RC24956 (W1003-034) March 11, 2010
Computer Science

IBM Research Report

Always Up-to-Date - Scalable Offline Patching of VM Images in a Compute Cloud

Wu Zhou[1], Peng Ning[1], Xiaolan Zhang[2], Glenn Ammons[2], Ruowen Wang[1], Vasanth Bala[2]

[1]North Carolina State University

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Always Up-to-date – Scalable Offline Patching of VM Images in a Compute Cloud

Wu Zhou Peng Ning
North Carolina State University
{wzhou2, pning}@ncsu.edu

Xiaolan Zhang Glenn Ammons
IBM T.J. Watson Research Center
{cxzhang, ammons}@us.ibm.com

Ruowen Wang
North Carolina State University
rwang9@ncsu.edu

Vasanth Bala
IBM T.J. Watson Research Center
vbala@us.ibm.com

Abstract

Patching is a critical security service to keep computer systems up to date and to defend against security threats. Existing patching systems all require running systems. With the increasing adoption of virtualization, there is a growing number of dormant virtual machine (VM) images. Such VM images cannot benefit from existing patching systems, and thus are often left vulnerable to emerging security threats. It is possible to bring VM images online, apply patches, and capture the VMs back to dormant images. However, such approaches suffer from performance challenges and high operation costs, particularly in large-scale compute clouds where there could be thousands of dormant VM images.

This paper presents a novel tool named *Nüwa* that enables efficient and scalable offline patching of dormant VM images. Nüwa analyzes patches and, when possible, converts them into patches that can be applied offline by removing operations that require a runnning system. Nüwa also leverages the VM image manipulation technologies offered by the Mirage image library to provide an efficient and scalable way to patch VM images in batch. Nüwa has been evaluated with real-world patches and on VM images configured with popular packages according to the Ubuntu popularity contest. Our implementation of Nüwa is based on the Debian package manager and our evaluation applies 406 patches to a fresh installation of Ubuntu-8.04. Nüwa successfully applies 402 out of the 406 patches, and speeds up the patching process by more than 4 times compared to the online approach. This can be further sped up by another 2–10 times when the tool is integrated with Mirage, making Nüwa an order of magnitude more efficient than the online approach.

1 Introduction

Patching is a basic and effective mechanism for computer systems to defend against most, although not all, security threats, such as viruses, rootkits, and worms [14, 21, 22]. Failing to promptly patch physical machines can subject the systems to huge risks, such as loss of confidential data, compromise of system integrity, and failure to provide regular system services. Unfortunately, applying security patches is a notoriously tedious task, due to the sheer large number of patches and the high rate at which they are released – It is estimated that, in the average week, vendors and security organizations release about one hundred and fifty vulnerabilities and associated patching information [16]. As a result, most software runs with outdated patches [12, 13].

The problem is exacerbated by the IT industry's recent shift to virtualization. Virtualization allows a complete system state to be conveniently encapsulated in a virtual machine (VM) image, which can be run on any compatible hypervisor. Because VM images are normal files, they can be easily copied to create new VM images. This has led to a new "VM image sprawl" problem. A direct result of the VM image sprawl problem is the significantly increased management cost to maintain these VM images. Because each VM image is a logical computer, it needs to be maintained like its physical counterpart, including regularly applying security patches. This is true even for dormant VM images that are not being actively used – they resemble physical machines that are powered down.

We argue that existing patching utilities, originally designed for running systems, are a poor fit for VM images. Because they require the VM images to be online before the patch can be applied, they do not scale to a large number of images – it takes on the order of minutes just to power up and shut down a VM image.

Current approaches to VM image patching are based on the traditional model where a VM image must be running in order for the patch to be applied. Thus they share the same scalability limitations as the traditional patching approach. One example is the Microsoft Offline Virtual Machine Servicing Tool [11], which adopts a method to bring the VM image online, apply the patches, and capture the VM back to a dormant image.

Realizing that not all VM images are needed immediately by their users, some solutions [25] use a clever optimization where patches are applied lazily. The patch installer and patch data are injected into the VM image in such a way that the patch process is triggered at the next booting time. This optimization can yield significant savings in the total time spent in patching in the case where only a small percentage of dormant images will ever be used. However, the tradeoff is that users will now see a significant delay in their image startup time, especially for images that have accumulated a long list of yet-to-be-applied patches.

Another weakness shared by all online patching approaches is that it assumes that the vulnerable versions of software on the target VM image will not be attacked during the time period between when the image boots up and when the patch finishes. That is not the case for the Windows Blast worm [4], where a machine could get infected immediately after it boots up and before it could get a chance to apply the patch. It is also possible for an already infected machine to constantly reboot itself such that the patching process never has a chance to complete.

We propose an approach that is fundamentally different from the traditional online model. We argue that the only way to make the patching process scalable in a cloud environment where the number of images can potentially reach millions [1] is to do it offline. A closer look into the patching process reveals that it can be decomposed into a sequence of actions, not all of which require a running system. In fact, most of the patching actions only depend on and have impact on file system objects, which are already encapsulated in the VM image itself. Among the actions that do depend on or have impacts on a running system, we find that many are unnecessary to execute when patching offline, and some can be safely replaced by other actions that do not need the running system. Based on these findings, we design and implement Nüwa [2], a scalable offline patching tool for VM images. By performing the patching offline, Nüwa avoids the expensive VM start and stop time, and ensures that for the majority cases, when a VM image is ready to be started, it always has the latest patches installed.

Because Nüwa is an offline patching tool, it can leverage novel VM image manipulation technologies to further improve scalability. In particular, Nüwa is integrated with the Mirage image library [23] which provides a rich set of tools and APIs for efficient offline image manipulation.

Our implementation of Nüwa is based on the Debian package manager [7]. We evaluated Nüwa with 406 patches for a freshly installed Ubuntu-8.04. Our evaluation results show that Nüwa can successfully apply 402 out of the 406 patches, and speeds up the patching process by more than 4 times compared to the online approach. This can be further improved by another 2–10 times when the tool is integrated with Mirage, making Nüwa an order of magnitude more efficient than the online approach.

[1] Amazon EC2 already contains over 6,000 public VM images. This number does not include private images that users choose not to share with others [20].

[2] Named after the Chinese Goddess who patches the sky.

We summarize our contributions below:

1. We designed a scalable offline patching tool for VM images that is backward compatible with an existing patch format. To the best of our knowledge, this is the first tool of its kind in the academic literature.

2. We implemented and evaluated the tool based on `dpkg`, a popular package installation system for Debian-based Linux distributions, and demonstrated its superior scalability – it sped up patching time by 8–40 times compared to the traditional online based approach.

This paper is organized as follows. Section 2 gives some background information on patching and describes our design choices and technical challenges. Section 3 presents an overview of our approach. Section 4 describes the mechanisms we use to convert an online patch into one that can be safely applied offline. Section 5 describes how we leverage efficient image manipulation mechanisms to further improve scalability. Section 6 presents our experimental evaluation results. Section 7 discusses related work. Section 8 concludes this paper with an outlook to the future.

2 Problem Statement

2.1 Background

Software patches, or simply patches, are often distributed in the form of software update packages (e.g., *.deb* or *.rpm* files), which are installed using a package installer, such as `rpm` and `dpkg`. In this section, we give background information on the format of software packages and the package installation process. We use the Debian package management tool `dpkg` as an example. Most software package management tools follow the same general style with only slight differences (e.g., a different shell to interpret the installation scripts).

