# Shadow Patching: Minimizing Maintenance Windows in a Virtualized Enterprise Environment

Duy Le*, Jidong Xiao*, Hai Huang†, Haining Wang*

*The College of William and Mary, Williamsburg, Virginia, USA
†IBM T. J. Watson Research Center, Hawthorne, New York, USA

***Abstract*—Software is growing bigger and more complex, which results in bugs and defects being no longer dealt as exceptions, but rather as normal artifacts in a software's lifecycle. In fact, many patches are released by vendors on a preset schedule. This implies that managing patches in a correct and timely manner has become an important factor in smoothly running an IT environment. However, when a patch is applied, the affected software is often required to stop temporarily, which can cause a disruption of service. The down time is commonly called a maintenance window. Although sophisticated live patching techniques have been previously proposed, their applicability in practice is very limited. In this paper, we propose a novel patch management technique based on commonly available virtualization capabilities. It allows system administrators to perform a majority of the patch work outside of the maintenance window, such as downloading patches, installing them, and performing post-installation testing and fixes. By capturing the disk activities and replaying them during the actual maintenance window, we can transform a complex software patching operation to a series of more deterministic file I/O operations, and thus, reducing maintenance window from hours to minutes.**

## I. Introduction

In data centers, there is a strict requirement on applying patches before their deadlines to satisfy Service Level Agreements (SLAs). Patch scheduling has become a complex task, similar to a multi-dimensional optimization problem, e.g., simultaneously trying to minimize service downtime, avoid change freeze due to business requirements, and line up all the relevant support teams to be available during the maintenance window. As SLAs are getting adopted by more Cloud service providers, patch management will be offered more as a standard managed service to their customers. Any benefit result from reducing downtime can grow significantly when multiplied by the number of virtual machines managed in a cloud.

In a managed environment, when a patch is released, it is always a challenge to strike the right balance between applying the patch to the affected systems as soon as possible and the unfortunate fact that patching the affected systems will almost always result in service downtime. During this time, the affected software is first shut down or put into a quiescent state before patches are applied. Afterwards, various tests are performed to verify if the system is still in a working state, in terms of functionality, performance, and scalability, which all are time consuming tasks. If a patch or one of the post-installation tests has failed, then it could take many hours for system administrators to find a fix and make the necessary adjustments. Furthermore, if the fix cannot be found within the originally scheduled time frame (e.g., 6-8 hours), SLAs could be violated.

Due to such potential complications (to both service providers and customers), some may choose to run outdated software [1] to avoid or minimize service disruptions, and resort to other means as a temporary solution. However, not fixing a known problem at its root, i.e., in the affected software, could lead to serious consequences, especially when the problem exposes security vulnerabilities.

In this paper, we present a patch management technique called Shadow Patching (SP). It uses commonly available virtualization capabilities to transform a set of interactive steps to patch a system into a series of deterministic file operations that can quickly be replayed in a batch mode. In the framework of SP, we first instantiate an exact replica of the virtual machine (VM) that is to be patched. Second, the replica VM will be patched in the conventional way, i.e., download the patch, install the patch, test the patch, and make any necessary changes. Finally, during the maintenance window when the original VM is to be patched, we will need to merge (1) the data that has been changed by the running application in the original VM, and (2) the data that has been changed by installing the patch in the replica VM.

The remainder of this paper is organized as follows. Section II surveys related work. Section III describes the architectural design and implementation details of Shadow Patching. Section IV presents the experimental results of using SP to apply patches to different types of software. Finally, Section V concludes the paper.

## II. Related Work

SP framework is based on various online and offline patching techniques. Online patching techniques attempt to eliminate downtime all together by performing various in-memory modifications. However, by nature, these techniques target very specific software as they need to understand the intricacies of a particular software. It is challenging to build a generic online patching framework, and it would require significant modifications to system infrastructure in order to employ such techniques [2]. Another approach to online patching is to make use of live migration capabilities that is commonly available in virtualized environments [3], [4], [5]. Before a patch is applied, one could migrate applications and services to another environment, and migrate them back when the change is done. Depending on what needs to be patched, different migration capabilities are needed. For example, when the hypervisor needs to be patched, VM migration can be used

so VMs are not impacted during the change. When a VM needs to be patched, its processes will need to be migrated to avoid being impacted. Unfortunately, in the current state-of-art, not everything can be migrated cleanly, let alone in a live manner. Moreover, if a patch changes the underlying system data structure and/or interfaces related to migration, this will prevent whatever was migrated from being migrated back.

