Patch Auditing in Infrastructure as a Service Clouds

Lionel Litty *
VMware, Inc.
llitty@vmware.com

David Lie
Department of Electrical and Computer Engineering
University of Toronto
lie@eecg.toronto.edu

Abstract

A basic requirement of a secure computer system is that it be up to date with regard to software security patches. Unfortunately, Infrastructure as a Service (IaaS) clouds make this difficult. They leverage virtualization, which provides functionality that causes traditional security patch update systems to fail. In addition, the diversity of operating systems and the distributed nature of administration in the cloud compound the problem of identifying unpatched machines.

In this work, we propose P2, a hypervisor-based patch audit solution. P2 audits VMs and detects the execution of unpatched binary and non-binary files in an accurate, continuous and OS-agnostic manner. Two key innovations make P2 possible. First, P2 uses efficient information flow tracking to identify the use of unpatched non-binary files in a vulnerable way. We performed a patch survey and discover that 64% of files modified by security updates do not contain binary code, making the audit of non-binary files crucial. Second, P2 implements a novel algorithm that identifies binaries in mid-execution to allow handling of VMs resumed from a checkpoint or migrated into the cloud.

We have implemented a prototype of P2 and and our experiments show that it accurately reports the execution of unpatched code while imposing performance overhead of 4%.

Categories and Subject Descriptors K.6.3 [*Software Management*]: Software maintenance

General Terms Algorithms, Design, Experimentation, Management, Measurement, Security

Keywords virtualization, cloud computing, infrastructure as a service, patch management, application discovery

1. Introduction

A large number of security vulnerabilities stem directly from software implementation flaws in critical code. To maintain the security of a system against attackers, it is critical that patches, which fix these flaws, be applied in a timely manner. As a result, many software packages and operating systems (OSs) contain support for automatic patch installation. These systems are simple – they periodically check a central server for the existence of patches and apply any patches that have not yet been applied to the system they are running on.

However, automatic patch installation cannot always prevent the exploitation of known vulnerabilities and can reduce the stability of computer systems. Patches are not applied instantly as the patch installation system only checks for updates periodically. This creates an exploitable window of vulnerability between the time the patch is disclosed and the time it is applied [4]. In addition, automatic patch systems are unaware of which patches need to be applied, and proactively apply all available patches, even if the patched component is never used. Unfortunately, patches can have unintended side effects, so such unnecessary patches can needlessly cause system failures or performance degradation [2].

This problem is made more acute in a virtualized cloud environment. Public and private virtualized cloud environments offering Infrastructure as a Service (IaaS) have grown in popularity due to their ability to provide elasticity of resources and reduce user costs. These environments have two characteristics that make patch management even more complicated than in a unvirtualized environment. First, virtualization introduces new usage models that break standard automatic patch installation systems [11]. Patch installation systems rely on machines always being powered on so that they can check for updates and apply them. They also assume that time proceeds in a linear fashion so that patches are naturally applied in sequence. Unfortunately, virtual machines (VMs) can be archived and left powered off for long periods of time. They can be rolled back to a previous checkpoint in time, cloned, migrated, created and destroyed easily and frequently. These capabilities skew time in the VM and can cause automatic patch installation systems to fail.

Second, by separating the administration and maintenance of hardware from that of the software, IaaS clouds permit a larger number of administrative domains, resulting in a greater diversity of OSs and software environments. Whereas a single organization with a single IT department may have forced all users to conform to a homogeneous computing environment, a private cloud environment removes that restriction, giving users more freedom. Users may have any OS (i.e. Windows, Linux), any flavor of the OS (i.e. Vista, XP, Ubuntu, Red Hat), and any version level (i.e. Linux kernel version, Windows service pack). In a public cloud, such as those implemented by Amazon's EC2, GoGrid and Mosso, there are no restrictions at all on what OSs and software users may install. The de-federalization of administrative control and greater diversity in software make it difficult for a cloud provider to ascertain the patch level of VMs running on their infrastructure. Unfortunately, the insecurity of a single cloud user can negatively impact both their fellow cloud users and the cloud provider itself [18][26]. Thus, cloud providers are motivated to identify and protect themselves from VMs on their cloud that are vulnerable to attack.

* The majority of this work was done while Lionel Litty was at the University of Toronto

VEE'11 March 9–11, 2011, Newport Beach, California, USA.