Packages are distribution units of specific software. A package usually includes files for different purposes and associated metadata, such as the name, version, dependence, description and concrete instructions on how to install and uninstall this specific software. Different platforms may use different package formats to distribute software to their users. But the contents included inside are mostly the same. A Debian package, for example, is a standard Unix `ar` archive, composed of two compressed tar archives, one for the filesystem tree data and the other for associated metadata for controlling purposes. Inside the metadata, a Debian package includes a list of configuration files, md5 sums for each file in the first archive, name and version information, and shell scripts that the package installer runs at specific points in the package lifecycle.

The main action in patching is to replace the old buggy filesystem data with the updated counterparts. Beside this, the package installer also needs to perform some other operations to ensure the updated software will work well in the target environment. For example, a dependence or conflict must be resolved, a new user or group might have to be added, configuration modifications by the user should be kept, other software packages dependent on this one may need to be notified of this update, and running instances of this software may need to be stopped and restarted. Most of these actions are specified in the shell scripts provided by the package developers. Because these scripts are intended to be invoked at certain points during the installation process, they are called *hook scripts*. The hook scripts that are invoked before (or after) file replacement operations are called *pre-installation* (or *post-installation*) *scripts*. There are also scripts that are intended to be invoked when relevant packages (e.g., dependent software, conflicting software) are installed or removed.

More details about Debian package management tools can be found in the Debian Policy Manual [8].

2.2 Design Choices and Technical Challenges

Our goal is to build a patching tool that can take *existing* patches intended for online systems and apply them *offline* to a large collection of dormant VM images in a manner that is *safe* and *scalable*. By safety we mean that applying the patch offline achieves the same effect on the persistent file systems in the images as applying it online. By scalability we mean that the tool has to scale to thousands, if not millions of VM images. We would like to point out that in this paper we only consider dormant VM images that are completely shutdown; VM images that contain suspended VMs are out of the scope of this paper.

We made a conscious design decision to be backward compatible with an existing patch format. It is tempting to go with a "clean slate" approach, where we define a new VM-friendly patch format and associated tools that do not make the assumption of a running system at the time of patch application. While this is indeed our long-term research goal, we think its adoption will likely take a long time, given the long history of the traditional online patching model and the fact that it is an entrenched part of today's IT practices, ranging from software development and distribution to system administration. Thus, we believe that an interim solution that is backward compatible with existing patch format, and yet works in an offline manner and provides much improved scalability, would be desirable.

Several technical challenges arise in implementing such a scalable offline patching tool. The most outstanding challenges come from the fact that all current patching solutions are designed for running systems. These patching solutions require the system to be running to execute many transactions, such as related component discovery and notification, environment specific customization, and application or service restart. Another challenge is to scale the patching tool to an extent that it can be routinely run in a cloud environment where the image collection is expected to be huge. We next elaborate these challenges in detail.

Identifying Runtime Dependences: The current software industry is centered around running systems and so are the available patching solutions. A running system provides a convenient environment to execute the installation scripts in the patch. The installation scripts query the configuration of the running system in order to customize the patch appropriately for the system. Some scripts also restart the patched software at the end of the patching process to make sure its effect takes place. Some patches require running daemons. For example, some software stores configuration data in a database. A patch that changes the configuration requires the database server to be running in order to perform schema updates.

The challenge is to separate runtime dependences that can be safely emulated (such as information discovery that only depends on the file system state) or removed (such as restarting the patched software) from the ones that cannot (such as starting a database server to do schema updates). We address this challenge by a combination of manual inspection of commands commonly used in scripts (performed only once before any offline patching) and static analysis of the scripts.

Removing Runtime Dependencies: Once we identify runtime dependences that can be safely emulated or removed, the next challenge is to safely remove these runtime dependences so that the patch can be applied to a VM image offline and in a manner that does not break backward compatibility. Our solution uses a script rewriting approach that preserves the patch format and allows a patch intended for an online system to be applied safely offline in an emulated environment.

Patching at a Massive Scale: As the adoption of virtualization and cloud computing accelerates, it is a matter of time before a cloud administrator is confronted with a collection of thousands, if not millions of VM images. Just moving from online to offline patching is not sufficient to scale to image libraries of that magnitude. We address this challenge by leveraging Mirage's capabilities in efficient storage and manipulation of VM images [23].

```
1  if [ "$1" = "configure" ]; then
2    if [ -e /var/run/dbus/pid ] &&
3      ps -p $(cat /var/run/dbus/pid); then
4      /usr/share/update-notifier/notify-reboot-required
5      ...
6   fi
7  fi
8  ...
9  if [ -x "/etc/init.d/dbus" ]; then
10  update-rc.d dbus start 12 2 3 4 5 . stop 88 1 .
11  if [ -x "‘which invoke-rc.d‘" ]; then
12     invoke-rc.d dbus start
13  else
14     /etc/init.d/dbus start
15  fi
16 fi
```

Figure 1: Excerpts of the `dbus.postinst` script

3 Approach

It seems plausible that patching VM images offline would work, given the fact that the goal of patching is mainly to replace old software components, represented as files in the file system, with new ones. Indeed, to patch an offline VM image, we only care about the changes made to the file system in the VM image; many changes intended for a running system do not contribute to the VM image directly.

One straightforward approach is to perform the file replacement actions from another host, referred to as the *patching host*. Specifically, the patching host can mount and access an offline VM image as a part of its own file system. Using the `chroot` system call to change the root file system to the mount point, the patching host can emulate an environment required by the patching process on a running VM and perform the file system actions originally developed for patching a running VM. For the sake of presentation, we call this approach *simple emulation-based patching* and call the environment set up by mounting the VM image and changing the root file system to the mount point the *emulated environment*.

Unfortunately, our investigation shows that the installation scripts used by the patching process pose a great challenge to simple emulation-based patching. Many patches use scripts to perform pre-installation and post-installation configurations, such as detecting system environment to perform conditional actions, restarting a patched daemon, and notifying relevant software components or the system about the updates so that they can take extra actions.

Figure 1 shows two segments of code from `dbus.postinst`, the post-installation script in the `dbus` package. The first segment (lines 1 to 7) detects possibly running `dbus` processes and sends a reboot notification to the system if there exists one. The second segment (lines 9 to 16) restarts the patched `dbus` daemon so that the system begins to use the updated software. Both segments depend on a running VM to work correctly. The simple emulation-based patching will fail when coming across this script.

To address this challenge, we look further into the internals of patching scripts. After analyzing patching scripts in more than one thousand patching instances, we made some important observations. First, most commands used in the patching scripts are *safe* to execute in the emulated environment, in the sense that *they do not generate undesirable side effects on the persistent file system that would make the patched VM image different from one patched online except for log files and timestamps*. Examples of such commands include the test commands in lines 2, 9 and 11, `cat` in line 3, `/usr/share/update-notifier/notify-reboot-required` in line 4, `update-rc.d` in line 10, and `which` in line 11. Second, some command executions have no impact on the offline patching and thus can be skipped. For example, `invoke-rc.d` in line 12 of Figure 1 is supposed to start up a running daemon, and its execution has no impact on the persistent file system. Thus, we can just skip it. We call such code *unnecessary code*. Third, there are usually more than one way to achieve the same purpose. Thus, it is possible to replace an unsafe command with a safe one to achieve the same effect. For example, many scripts use `uname -m` to get the machine architecture; unfortunately, `uname -m` returns the architecture of the patching host, which is not necessarily the architecture for which the VM image is intended. We can achieve the same purpose by looking at the file system data, for example, the architecture information in the ELF header of a binary file.