There are also many offline patching techniques. VMWare's vSphere Update Manager inserts patches into the software update process [6]. Microsoft's VMST wakes a dormant VM up, and then applies software patches on this system [7]. Nuwa requires rewriting installation scripts to apply software patches directly on a VM image [8]. However, if a software is patched offline, the patched system needs to be restarted [6] or stayed at dormant states [8]. In this work, we are primarily concerned with patching of running systems and how to reduce their downtime. Although offline patching techniques greatly facilitate software maintenance, they do not reduce system downtime.

## III. Shadow Patching Framework

### A. Patching scenario

An important design goal of Shadow Patching is to be practical and generally applicable. It should be usable by anyone who possesses the basic system administration skills. A high level step-by-step process of using SP is described as follows:

- **VM snapshot:** VM snapshot allows one to capture all states of a VM at a particular point in time. The snapshot can be deleted, reverted, and used to created additional snapshots and clones of this VM. As shown in Figure 1, a VM can be running on top of multiple layers of snapshots. For example, the base image was snapshotted when the OS was first installed, and Delta-0 was snapshotted when various software were installed and configured. We take a snapshot of the virtual machine before patching, and from which we will create a clone of it.
- **Patching and restarting:** Assume VM1 is the original VM that needs to be patched, we first clone it to create VM2. Two delta files — Delta-1 and Delta-2 — are created at this time to track changes in the two VMs. This operation should have virtually no impact to VM1, and VM2 can be instantiated as an exact replica of VM1. Patch downloading, installation, configuration, testing, and any necessary adjustments will be done in VM2, and all persistent data changes due to these interactive operations will be captured in Delta-2. As VM2 is configured exactly the same as VM1, any operations (e.g., patching, testing) done in VM2 should yield the same result as if the operations were performed in VM1.
- **Merging deltas.** After a patch is applied and tested in VM2, the VM can be shut down to minimize its resource usage. During maintenance window, VM1 is first shutdown so Delta-1 can be kept at a consistent state, and then the modified data captured in the two delta files are analyzed and merged. As Delta-1 has captured data changes due to the running workload in VM1 and Delta-2 has captured data changes due to software patching in VM2, merging the delta files are usually a straight-forward operation. However, sometimes conflicts do occur, where a file is modified by both VMs. We propose various conflict resolution methods in Section III-B2.
- **Activating the original system**. After the delta files are merged, Delta-2 can also be discarded. The time it takes to analyze and merge the delta files is the downtime that is now required, and these steps are all done in a batch mode, thus, resulting in much smaller downtime than the traditional in-place patching. VM1 can be started after the merge operation has completed and continue its normal operations.

It is worth to note that we do not do less work to achieve smaller downtime. Instead, the work is simply shifted to a cloned VM while allowing the primary VM to continue to run undisrupted. The reduced downtime is a result of compacting and recording interactive work a sysadmin has done to patch the cloned system and replaying it on the primary in a batch mode.

### B. Managing delta files

A key component in SP is how it manages delta files. Specifically, it needs to monitor for data changes so it can merge them. And in case of conflicts, it is also responsible for resolving them to minimize human involvement.

*1) Monitoring deltas:* Keeping track of data changes can be done at either the data block level or the file system level. At the data block level, one can directly compare Delta-1 and Delta-2, block by block. There are several drawbacks to this approach. Most notable ones are: i) it is specific to the image format of the delta files, i.e., we would need to implement one method to compare 2 QCOW delta files and another for 2 VMDK delta files, and ii) we would need to perform a reverse lookup to find which file a modified data block belongs to, and this operation is also different for different guest file systems. On the other hand, at the file system level, modified files can be found simply by scanning and comparing the entire file systems of VM1 and VM2. However, as a file system can easily contain millions of files, we could spent many hours just scanning through the file systems, which defeats the purpose of reducing maintenance window. SP monitors for data changes at the file system level by using *inotify* [9]. We use what inotify captures as our delta files.

