an administrator to track the progress of more than 8 VMs at a time. We note that *ImageElves* outperforms even a parallelization of 8 VMs updated online at a time, when the number of images is more than 15 for the dormant case, and when the number of images is more than 35 for the active case.

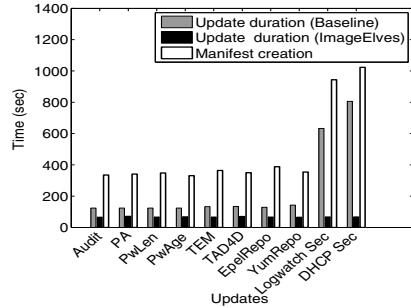


Fig. 12. Overhead of manifest creation and application for a dormant VM using *ImageElves* compared to the time taken for applying the update online

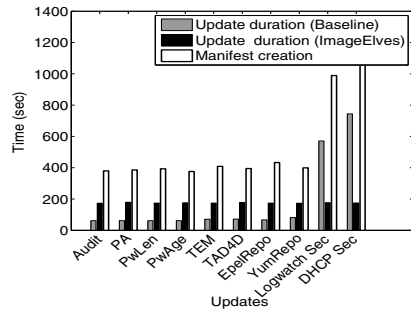


Fig. 13. Overhead of manifest creation and application for an active VM using *ImageElves* compared to the time taken for applying the update online

Finally, we quantify the overhead of update manifest creation and update application for *ImageElves* with reference to the time taken to apply the update using the baseline process. This is shown in Figures 12 and 13 for dormant and active VMs, respectively. Note that the overhead of update creation

is required only for one VM in each equivalence class and the overhead of update application applies to all VMs on which the update is applied offline. The update application times are significantly lower than the time taken to apply the update using the baseline process for the dormant VMs case, but is more for the active VMs case. This is because a majority of the time is consumed in shutting down and booting up the VMs for *ImageElves*. However, for larger updates such as the logwatch and DHCP security updates, the update application times for *ImageElves* is significantly lower than the time taken for applying the update online using the baseline process for both dormant as well as active VMs. Thus, if small updates are grouped into sufficiently large batches, *ImageElves* can significantly reduce the update application times for both dormant as well as active VMs.

VI. RELATED WORK

The related work to *ImageElves* can broadly be classified into semantic mining of virtual servers, online system management tools, and offline updation tools.

Semantic Mining of Virtual Servers: In this work, we annotate virtual machines into equivalence classes based on the content and configuration in the virtual machines. Classical discovery of configuration information in images was performed using live profiling of the applications. The Galapagos system used network profiling to capture dependencies between systems [13]. In [18], the authors present a system to automatically generate predicates for configuration troubleshooting. The advent of virtualization has led to image analysis techniques, which can analyze images much more quickly. The Mirage system [1] can index virtual images and support operations to identify difference between images. The Morpheus system can configure applications using example annotated configuration files [9].

Online System Management Tools: There are several tools available today that aid in automating patching and compliance tasks for online virtual machines. The IBM Tivoli Endpoint Manager [5], [6], provides a host of endpoint management services including patch management, security and policy compliance. Fixlets or messages that have in-built relevance clauses and action buttons are used to deliver one or more patches or fixes to specified endpoints. The tool is generic for physical as well as virtual machines, but does not cater to offline VM images. Windows Server Update Services [15] enables the distribution of patches released through Microsoft Update to computers within a network. The VMware vSphere Update Manager [8] allows administrators to propagate patches and the VMware vCenter Configuration Manager [7] aids security and policy compliance for VMs managed using VMware vSphere. The Oracle Enterprise Manager Ops Center [16] is an end-to-end systems management solution that includes patch management and compliance. All the above tools require some amount of scripting and human intervention to schedule patches. Both Rackspace and Microsoft Azure provide patching as a service in their respective public cloud offerings, but the technology used is not publicly available. These services are rather expensive - for instance, Rackspace charges \$100 per VM per month to ensure that the

VMs are at the latest patch level. At the OS level, there have been attempts at virtualizing and migrating the applications so that the underlying OS can be patched without disturbing the running application [12], [17]. Researchers have also looked at generic IT Service automation frameworks [2], which combine manual and automated steps to reduce the service delivery cost in data centers.

Offline Updation Tools: While we are unaware of any tools that help ensure compliance for dormant VM images, there are a few tools that cater to the problem of patching dormant VM images. The Microsoft Virtual Machine Servicing Tool [14] patches dormant VMs by first creating a VM from the image, powering on the VM, patching it online and then powering off and returning the image to the library. This can be extremely costly and time consuming when there are hundreds of dormant images. Further, powering on VMs that haven't been used for several months could expose the datacenter to security threats. The VMware vCenter Protect tool (formerly Shavlik NetChk Protect) [20] applies patches to images in a lazy fashion. The patch installer and data are injected into the image so that the patch will be applied the next time the image is booted up. This allows the downtime needed for patching to be reduced, but there could be a significant delay during image startup. This could be of the order of tens of minutes even for images that have been dormant for one month, which can be prohibitive.

A novel tool called Nüwa to patch dormant VM images is presented in [21]. In order to apply a patch that is designed to be applied online, on to an offline image, they automatically rewrite the patch script to safely exclude or replace statements that require a running system. In addition to the cost of rewriting patches, this technique has additional limitations in that not all patches are guaranteed to succeed using this process and the tool is not capable of handling online VMs. Their current implementation supports Debian and RPM package managers for Ubuntu, and they showed that 99% of patches could be successfully applied with a speed-up of about four times.