Motivated by these observations, in this paper, we propose a systematic approach that combines safety analysis and script rewriting techniques to address the challenge posed by scripts. The safety analysis examines whether it is safe to execute a script in the emulated environment, while the rewriting techniques modify unsafe scripts to either eliminate unsafe and unnecessary code, or replace unsafe code with safe one that achieves the same purpose. Our experience in this research indicates that the majority of unsafe scripts can be rewritten into safe ones, and thus enable patches to be applied to offline VM images in the emulated environment.

However, not all script can be handled successfully in this way. We find some patching instances, after safety analysis and rewriting, still fail in the emulation-based environment. Some patches have requirements that can only be handled in a running environment. For example, the post-installation script in a patch for MySQL may need to start a transaction to update the administrative tables of the patched server. As another example, `mono`, the open source implementation of C# and the Common Language Runtime, depends on a running environment to apply the update to itself.

Given the above discussion, we adopt a hybrid approach in the development of Nüwa. Figure 2 shows an overview of the Nüwa approach. When presented with a patch, Nüwa first performs safety analysis on the patching scripts included in the original patch. If all scripts are safe, Nüwa uses simple emulation-based patching directly to perform offline patching. If some scripts are unsafe, Nüwa applies various rewriting techniques, which will be discussed in detail in Section 4, to these scripts, and performs safety analysis on the rewritten scripts. If these rewriting techniques can successfully convert the unsafe scripts to safe ones, Nüwa will use simple emulation-based patching with the rewritten patch to finish offline patching. However, in the worst case, Nüwa may fail to derive safe scripts through rewriting, and will resort to online patching. In reality, we have found such cases to be rare – our results show that less than 1% of the packages tested in our experiments fall into this category (Section 6.1).

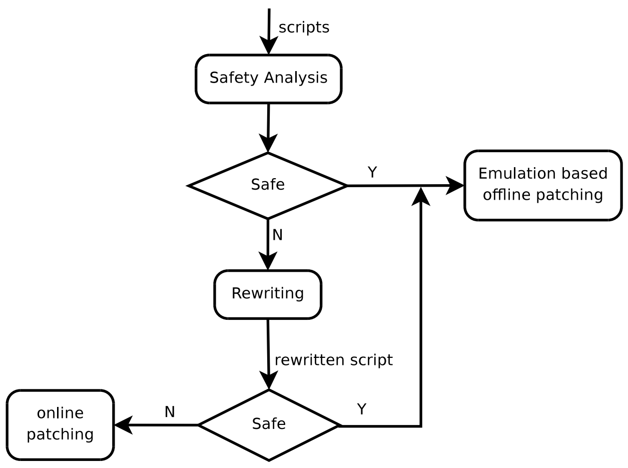

Figure 2: Overview of the Nüwa approach

The online patching may take different forms. For performance reasons, Nüwa takes an automated online patching approach. Specifically, Nüwa inserts the patch data into the VM image through the emulated environment and then schedules a patching process at boot time by modifying the booting script in the VM image. The idea is to have the VM run the patching process before it has a chance to interact with other systems (that is, before any networking capability is established during the boot process). Nüwa then boots the VM, performs online patching, and shuts down the VM automatically once the patching is complete. Note that similar techniques have been proposed before (e.g., [11, 25]).

In addition to patching individual VM images, Nüwa also leverages VM image manipulation technologies to further improve scalability. In particular, Nüwa uses features of the Mirage image library [23] to enable scalable patching of a large number of VM images in batch.

To distinguish between the two variations of Nüwa, we refer to the former as *standalone Nüwa*, and the latter, which leverages Mirage, as *Mirage-based Nüwa*. In the remainder of this paper, we describe the novel techniques developed for offline patching in the context of both standalone Nüwa and Mirage-based Nüwa.

4 Safety Analysis and Script Rewriting

This section explains how safe patch scripts are identified and, when possible, unsafe scripts are transformed into safe scripts. The analysis is based on three concepts — impact, dependence, and command classification, which are defined in Section 4.1. Section 4.2 presents rewriting techniques that, using information from safety analyses, convert many unsafe scripts into safe scripts. Section 4.3 describes how these techniques are put together to develop standalone Nüwa.

In our implementation, safety analysis and script-rewriting run immediately before `dpkg` executes a patch script. As a result, analyses and transformations have access to the script's actual environment and arguments and to the image's filesystem state.

Patch scripts in Debian are SUSv3 Shell Command Language scripts [18] with three additional features mandated by the Debian Policy Manual [8]. Shell scripts are executed by a interpreter that repeatedly reads a command line, expands it according to a number of expansion and quoting rules into a command and arguments, executes the command on the arguments, and collects the execution's output and exit status. The language is very dynamic (for example, command-lines are constructed and parsed dynamically), which forces our analyses and transformations to be conservative. Nonetheless, simple, syntax-directed analyses and rewritings suffice to convert unsafe scripts to safe versions for 99% of the packages we considered.

4.1 Impact, Dependence, and Command Classification

The goal of command classification is to divide a script's command lines into three categories: (1) safe to execute offline, (2) unsafe to execute offline, and (3) unnecessary to execute offline. To classify command lines, we divide a running system into a "memory" part and a "filesystem" part, and determine which parts may influence or be influenced by a given command line. The intuition is that the "filesystem" part is available offline but the "memory" part requires a running instance of the image that is being patched.

We say that a command-line execution *depends on the filesystem* if it reads data from the filesystem or if any of its arguments or inputs flow from executions that depend on the filesystem. An execution *impacts the filesystem* if it writes data to the filesystem or if its output or exit status flow to executions that impact the filesystem.

Table 1 lists some commands whose executions impact the filesystem:

Table 1: Commands with FS-only impacts

Command Type	Example Commands
File attribute modification	`chown, chmod, chgrp, touch`
Explicit file content modification	`cp, mv, mknode, mktemp`
Implicit file content modification	`adduser, addgrp, remove-shell`

We say that a command-line execution *depends on memory* if it inspects any of a number of volatile components of the system's state (perhaps by listing running processes, opening a device, connecting to a daemon or network service, or reading a file under `/proc` that exposes kernel state) or any of its arguments or inputs flow from executions that depend on memory. An execution *impacts memory* if it makes a change to a volatile component of the system's state that outlives the execution itself, or if its output or exit status flow to executions that impact the filesystem.

Note that all executions have transient effects on volatile state: they allocate memory, create processes, cause the operating system to buffer filesystem data, and so forth. For the purposes of classification, we do not consider these effects to be impacts on memory; we assume that other command-line executions do not depend on these sorts of effects.

Table 2 lists some commands that impact or depend on memory.

Table 2: Commands with memory impact or dependence

Command Type	Example Commands
Daemon start/stop	`invoke-rc.d, /etc/init.d/`
Process status	`ps, pidof, pgrep, lsof, kill`
System information inquiry	`uname, lspci, laptop-detect`
Kernel module	`lsmod, modprobe`
Others	Database update, mono gac-install

The definitions for command-line executions are extended to definitions for static command lines. A command line depends on memory (or the filesystem) if any of its executions depend on memory (or the filesystem). A command line impacts memory (or the filesystem) if any of its executions impact memory (or the filesystem).