We have configured inotify to only monitor for data changes. To track changes of all files and directories in the file system, inotify recursively monitors I/O events from the system's root directory. Four types of events are monitored, including IN_MODIFY, IN_MOVE_FROM/TO, IN_DELETE, and IN_CREATE. Once an event is triggered, inotify captures this timestamped event. This information is stored in a log file. Files and directories are frequently modified due to background system processes and other running applications. To minimize monitoring overheads, SP configures inotify to maintain a list files and directories to be excluded, such as device files, pipes, sockets, temporary directories, log directories, etc.

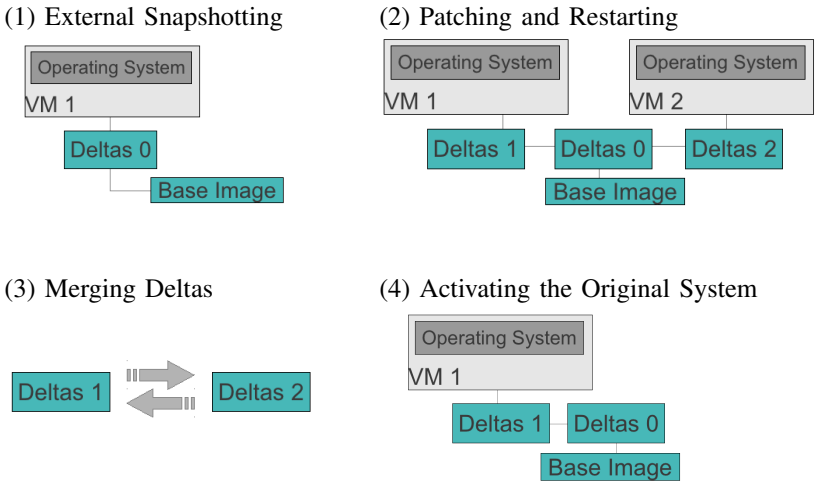


Fig. 1: Scenario of SP session

TABLE I: Time base comparison between files/directories. $SS$: Snapshot time, $T_1$: Last modification time of file on Deltas-1, $T_2$: Last modification time of file on Deltas-2

| Rules | File on Deltas-1 | File on Deltas-2 | Time conditions | Decisions |
|---|---|---|---|---|
| 1 | Yes | No | $T_1 \leq SS$ | Delete on Deltas-1 |
| 2 | No | Yes | $SS \leq T_2$ | Copy from Deltas-2 to Deltas-1 |
| 3 | Yes | Yes | $T_1 \leq SS$ and $SS \leq T_2$ | Copy from Deltas-2 to Deltas-1 |
| 4 | Yes | Yes | $SS \leq T_1$ and $SS \leq T_2$ | *Hybrid* copy from Deltas-2 to Deltas-1 |

*2) Merging deltas:* The underlying technique of merging deltas between Deltas-1 and Deltas-2 is the proper management of files and directories from the file systems of VM1 and VM2. Based on the delta files and exclude list provided by inotify, SP can determine the modified files and directories. In particular, the focus of this merging is to decide whether or not the files or directories in Deltas-1 should be kept, deleted, or replaced. The rule is based on the modification time of the file or directory and the snapshot time of VM1. Here, the modification time of a file or directory can be retrieved from its inode. Based on these estimated times, Table I lists the rules made on those files or directories. The details of these rules are described as follows:

- **Rule 1:** A file on VM1 has not been written or modified since the snapshot time. If the cloned version of this file on VM2 is deleted by a software update, the file should be deleted.
- **Rule 2:** A file exists on VM2 but does not on VM1. This file is most likely created due to patching. It should be copied back to VM1.
- **Rule 3:** A file on VM1 has not been accessed or modified since the snapshot time. However, the cloned version of this file on VM2 is modified because of a software update. Thus, the file must be replaced by the newer version, which is copied back from VM2.
- **Rule 4:** A file on VM1 and its cloned version on VM2 are modified after the snapshot time. These two files must be kept on VM1 after the merging by performing a *hybrid* copy. Basically, a hybrid copy consists of three steps: (1) renaming those two files based on their inode information, so that their names are different; (2) copying a newly renamed file from VM2 to VM1; and (3) creating a symbolic link on VM1 based on the original name of the file. To guarantee that the freshly copied file will be used once VM1 starts, the symbolic link is linked to the newly copied file rather than its original version.