In contrast to the above methods, *ImageElves* can rapidly and automatically propagate and apply any patch, system update, or compliance check to online VMs as well as offline images. This is done by first profiling each update and dividing VMs into equivalence classes. The update is then applied online on one member of each equivalence class, which is then converted into a patch for offline images using a background logging mechanism. Any system-specific or random information such as hostname or IP address is identified and replaced using a cross-difference mechanism. We guarantee that all patches applied offline will work reliably leading to reduced and predictable downtimes.

VII. CONCLUSION

In this paper, we presented *ImageElves*, a tool to rapidly, reliably and automatically propagate system updates such as patches, software installs, or compliance checks to dormant as well as active virtual machines. By identifying classes of equivalent images for each update, *ImageElves* applies offline update manifests on equivalent images in a reliable fashion. Such updates applied offline are guaranteed to succeed, and

ImageElves is able to reduce the update application time by up to 80% for dormant as well as active VMs, and reduce the total downtime needed for updates by over an order of magnitude. As future work, we plan to experiment and evaluate proposed approach on large scale data centre. Also, there is ample scope for improving the construction of equivalence classes and extending this technique on kernel update patching. This improvement can be achieved by exploring a wider range of system updates and can lead to a further reduction in the update application time of *ImageElves*.

REFERENCES

- [1] Virtual machine images as structured data: the Mirage image library. In *Proc. HotCloud*, 2011.
- [2] Naga Ayachitula, Melissa J. Buco, Yixin, Maheswaran Surendra, Raju Pavuluri, Larisa Shwartz, and Christopher Ward. It service management automation - a hybrid methodology to integrate and orchestrate collaborative human centric and automation centric workflows. In *IEEE SCC*, 2007.
- [3] Rahul Balani, Deepak Jeswani, Dipyaman Banerjee, and Akshat Verma. PoVMine: Highly Accurate Configuration Mining for Error-free Re-configuration. Technical Report R113001, IBM Research, 2013.
- [4] Olivier Crameri, Nikola Knezevic nd Dejan Kostic, Ricardo Bianchini, and Willy Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. In *SOSP*, 2007.
- [5] Hai Huang, Salman Baset, Chunqiang Tang, Ashu Gupta, K. N. Madhu Sudhan, Fazal Feroze, Rajesh Garg, and Sumithra Ravichandran. Patch management automation for enterprise cloud. In *Proc. of NOMS*, pages 691–705, 2012.
- [6] IBM. Tivoli Endpoint Manager. Online, <http://www-01.ibm.com/software/tivoli/solutions/endpoint/>.
- [7] VMware Inc. vCenter Configuration Manager. Online, <http://www.vmware.com/products/configuration-manager/overview.html>.
- [8] VMware Inc. vSphere Update Manager. Online, <http://www.vmware.com/products/datacenter-virtualization/vsphere/update-manager.html>.
- [9] D. Jeswani, R. Balani, A. Verma, and K. Bhattacharya. Morpheus: Learning configurations by example. In *Proc. of IEEE IM*, 2013.
- [10] Evangelos Kotsovinos. Virtualization: blessing or curse? In *Communications of the ACM*, 2011.
- [11] Paul Kranenburg. debugging utility for linux. <http://linux.die.net/man/1/strace>.
- [12] D.E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. of ASPLOS*, pages 211–223, 2004.
- [13] Kostas Magoutis, Murthy Devarakonda, and Kiran Muniswamy-Reddy. Galapagos: Automatically discovering application-data relationships in networked systems. In *Proc. of IEEE IM*, 2007.
- [14] Microsoft Corporation. Virtual Machine Servicing Tool 3.0. Online, <http://technet.microsoft.com/en-us/library/cc501231.aspx>.
- [15] Microsoft Corporation. Windows Server Update Services. Online, <http://technet.microsoft.com/en-us/windowsserver/bb332157.aspx>.
- [16] Oracle Corporation. Realizing the Economic Benefits of Oracle Enterprise Manager Ops Center (White Paper). Online, <http://www.oracle.com/us/products/enterprise-manager/economic-benefits-ops-center-185035.pdf>, April 2012.
- [17] S. Potter and J. Nieh. Reducing downtime due to system maintenance and upgrades. In *Proc. of Large Installation System Administration Conference*, 2005.
- [18] Ya-Yunn Su and Jason Flinn. Automatically Generating Predicates and Solutions for Conguration Troubleshooting. In *Proc. of Usenix Technical Conference*, 2009.
- [19] Balaji Viswanathan, Akshat Verma, Bharat Krishnamurthy, Praveen Jayachandran, Kamal Bhattacharya, and Rema Ananthanarayanan. Rapid adjustment and adoption to miasa clouds. In *Proc. of Middleware, Industry track (to appear)*, December 2012.
- [20] VMware Inc. (formerly Shavlik). VMware vCenter Protect. Online, <http://www.vmware.com/products/datacenter-virtualization/vcenter-protect/overview.html>.
- [21] Wu Zhou, Peng Ning, Xiaolan Zhang, Glenn Ammons, Ruowen Wang, and Vasanth Bala. Always up-to-date - scalable offline patching of vm images in a compute cloud. In *ACSAC*, 2010.