To seed impact and dependence analysis, we manually inspected all commands used in patch scripts to determine their intrinsic memory and filesystem impacts and dependences. This might seem to be an overwhelming task but, in practice, scripts use very few distinct commands; we found only about 200 distinct commands used by more than 1,000 packages. It may be possible to derive this information by instrumenting command executions. In practice, we expect that it would be provided by package maintainers.

Our analysis concludes that a static command-line depends on memory if one of the following holds:

- The command is unknown.
- The command has an intrinsic memory dependence.
- One or more of the arguments is a variable substitution.
- The input is piped from a command that depends on memory.
- The input is redirected from a device, a file under `/proc`, or from a variable substitution.

The rules for filesystem dependences and for impacts are similar. Note that the analysis errs on the side of finding spurious dependences and impacts. That is, these analyses are simple "may-depend/may-impact" analyses, which are both flow and context insensitive.

Table 3: Command classification

Dependence on FS	Dependence on Memory	Impact on Memory	Impact on FS	Safety
Yes/No	No	No	Yes/No	Safe
Yes/No	No	Yes	Yes	Unsafe
Yes/No	Yes	No	Yes	Unsafe
Yes/No	Yes	Yes	Yes	Unsafe
Yes/No	No	Yes	No	Unnecessary
Yes/No	Yes	No	No	Unnecessary
Yes/No	Yes	Yes	No	Unnecessary

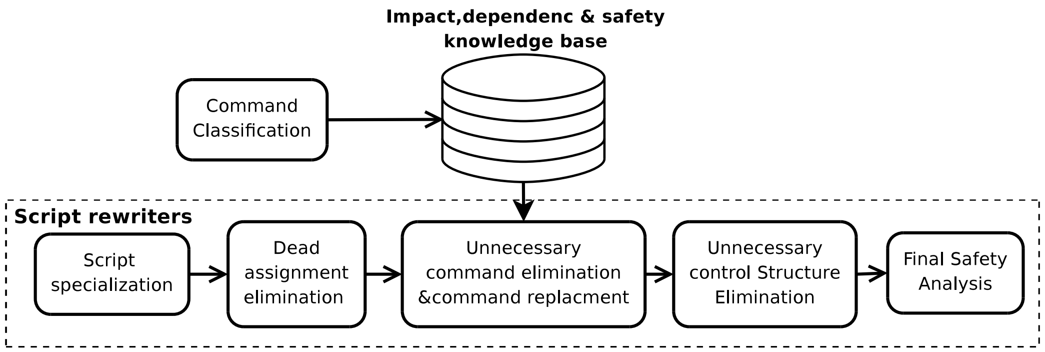
Impact,dependenc & safety knowledge base
Command Classification
Script rewriters
Script specialization
Dead assignment elimination
Unnecessary command elimination &command replacment
Unnecessary control Structure Elimination
Final Safety Analysis

Figure 3: Flow of script analysis and rewriting

Table 3 shows how each command line's classification as safe, unsafe, or unnecessary is determined from its filesystem and memory impacts and dependences. Safe command lines do not depend on or impact memory. These are the commands that can and should be executed offline. Script rewriting preserves these commands. Unnecessary command lines have no impact on the filesystem. There is no reason to execute them offline because they do not change the image. In fact, if they depend on or impact memory, then they must be removed because they might fail without a running instance. Script rewriting removes these commands. Unsafe command lines may execute incorrectly offline because they depend on or impact memory and also impact the filesystem. In some cases, script rewriting cannot remove these command lines because their filesystem impacts are required. If any unsafe command line cannot be removed, then the patch cannot be executed offline.

4.2 Rewriting Techniques

Figure 3 shows the rewriting techniques that Nüwa applies before executing each patch script. Rewriting a script can change the results of safety analysis, so Nüwa reruns safety analysis after applying these techniques. If safety analysis proves that all command lines in the script are safe, then the rewritten script is executed offline. Otherwise, Nüwa resorts to online patching.

Nüwa currently applies five rewriting techniques, which are described below. For clarity, the presentation does not follow the order in which the techniques are applied (that order is shown in Figure 3). The first two techniques consider command-lines, annotated by safety analysis, in isolation; the last three analyze larger scopes.

Unnecessary Command Elimination: This technique removes unnecessary commands, which, by definition, have neither direct nor indirect impact on the filesystem. Figure 4 shows examples found in actual patch scripts.

```
/etc/init.d/acpid
/etc/init.d/cupsys
killall CONSTANT
```

Figure 4: Examples of command lines that are removed by unnecessary command elimination

Command Replacement: Some command lines that depend on memory can be replaced with command lines that depend only on the filesystem. This often happens with commands that need information about the system, in particular when the information is available both in the filesystem and, if there is a running instance, in memory.

```
uname -m
  -> dpkg --print-architecture
```

```
uname -s
  -> echo "Linux"
```

Figure 5: Memory-dependent command lines and their replacements

For example, the `uname` command prints system information; depending on its arguments, it will print the hostname, the machine hardware name, the operating system name, or other fields. `uname` gets its information from the kernel through the `uname` system call. Without a running instance, information from the kernel cannot be trusted. However, certain fields are statically known constants or available through commands that depend only on the filesystem; Figure 5 shows two examples.

Note that command replacement not only removes memory-dependent commands but also ensures that the offline script uses values appropriate to the image instead of values from the host. Nüwa's implementation of command replacement consults a manually constructed table of command lines and their known replacements.

Unnecessary Control-structure Elimination: This technique, a generalization of unnecessary command elimination, removes compound commands like `if` and `case` statements.

Figure 6 shows an example. Both the true branch and the false branch of the `if`-statement are unnecessary and would be eliminated by unnecessary command elimination. The conditional would not be eliminated because safety analysis conservatively assumes that all conditionals impact both memory and the filesystem through control-flow. By contrast, unnecessary control-structure elimination eliminates the entire `if`-statement because, after eliminating both branches of the `if`-statement, the conditional is unnecessary: It clearly has no filesystem impact through control-flow or any other means.

Before rewriting:

```
if [ -x "‘which invoke-rc.d‘" ]; then
   invoke-rc.d dbus start
else
   /etc/init.d/dbus start
fi
```

After rewriting:

```
All eliminated
```

Figure 6: Example of control structure analysis (from `dbus.postinst`)

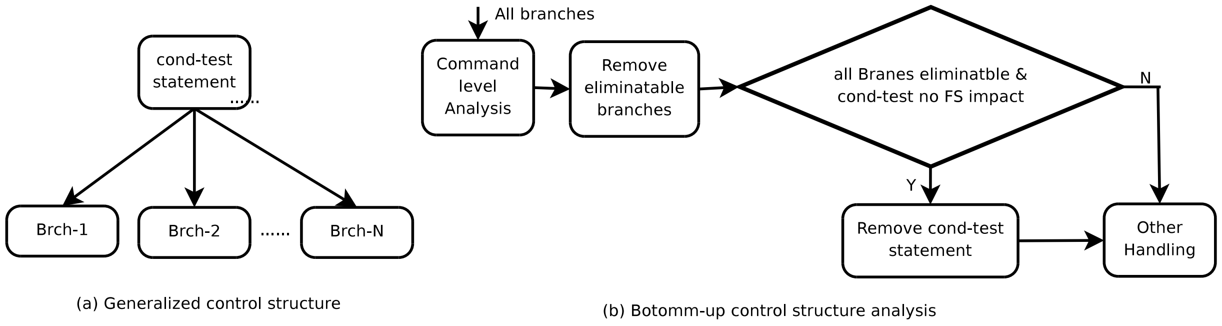
cond-test statement
......
Brch-1
Brch-2
......
Brch-N
(a) Generalized control structure
All branches
Command level Analysis
Remove eliminatable branches
all Branes eliminatble & cond-test no FS impact
N
Y
Remove cond-test statement
Other Handling
(b) Botomm-up control structure analysis

Figure 7: Unnecessary control-structure elimination

Unnecessary control-structure elimination proceeds bottom-up as shown in Figure 7. For each control structure, we first try to eliminate all statements in each branch of the structure. If all statements in every branch can be eliminated, we consider the conditional itself: If it no longer impacts the filesystem, the entire control structure is removed.