### C. Prototype of Shadow Patching

A working prototype of SP is built on a Linux system. SP requires disk images to be in one of the copy-on-write image formats, e.g., QCOW, QCOW2, VMDK, etc. Copy-on-write image formats allows disk images to be externally snapshotted without having an impact on the running VM. This cannot be achieved on a raw disk.

To merge deltas, QCoW disk images are exposed as mount points at the file system level of the host machine. Different storage utilities can be used to leverage this mounting feature, such as kvm/qemu-nbd [10] for QCoW/QCoW2, Vmount [11] for vmdk, and losetup [12] for raw formats. SP benefits from kvm/qemu-nbd that includes two components: client and server. As a kernel module, the client handles requests passed through the device node. These requests are forwarded to the server that stays at the user level. Then, the server processes the requests in order to access the data residing on the QCoW disks.

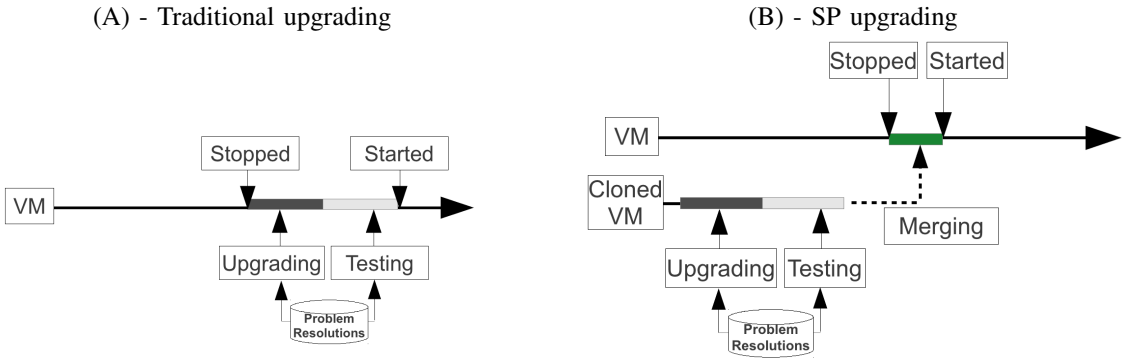


Fig. 2: Upgrading process of application service

TABLE III: Upgraded application services, inotify overhead, and utilized benchmarks.

| Services | Old Packages | | | New Packages | | | Overhead (%) | Benchmarks |
|---|---|---|---|---|---|---|---|---|
| | Versions | Sizes (KB) | # Files | Versions | Sizes (KB) | # Files | | |
| Bind9 | 9.4.2 | 744 | 36 | 9.7.0 | 1.024 | 51 | 1.23 | DnsPerf |
| SubVersion | 1.4.6 | 3.400 | 28 | 1.6.6 | 4.204 | 35 | 1.30 | Collabnet |
| NFS | 1.1.2 | 504 | 35 | 1.2.2 | 640 | 43 | 1.01 | IOzone |
| OpenVPN | 2.1 | 1.060 | 86 | 2.1.3 | 1.208 | 93 | 1.34 | NetPerf |
| PostgreSQL | 8.3.16 | 13.884 | 95 | 8.4.9 | 14.804 | 92 | 1.40 | PGbench |
| Samba | 3.0.28 | 9.216 | 43 | 3.4.7 | 16.676 | 55 | 1.67 | Dbench/Netbench |
| Squid | 1.9 | 1.584 | 33 | 2.7 | 1.892 | 36 | 0.95 | Web Polygraph |
| Apache2 | 2.2.8 | 4.356 | 492 | 2.2.14 | 8.864 | 564 | 1.86 | Apache Bench |
| VsFTPd | 2.0.6 | 396 | 41 | 2.2.3 | 460 | 44 | 1.23 | Dkftpbench |