Note that Nüwa applies unnecessary control-structure analysis to many kinds of compound commands and command lists, including the `case` construct and command lists built from the short-circuiting statements (|| and &&).

Script Specialization: This technique removes command lines and control structures that cannot execute, given the script's actual environment and arguments and the VM image's filesystem state. Recall that this context is available because safety analysis and script-rewriting run immediately before `dpkg` executes a patch script.

Figure 8 shows an example, which was extracted from the post-installation script for the `acpid` package. Except during error recovery, `dpkg` calls post-installation scripts with `configure` as the first positional parameter (that is, `$1`). Therefore, the `case` statement can be replaced with the first branch. Next, because the rest of the script changes neither `/etc/init.d/hal` nor `/var/run/hald/hald.pid`, the conditional can be evaluated at rewriting time; in this case, the conditional is false and the false branch is empty so the entire if statement is removed.

Before rewriting:

```
HAL_NEEDS_RESTARTING=no
case "$1" in
  configure)
    if [ -x /etc/init.d/hal ] &&
      [ -f /var/run/hald/hald.pid ]; then
      HAL_NEEDS_RESTARTING=yes
      invoke-rc.d hal stop
    fi
    ;;
  reconfigure)
    ...
esac
```

After rewriting:

```
HAL_NEEDS_RESTARTING=no
```

Figure 8: Example of script specialization (from `acpid.postinst`)

The current implementation of script specialization is a collection of ad hoc rewriting passes, which Nüwa applies before applying any other rewriting techniques. One pass replaces positional parameters with actual parameters. Another evaluates conditionals built from filesystem tests, when the tests depend only on the initial filesystem state. A third evaluates the command line `dpkg --compare-versions`, which is used frequently and whose result can be determined from the VM image's package database.

All passes are conservative and err on the side of missing rewriting opportunities. For example, positional-parameter replacement leaves the script unchanged if the script uses the `shift` statement, which renames the positional parameters.

Dead-assignment Elimination: This technique removes assignments to unused variables. Some dead assignments come from the original scripts; others are created by script specialization, which can convert conditionally dead assignments to dead assignments.

Before rewriting:

```
LAPTOP=""
if [ -n "$(which laptop-detect)" ]; then
    if laptop-detect >/dev/null; then
            LAPTOP=true
    fi
fi
```

After rewriting:

```
All eliminated
```

Figure 9: Example of dead-assignment elimination (from `xfonts-scalable.postinst`)

Figure 9 shows an example of dead assignment from an original script, namely `xfonts-scalable.postinst`. In this script, the `laptop-detect` command is intrinsically memory-dependent. If its result flows to a command line that impacts the filesystem, the script would be unsafe. Fortunately, the `LAPTOP` variable is unused in the rest of the script. Removing its assignment leaves the body of the inner `if` statement empty, which makes the conditional unnecessary, which in turn allows the entire inner `if` statement to be removed. The outer `if` statement is then removed in a similar fashion.

The first assignment in Figure 8, which is conditionally dead in the original script, could be transformed into a dead assignment by script specialization.

Dead-assignment elimination depends on a syntax-directed data-flow analysis of the main body of the script. An assignment is *dead* if the assigned value cannot reach a *use* before reaching the end of the script or another assignment; the analysis conservatively judges an assignment to be dead if it it does not occur in a loop and is followed by another assignment in the same syntactic scope, with no intervening uses in any syntactic scope, or if no uses follow at all. Function bodies are not considered, except that any use of a variable within a function body is considered reachable from any assignment to that variable in the entire program.

4.3 Implementation of Standalone Nüwa

The implementation of standalone Nüwa is composed of three parts: (1) setting up the emulated environment, (2) rewriting scripts and safety analysis, and (3) applying the patch to a VM image.

As discussed in Section 3, setting up the emulated environment is achieved through a chroot jail [19]. In this emulated environment, we use the Advanced Package Tool (APT) [5] to download patches, using the patching host's connection to the network. APT also helps resolve package dependences, but the real job of applying patches is done by `dpkg` [7], which gets input from the package file, extracts the file content and hook scripts, and performs the patch. In general, `dpkg` will fork a new process to execute each hook script. We extend `dpkg` by inserting a script rewriter before the execution of each hook script.

The implementation of the script rewriter is based on the source code of the Debian Almquist Shell (i.e., `dash`) [6], the default shell interpreter for system tasks in Debian-like environments. The script rewriter applies the rewriting techniques discussed in Section 4.2 to each script through multiple passes of the script. The classification of known commands and related information used in these rewriting techniques are preconfigured in a file, which can be updated as needed.

The safety analysis is performed by scanning a (rewritten) script to decide if unsafe commands remain. If the answer is "no" for all the (rewritten) scripts involved in a patch, Nüwa considers the (rewritten) patch to be safe and applies it in the emulated environment. However, if after script rewriting there are still unsafe commands in any of the scripts in a patch, Nüwa considers the patch not applicable offline, and resorts to automated online patching, as discussed in Section 3.

Though our current implementation is based on the Debian package manager, it should be possible to port this implementation to other Linux distributions, such as RPM-based Linux systems. Our initial investigation into porting our tools to RPM-based Linux systems indicates that, when it comes to executing scripts, RPM is very similar to dpkg. One difference is where the hook scripts are located. Another difference is that the two kinds of distributions use different shells (i.e., `dash` and `bash`). We plan to port our tools to support RPM in the near future.

5 Scalable Batch Patching

A motivating assumption of this work is that, as cloud computing becomes more widely adopted, image libraries will grow to contain thousands or perhaps even millions of images, many of which must be patched as new vulnerabilities are discovered. Even with the offline patching techniques presented in Section 4, patching this many images individually would take a significant amount of time.

This section explains an approach to batch patching a large number of images offline that exploits an observation and a conjecture about patching images. The observation is that, if the same patch is applied to two similar images, then any given patch-application step is likely to have the same effect on both images. For example, the same files will be extracted from the patch both times. The conjecture is that the images that must be patched are likely to be similar to one another; this conjecture seems particularly reasonable for clouds (such as Amazon's EC2 [2]) that encourage users to derive new images from a small set of base images.

Nüwa's batch patching implementation exploits the Mirage image library, which exposes image similarity by storing images in a special format called the Mirage Image Format (MIF) [23]. The rest of this section first gives a brief overview of Mirage and then describes Nüwa's approach to and implementation of batch patching.

5.1 Overview of Mirage

The Mirage image library maintains a collection of virtual-machine images and provides an image-management interface to users: users can import images into the library, list existing images in the library, check out a particular image, and check in updates of the image to create either a new image or a new version of the original image. A separate interface allows system administrators to perform system-wide operations, such as backup, virus scan, and integrity verification of all image content.

A design goal of Mirage is to support operations on images as structured data. To this end, Mirage does not store images as simple disk images. Instead, when an image is imported into the library, Mirage iterates over the image's files, storing each file's contents as a separate item in a content-addressable store (CAS); the image as a whole is represented by a manifest that refers to file-content items and serves as a recipe for rebuilding the image when it is checked out. An earlier paper [23] described this format and explained how it allows certain operations on images to be expressed as fast operations on the image's manifest. For example, creating a file, assuming that the desired contents are already in the CAS, reduces to adding a few hundred bytes to the manifest.

Mirage's new *vmount* feature, which was not described in the earlier paper, allows users to mount library images without rebuilding them. Vmount is implemented as a FUSE [24] daemon and fetches data from the CAS as it is demanded; by contrast, checking out an image requires fetching every file's contents from the CAS. Vmount also implements a new extended filesystem attribute that allows direct manipulation of the MIF manifest. For each regular file, the value of this attribute is a short, unique identifier of the file's contents. Setting the attribute atomically replaces the file's contents with new contents.

After modifying an image through Vmount, the user can check in the changes as a new image or a new version of the original image. The original image is not disturbed, and the time to check in is proportional to the amount of new data instead of to the size of the image.

Vmount has three benefits for batch patching. First, there is no need to rebuild each image. Arguably, this is merely a workaround for a problem created by the decision to store images in MIF.

Second, if two images share data in the CAS and are patched sequentially through Vmount, then reading the shared data the second time is likely to be fast, because the data will be in the host's buffer cache. By contrast, if two disk images are patched sequentially, then the fact that they share data is effectively hidden from the host's operating system [3].

The largest benefit is that Vmount allows batch patching to operate on manifests without major modifications of system tools like `dpkg`. Time-critical patching steps can be changed to use the new filesystem attribute, without creating a dependence on the manifest format, while less profitable steps continue to use the normal filesystem interface.

5.2 Batch Patching via Mirage

A straightforward way to patch a batch of images is to iterate the patching process for individual images. For images stored in Mirage, each iteration mounts an image with Vmount, applies the patch as shown in Figure 2 [4], and checks in the modified image.

[3]Another, albeit unwieldy, solution to this problem is to compose images carefully from differencing disks.

[4]If the patch must be applied online, then the image must be rebuilt.

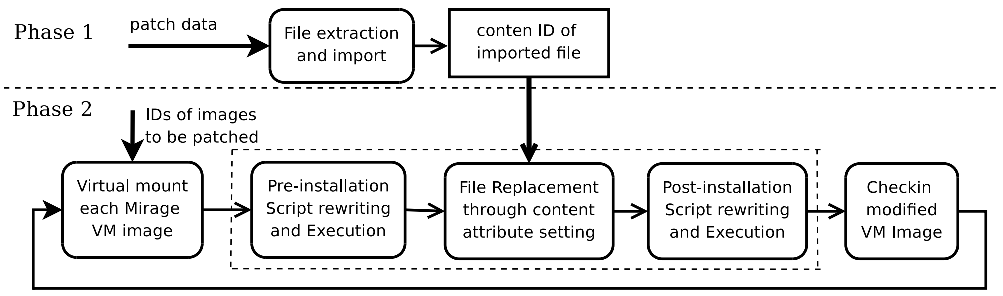
Phase 1
patch data
File extraction and import
conten ID of imported file
Phase 2
IDs of images to be patched
Virtual mount each Mirage VM image
Pre-installation Script rewriting and Execution
File Replacement through content attribute setting
Post-installation Script rewriting and Execution
Checkin modified VM Image

Figure 10: Batch patching VM images via Mirage

Our approach begins with this straightforward method and optimizes it by moving invariant operations out of the loop that visits each image. Currently, Nüwa optimizes one source of invariant operations: unpacking the patch, which copies the patch's files to the image and, ultimately, adds their contents to the Mirage CAS. These copies and CAS additions are good operations to move out of the loop because they consume most of the time of applying most patches; in future work, we plan to hoist more invariants out of the loop.

Figure 10 shows the two phases of batch patching via Mirage. Phase 1 performs the loop-invariant operation: Nüwa extracts the patch's files and imports them into Mirage. The result is a list of content identifiers, one for each file.

In phase 2, Nüwa iterates over the images. For each image, Nüwa

1. mounts the image with Vmount;
2. rewrites and executes the pre-installation scripts;
3. emulates the "unpack" step of `dpkg`, using the Mirage filesystem attribute to set the contents of the patch's files;
4. rewrites and executes the post-installation scripts; and
5. checks in the modified VM image.

Of course, if script rewriting ever fails to produce a safe script, then Nüwa resorts to online patching.

The program that emulates the "unpack" step of `dpkg` is approximately 100 lines of Python code. There are some `dpkg` features that it does not handle correctly yet, including diversions and the "Replaces" field. The lack of support for these features did not affect the experiments in Section 6.

6 Experimental Evaluation

We have implemented both standalone Nüwa and Mirage-based Nüwa. Our implementations assume a Linux host system. We have tested the standalone Nüwa on patching hosts running CentOS 5.2, Ubuntu 9.0.4 and OpenSuSE 11.1. However, Mirage-based Nüwa currently only works on patching host running SuSE Linux, due to its dependence on Mirage, which only runs on SuSE Linux systems right now. Our implementations currently support VM images of any Linux distributions based on Debian package management tools, such as Debian, Ubuntu, and Knoppix. Porting our tools to handle other Linux distributions (e.g., Redhat Package Manager (RPM)-based Linux) is technically straightforward, but requires additional implementation effort.

We performed two sets of experiments to evaluate the performance of Nüwa, one for patching individual VM images offline, and the other for Mirage-based offline patching in batch. Both sets of experiments were performed on a DELL OptiPlex 960 PC, with a 3GHz Intel Core 2 Duo CPU and 4GB DDR2 memory. In our evaluation, we used the x86-64 version of OpenSuSE 11.1 version as the host OS. For compatibility reasons, we updated its kernel to version `2.6.31.11-0.0.0.2.9c60380-default`.

Since our focus is to evaluate the new techniques proposed for offline patching, in this section, we use Nüwa to only refer to its offline portion, and treat its automated online patching separately.

6.1 Patching Individual VM Images

The objective of this set of experiments is two-fold: First, we would like to evaluate the correctness of our offline patching approach used in Nüwa (i.e., whether the offline patching approach has the same effect on the VM images as online patching). Second, we would like to see the efficiency of our offline patching approach in Nüwa compared with the online patching approach.

In this set of experiments, we used the Linux Kernel-based Virtual Machine (KVM) [9] to start instances of VM images for online patching. For offline patching, we used the VMware disk library to mount the VM images in the host environment. Our tool can be logically decomposed into two parts: the script rewriter and the patch applier. We copied both components into the mounted VM image, with the patch applier replacing the original package installer inside the target VM image.

To perform the evaluation, we first created an empty disk image in the flat VMDK disk format with the `kvm-img` image creation tool. We then brought this disk image online through KVM and installed a default configured 64-bit Ubuntu-8.04 inside. This was used as the base VM image for both offline and online patching in our experiments.

We gathered all 406 patches available for the base VM image (64-bit Ubuntu-8.04) on October 26, 2009. The correctness of offline patching is verified by comparing the result of the offline patching with that of online patching: If two VM images, which are obtained through patching the base VM image online and offline, respectively, differ only in log files and timestamps, we consider the offline patching to be correct. To further evaluate the effectiveness of the rewriting techniques, we used the simple emulation-based patching mentioned in Section 3 as a reference.