TABLE II: Testbed setup

| | Hardware | Software |
|---|---|---|
| **Host** | Pentium D 3.4GHz 1TB SATA, 2GB RAM | Ubuntu 11.10, kernel 3.0.0-12, libvirt 0.9.8 |
| **Guest** | Qemu 0.14.1 1 GB RAM | Ubuntu (10.04, 10.10, 11.04) |

## IV. Experimentation

### A. Experimental setup

To evaluate the effectiveness of SP, we use two metrics: correctness and the size of the maintenance window required. To verify correctness, we run application-specific benchmarks after patch deployment to check if the patched software has the right version and achieves the expected functions and performance. Additionally, as patching in SP is transformed to file compare, replace, merge, and delete operations, we scan file systems to verify all files and directories associated with a patch are correctly placed. The maintenance window is another key metric. We compare SP with the traditional patch management method for both success and failure scenarios. The software and hardware configurations of our test machine are listed in Table II.

### B. When Patch Succeeds

In traditional software patching practice, a maintenance window is scheduled for making changes to running systems. The action of applying a patch (usually would succeed) takes only a few minutes. However, running a regression test and resolving any unexpected problems would take much longer time. Thus, maintenance windows are usually scheduled to range from hours to days depending on the complexity of the patch. To allow sufficient amount of time to perform problem diagnosis and resolution, service providers are usually conservative in scheduling the maintenance windows. However, even if the entire window is not used, it would be difficult for users to salvage any of the remaining time to reduce service downtime because the patch completion time within the window is non-deterministic. This traditional process is illustrated in Figure 2(a), and SP's is shown in Figure 2(b).

In our experiments, we use Ubuntu's dpkg to perform upgrades or patching. Nine applications, listed in Table III, are selected to be patched. As mentioned before, one method to ensure correctness post-patch is by running application-specific regression tests, for which, we ran those commonly used benchmarks that are also shown in the table. Besides a detailed specification of each application software, the column *Overhead* denotes the results of overhead induced by inotify. Since inotify works as a part of the Linux's virtual file system, it only induces 1-2% overhead on system I/Os, which is fairly negligible.

We compare user perceived service downtime, which is shown in Figure 3. In the traditional approach, the time it takes to apply the patch and perform a regression test will all be visible to users. However, in the case of SP, patching and testing occur in a separate cloned VM. This is completely hidden from users and can be done before the maintenance window even starts. The downtime is only visible when we compare and merge disk deltas of the two VMs. In Figure 3, for each application, the left column shows the user perceived downtime when the traditional approach is used, and the right column shows the downtime when SP is used.

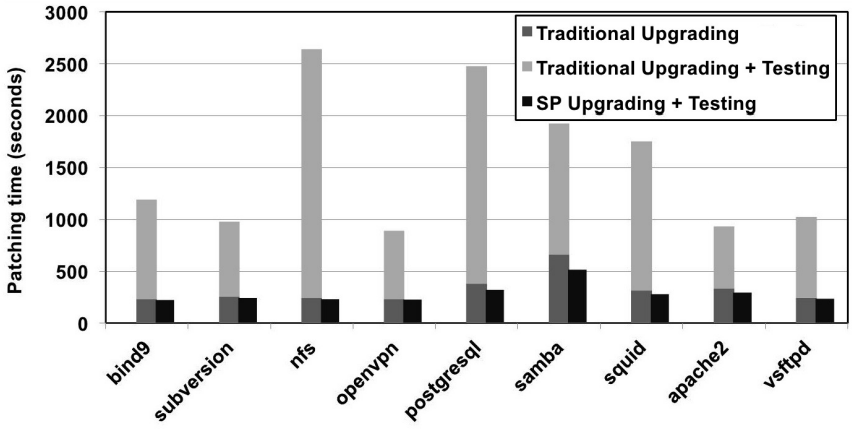


Fig. 3: Individual upgrading application services (lower is better)

We quantify the I/O activities caused by merging deltas and show the results in Table IV. The columns in the table are grouped based on the type of activities. Since rules 2 and 3 consist of regular copies, the results of these activities are combined into one column. Based on these results, we make the following observations:

- **SP significantly shortens service downtime**. Because the tests of upgraded services are conducted on the cloned VM, the functional testing time, or *SP testing*, does not impact on the patching time. As an example, a thorough test of a patched NFS server using IOzone can take up to 40 minutes. Running this test on the cloned VM would take the same time and provide the same results but without being visible to users.
- **SP lowers overhead incurred by software component replacement**. Comparing two versions of an application, if changes are minor, most files and directory structures will be similar, if not almost identical. If changes are more extensive, the similarities become less significant. Traditionally, patching an application involves three steps: (1) removing current application's files and directories, (2) extracting the new version of the application into a temporary location, and (3) copying the extracted files and directories into the right place. For a package whose changes are minor, this technique causes unnecessary I/Os on files and directories, which are identical between two versions. However, SP avoids this redundancy by comparing inode information of files and directories between two versions before each delta merge, and thus, resulting in fewer I/O operations. As an example, our results show that SP helps Postgresql and Samba minimize their upgrading time.
- **SP does not impact on the number of merging activities**. Application services include sets of files, which can be unchanged or significantly modified from their previous versions. SP utilizes this observation to minimize the number of copies. In addition, although merging activities include deletes, copies, and hybrid copies, we can see that the majority of the activities are regular copies. The number of copies occurred on each package depends on the difference between software versions, rather than its size or the number of files.

TABLE IV: Rule-based activities in merging deltas of application services: (1) deletes, (2 & 3) copies, and (4) *hybrid* copies.

| Services | Rule 1 | Rule 2 & 3 | Rule 4 |
|---|---|---|---|
| Bind9 | 2 | 48 | 0 |
| SubVersion | 4 | 29 | 2 |
| NFS | 2 | 35 | 3 |
| OpenVPN | 2 | 89 | 0 |
| PostgreSQL | 5 | 82 | 4 |
| Samba | 2 | 40 | 8 |
| Squid | 3 | 34 | 0 |
| Apache2 | 5 | 552 | 6 |
| VsFTPd | 5 | 41 | 0 |

### C. When Patch Fails

We further compare SP with the traditional approach when one or more patches fail. A service pack is a bundle of many patches to upgrade the current system version to the next stable version. Patches are applied in a certain order to satisfy software dependencies, and if any one fails, it is simply skipped (as well as any dependent ones). However, the failed patches will eventually need to be resolved within the maintenance window.

A patch can fail for many reasons, such as insufficient hardware, driver problems, incompatible setup process, inconsistent system configuration, wrong architecture edition, data loss, permission/access problems, or software bugs. To resolve a failed upgrade, the following steps are usually taken: (1) reporting a problem, (2) looking for solutions from different databases, while waiting for the problem being solved, and (3) applying solutions to fix the failed upgrade. If a failure is caused by software bugs, bug-fixing is a non-trivial task. Generally, the time to fix a bug can be up to 200 days, although this number depends on the nature of the bug [13]. Recent studies of system configurations to upgrade software indicate that on average the time to fix one particular issue is no more than 5 hours [14]. This average time is also known

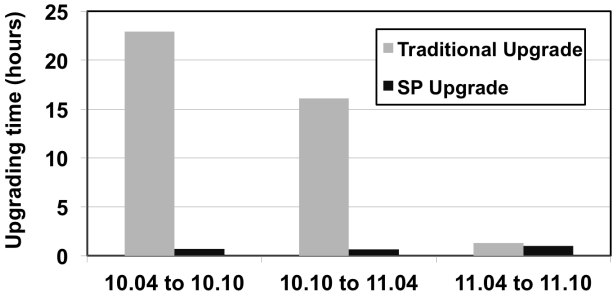


Fig. 4: Linux Ubuntu server distribution upgrade across different versions (lower is better)

as a *fixing effort* to denote an effort in person-hours to resolve an issue. In general, fixing issues existed in separate software packages can be accomplished in parallel, but requires more labor. Otherwise, the issues must be fixed sequentially.