Table 4 shows the experimental results for evaluating the correctness of our techniques. Nüwa can successfully apply 402 out of the 406 patches offline, achieving a 99.0% success ratio. The results also show that the rewriting techniques contributed significantly to the success; they helped improve the success ratio by about 10%. Note that the failure cases are failures of offline patching, not of Nüwa; Nüwa automatically detects all of these failures and can cope with them through automatic online patching, as discussed in Section 3.

Table 4: Comparison of the two offline patching methods

	# successes	# failures	success ratio
Simple emulation	369	37	90.9%
Nüwa	402	4	99.0%

The four failure cases are the `mono-gac` package[5] and three other packages that depend on `mono-gac`. Through further analysis, we found that `mono-gac` failed because the installer of Mono needed to access some kernel information (e.g., `/proc/cpuinfo`, `/proc/sys/fs/binfmt_misc`, and `/proc/self/map`) in order to work correctly. This information cannot be retrieved in the emulated environment.

To compare the efficiency of Nüwa's offline patching techniques with that of online patching, we performed another set of experiments. We assumed the most efficient form of online patch, automated online patching. This is in fact our fall-back solution in case offline patching is unsafe. We believe this is the most efficient form of online patching, since it automates all the steps required in online patching.

We collected two sets of data from these experiments. The first is the time (in seconds) required to apply each applicable patch to the base VM image through the offline patching approach in Nüwa, and the second is the time needed to apply the same set of patches through automated online patching.

[5] `mono-gac` is a utility to maintain the global assembly cache of mono, an open source implementation of $C\sharp$ and the CLR

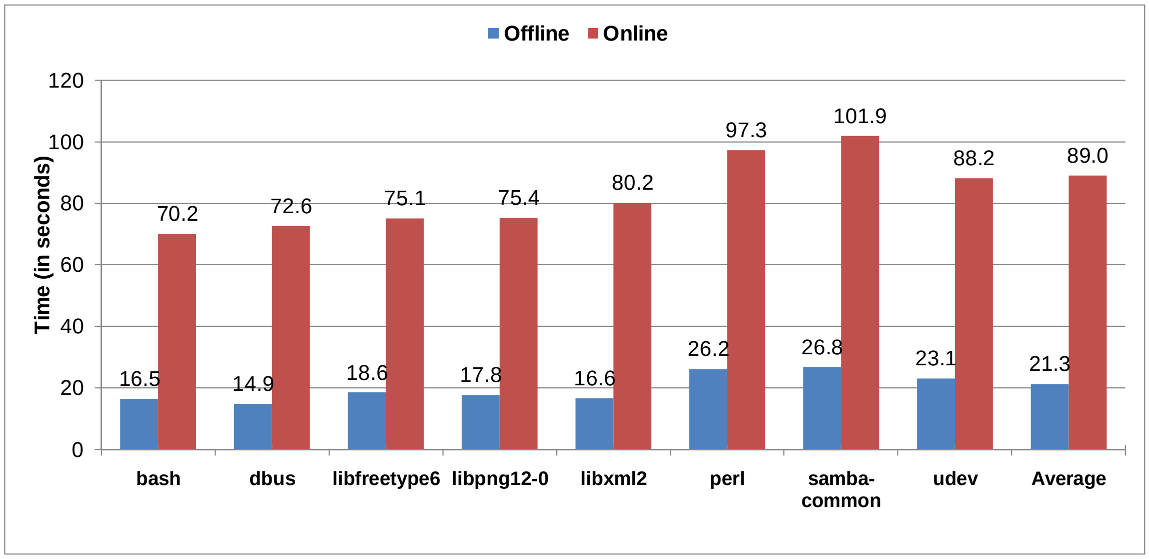

Figure 11: Time used by offline and online patching ("Average" is computed over 402 applicable packages)

Figure 11 shows the time (in seconds) required to apply some applicable patches to the base VM image through the Nüwa offline patching and the automated online patching, respectively. Due to the limited space, we only show the timing results for eight selected patches and the average for all 402 applicable patches. On average, the Nüwa offline approach takes only 23.9% of the time required by automated online patching (a factor of 4 speedup). This improvement, combined with the fact that Nüwa needs much less human intervention and physical resources, show that it brings significant benefits to patching VM images.

This set of experiments demonstrates that Nüwa's offline patching techniques, particularly the rewriting techniques, are effective and that offline patching using Nüwa can significantly reduce the overhead involved in patching.

6.2 Batch Patching via Mirage

The primary objective of this set of experiments is to measure the scalability offered by Mirage-based Nüwa by comparing the performance of Mirage-based batch patching with that of one-by-one patching.

Table 5: Basic Ubuntu tasks

No.	Task Name	No.	Task Name
1	lamp-server	2	mail-server
3	dns-server	4	openssh-server
5	print-server	6	samba-server
7	postgresql-server	8	ubuntustudio-audio
9	ubuntustudio-audio-plugins	10	ubuntustudio-graphics
11	ubuntustudio-video	12	ubuntu-desktop

We generated 100 VM images using 32-bit Ubuntu 8.04 as the base operating system for this set of experiments. The Ubuntu installer can install a support for a number of basic, predefined tasks; some of these tasks are for running specific servers, while others are for desktop use. We generated test VM images from 100 randomly selected combinations of 12 of these tasks (listed in Table 5).

We retrieved 154 security updates (i.e., security patches) for 32-bit Ubuntu 8.04 from Ubuntu Security Notices [26]. We also retrieved the ranking of Ubuntu packages given by Ubuntu's popularity contest [10], and sorted the 154 security patches accordingly. For our performance evalution, we selected the security updates corresponding to the eight most popular packages (as of January 18th, 2010). These packages are

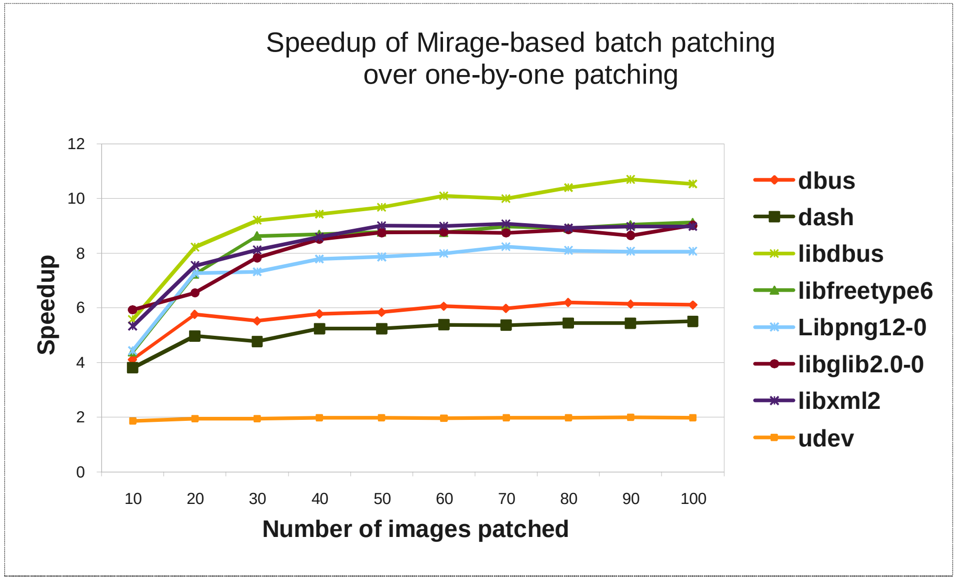

Figure 12: Scalability Evaluation of Mirage-based Nüwa

dash, libdbus, libglib-2.0, libfreetype, udev, libpng, libxml2, and dbus.