Conducting a thorough test on an upgraded system is a complex and time consuming task. This is because upgrading a Linux system requires multiple replacements of software components, including executable binaries, shared libraries, configuration settings, databases, etc. To verify the accuracy and stability of upgraded software, various regression tests should be conducted. However, it is non-trivial to fully understand and prepare thorough tests for all upgraded software, and it is also out of scope of this work. Due to this complexity, we focus on upgrading time, rather than testing time. Basically, the upgrading time includes the time to replace software components and the time to reboot the system. For SP, the upgrading time consists of both the time to reboot the system and the time to merge deltas. Based on the comparison of results between the traditional and SP upgrades, which are shown in Figure 4, we make following observations:

- **SP helps a system administrator avoid failed upgrades**. Since many reasons can cause failed upgrades, if the maintenance window is short (e.g., a few hours) such failed upgrades may not be resolved. Thus, the system will not be fully upgraded. Increasing the maintenance window gives more time to resolve the problem, however, it also greatly increases the service downtime. SP is able to address this problem, since resolving failed upgrades is performed in the cloned system.

- **SP shows a low variation in upgrading time between different versions of a Linux system**. Traditional software upgrades do not guarantee that a Linux system can be successfully upgraded after a specific amount of time. This is because the actual time spent for an upgrade can vary from one to several hours. More specifically, the upgrading time induced by SP depends on not only system's configuration, but also the number and type of software packages. Our experimentation to upgrade different versions of a Linux system on the same test-bed shows a low variation of upgrading time. This is due to the similarity between different versions of the service pack.

## V. Conclusion

In this paper, we have presented the Shadow Patching (SP) framework to reduce the maintenance window associated with deploying software patches. Software patching, testing, and troubleshooting are all done in a cloned VM so that these tasks will have no impact on the original VM. File system changes in the cloned VM are recorded and are subsequently merged with the original VM. The only down time perceived by the original VM is when it is taken offline to perform this merge operation, which is much faster and reliable than what is done in the traditional method. By hiding post-patch regression test and troubleshooting steps, the maintenance window can be significantly shortened.

Through extensive experiments, we have demonstrated that SP is able to not only avoid failed upgrades, but also significantly minimize the maintenance windows. We believe that our framework will help system administrators in enterprise environments to optimize the software maintenance process. We also expect that our work will motivate software developers and system administrators to carefully monitor deltas at different levels, such as file systems and disk blocks, to shorten the software upgrading time.

## References

[1] E. Rescorla, "Security holes... who cares?" in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. USENIX Association, pp. 6–6, 2003.

[2] C. Giuffrida and A. S. Tanenbaum, "A taxonomy of live updates," in *Proceedings of the 16th Annual Conference of theAdvanced School for Computing and Imaging (ASCI'10), 2010*.

[3] "Windows 2000 clustering: Performing a rolling upgrade," http://technet.microsoft.com/en-us/library/bb742504.aspx.

[4] D. E. Lowell, Y. Saito, and E. J. Samberg, "Devirtualizable virtual machines enabling general, single-node, online maintenance," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS-XI)*. ACM, pp. 211–223, 2004.

[5] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about it? toward dependable, online upgrades in enterprise system," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware'09), 2009*.

[6] "Vmware vsphere update manager," http://www.vmware.com/support/pubs/vum_pubs.html [Accessed: January 2012].

[7] "Microsoft Virtual Machine Servicing Tool 3.0," http://www.microsoft.com/download/en/details.aspx?displaylang=en\&id=23300.

[8] W. Zhou, P. Ning, X. Zhang, G. Ammons, R. Wang, and V. Bala, "Always up-to-date: scalable offline patching of vm images in a compute cloud," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*. ACM, pp. 377–386, 2010.

[9] "inotify monitoring file system events," http://www.kernel.org/doc/man-pages/online/pages/man7/inotify.7.html [Accessed: June 2012].

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, pp. 225–230, 2007.

[11] "Vmware DiskMount Utility," www.vmware.com/pdf/VMwareDiskMount.pdf [Accessed: June 2012].

[12] "Set up and control loop devices," http://linux.die.net/man/8/losetup.

[13] S. Kim and E. J. Whitehead, Jr., "How long did it take to fix bugs?" in *Proceedings of the 2006 international workshop on Mining software repositories (MSR'06)*. ACM, pp. 173–174, 2006.

[14] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07)*. IEEE Computer Society, pp. 1–1, 2007.