For each of the eight patches, we measured the time to apply the patch to the test VM images one-by-one and the time to apply the patch to the test VM images as batches of increasing sizes. Figure 12 shows that for all eight security patches, Mirage-Nüwa achieves considerable speedup over one-by-one patching. The speedup also increases as the number of images patched in a batch increases, and plateaus between 80 and 100 images.

For seven of the eight security patches (udev is the exception), the average speedup over one-by-one patching increases from 5.1 times to 8.5 times as the number of images in a batch increases from 10 to 100. Note that this speedup is on top of the factor of 4 speedup achieved over traditional online patching, thus bringing the total speedup over traditional online patching to about 30 when patching 100 images in a batch.

However, the speedup for udev is much smaller, compared with the other seven patches. In fact, the speedup for udev is only around 2. Further investigation showed that the udev patch spends more time in pre-installation and post-installation scripts than do the others; thus, the file replacement operations constitute a smaller portion of the entire patching process.

This set of experiments demonstrates that Mirage-based Nüwa is scalable and can further improve the performance of offline patching significantly. Overall, Nüwa offline patching is an order of magnitude more efficient than online patching.

7 Related work

Several available commercial tools [11, 25, 27] can apply patches to dormant VM images. But that does not mean the patches are applied in an *offline* manner. As a matter of fact, all of them require the image to be running when the patches are actually installed. Microsoft's Offline VM Servicing Tool [11] first "wakes" up the virtual machine (deploys it to a host and starts it), then triggers the appropriate software update cycle to apply the patches, and finally shuts down the updated virtual machine and returns it to the image library. In the cases of VMware Update Manager [27] and Shavlik NetChk Protect [25], patches are first inserted into image at some specified locations, then applied when the image is powered up. We resort to this approach when Nüwa identifies patches that contain unsafe commands.

In some cases, it is preferable to apply patches online. In general, systems that tend to stay online for a long period of time, such as highly available servers, fall into this category. In those cases, “dynamic update” techniques [1,3,15,17] are used to apply patches to the target software without shutting them down. In contrast, Nüwa targets VM images that have already been shut down and may stay in dormant state for an extended period of time. Thus, these approaches are complimentary to Nüwa.

8 Conclusion

In this paper, we developed a novel tool named Nüwa to enable efficient patching of offline VM images. Nüwa uses safety analysis and script rewriting techniques to convert patches, or more specifically the installation scripts contained in patches, which were originally developed for online updating, into a form that can be applied to VM images offline. Nüwa also leverages the VM image manipulation technologies offered by the Mirage image library [23] to provide an efficient and scalable way to patch VM images in batch. We implemented Nüwa based on the Debian package manager [7], including both a standalone version and a Mirage-based version. We evaluated Nüwa with security patches and VM images configured with popular packages according to Ubuntu popularity contest. Our experimental results demonstrate that 1) Nüwa’s safety analysis and script rewrting techniques are effective – Nüwa is able to convert more than 99% of the patches to safe versions that can then be applied offline to VM images and; 2) the combination of offline patching with additional optimization made possible through Mirage’s efficent image manipulation capabilities allows Nüwa to be an order of magnitude more efficient than online patching.

A limitation of Nüwa is that it currently does not support offline patching of suspended VM images, which also includes a snapshot of the system memory state in addition to the file system.

In our future research, we will investigate techniques to patch suspended VM images. We also plan to port Nüwa to support other popular package managers such as RPM. Finally, we will look into new issues that arise when applying Nüwa in cloud computing environments, such as efficient testing of patched VM images.

References

[1] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: online patches and updates for security. In *SSYM’05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 19–19, Berkeley, CA, USA, 2005. USENIX Association.

[2] Amazon. Amazon elastic compute cloud (EC2). `http://aws.amazon.com/ec2/`.

[3] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In *EuroSys ’09: Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198, New York, NY, USA, 2009. ACM.

[4] CERT. Cert advisory ca-2003-20 w32/blaster worm. `http://www.cert.org/advisories/CA-2003-20.html`, August 2003.

[5] APT community. Advanced packaging tool. `http://en.wikipedia.org/wiki/Advanced_Packaging_Tool`.

[6] Debian community. Debian almquist shell. `http://en.wikipedia.org/wiki/Debian_Almquist_shell`.

[7] Debian community. Debian package manager. `http://www.debian.org/dpkg`.

[8] Debian Community. Debian policy manual. `http://www.debian.org/doc/debian-policy/`, 2009.

[9] KVM community. Linux kernel-based virtual machine. `http://www.linux-kvm.org/`.

[10] Ubuntu Community. Ubuntu popularity contest. `http://popcon.ubuntu.com/`.

[11] Microsoft Corporation. Offline virtual machine servicing tool 2.1. `http://technet.microsoft.com/en-us/library/cc501231.aspx`.

[12] Forbes. Cybersecurity's patching problem. `http://www.forbes.com/2009/09/14/sans-institute-software-technology-security-cybersecurity.html`. Visited on 2009-11-06.

[13] Stefan Frei, Thomas Duebendorfer, Gunter Ollmann, and Martin May. Understanding the Web browser threat. Technical Report 288, TIK, ETH Zurich, June 2008. Presented at DefCon 16, Aug 2008, Las Vegas, USA. `http://www.techzoom.net/insecurity-iceberg`.

[14] Thomas Gerace and Huseyin Cavusoglu. The critical elements of the patch management process. *Commun. ACM*, 52(8):117–121, 2009.

[15] Deepak Gupta and Pankaj Jalote. On line software version change using state transfer between processes. *Softw. Pract. Exper.*, 23(9):949–964, 1993.

[16] Huseyin Cavusoglu Hasan, Hasan Cavusoglu, and Jun Zhang. Economics of security patch management. In *The Fifth Workshop on the Economics of Information Security (WEIS 2006)*, June 2006.

[17] Michael Hicks and Scott M. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1049–1096, November 2005.

[18] The IEEE and The Open Group. The single UNIX specification, version 3. `http://www.unix.org/version3/online.html`, 2004.

[19] P. Kamp and R.N.M. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd Intl. SANE Conference*, 2000.

[20] Cloud Market. The cloud market: EC2 statistics. `http://thecloudmarket.com/stats`.

[21] Microsoft. The microsoft security update release cycle. `http://www.microsoft.com/security/msrc/whatwedo/updatecycle.aspx`.

[22] United States General Accounting Office. Effective patch management is critical to mitigating software vulnerabilities. gao-03-1138t, September 2003.

[23] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120, 2008.

[24] Miklos Szeredi. Fuse: Filesystem in userspace. `http://fuse.sourceforge.net/`, 2010.

[25] Shavlik Technologies. Offline virtual machine image quick start guide. `http://www.shavlik.com/documents/qsg-prt-6-1-offline_vm.pdf`.

[26] Ubuntu. Ubuntu security notices. `http://www.ubuntu.com/usn/`.

[27] VMware. VMware vcenter update manager. `http://www.vmware.com/products/update-manager/`.