In this paper, we demonstrate that virtualization can also solve the patch management problems it creates in cloud environments. We design and implement P2, a patch audit solution that leverages the hypervisor to detect and mitigate vulnerabilities in unpatched software in VMs. P2 has several advantages over existing solutions. First, P2 provides *continuous* protection, and does not fail if the machine is powered off or check-pointed and rolled back. Second, P2 is more *accurate* than existing solutions. P2 only reports unpatched software if it is actually executed, reducing the number of alerts and enabling administrators to apply only the minimal number of patches required. Finally, P2 is *OS-agnostic*, allowing it to work on any standard commodity OS. This allows the cloud provider to have a single patch audit solution, instead of having to support an OS-specific one for every OS their customers use.

P2 accomplishes this by relying on *architectural introspection* [18], which monitors virtual hardware to infer events within a VM. By restricting monitoring to only hardware state, P2 is able to detect unpatched software in VMs without having to rely on any detailed or implementation-specific information about the OS and software in the VM. P2 detects unpatched binary and non-binary software. Binary software denotes software that will execute natively on the underlying processor, such as executables or libraries. Non-binary software generally refers to scripts or byte-code, which will execute with the aid of an interpreter or just-in-time compiler (JIT). However, non-binary software can also include configuration files and other application resources.

To detect the execution of an unpatched binary, P2 monitors memory pages in a way similar to Patagonix [17]. However, Patagonix needs to be invoked before an application starts to identify it, meaning that it cannot be applied to VMs that have been resumed from checkpoints or migrated into a cloud. In contrast, P2 does not suffer from this restriction and can identify applications mid-execution. To detect scripts and byte code that are executed by an interpreter or JIT compiler, P2 uses lightweight, coarse-grain information flow tracking to determine if an unpatched file is read from disk into the address space of an interpreter or JIT compiler that is capable of executing the non-binary file. This allows P2 to detect the execution of unpatched non-binary code with low overhead and few false positives.

When P2 detects unpatched code it can take one of two actions depending on the mode it is used in. In *reporting mode*, it simply reports that a VM has executed unpatched code to the cloud administrator, who can then inform the VM administrator and take actions according to their service agreement (i.e. the VM administrator may have to patch the code or the cloud administrator could adjust firewall rules to prevent exploitation). In *prevention mode*, in addition to reporting the unpatched code, P2 actively prevents the exploitation of the unpatched code by injecting code into the vulnerable process that prevents it from executing any more instructions.

We make three contributions in this paper. First, we perform a study of patches on the Fedora Core 10 and 11 Linux distributions. We find that across both distributions, 102,819 files were updated, of which 23,711 are non-documentation files that may contain executable code or configuration resources. Among these files, only 36% are binary code files. This serves as a strong motivation for the ability to be able to detect the execution of unpatched non-binary code. Second, we describe P2, which uses architectural introspection to detect the execution of unpatched binary and non-binary code. We demonstrate the OS-agnostic property of P2 by running it on both Windows XP and Linux VMs. Finally, we evaluate the effectiveness and performance overhead of P2 by evaluating it on workloads that contain both patched and unpatched code. We find that P2 can accurately detect the execution of unpatched applications with minimal overhead.

We give a detailed motivation for P2 and state its assumptions and guarantees in Section 2. Then we describe our patch survey in Section 3, which illustrates and characterizes the need for monitoring of non-binary files. Section 4 describes the architecture of P2 and implementation details of our prototype are described in Section 5. We then evaluate P2's effectiveness and performance overhead in Section 6. Finally, Section 7 compares P2 to other related work and we conclude in Section 8.

2. Overview

2.1 Motivation

A survey conducted by Bellissimo et al. in 2006 [3] found that 69 out of 71 CERT Technical Cyber Security Alerts recommended applying updates or patches to fix vulnerabilities. This supports the conventional wisdom that many security attacks can be prevented by keeping a system patched. Software developers have come to the same conclusion and recent software systems now commonly incorporate automatic software update components. In this section, we motivate P2 by summarizing the existing state of the art in patch update systems and illustrating their deficiencies compared to P2 in terms of how accurate, continuous and OS-agnostic they are.

Automatic update systems can be provided by the OS, or built specifically into an application. OS-wide update systems, such as the ones built into Windows, Mac OS and Linux periodically check for patches and can be configured to automatically download and apply them. OS-wide automatic update systems may also be built into system administration tool suites, such as IBM's Tivoli system, which also performs various other security and compliance auditing tasks. Application-specific automatic update systems only handle patches for a specific application. Many popular applications, such as the Firefox and Chrome browsers, Acrobat Reader, and many games implement such solutions. Application-specific update systems usually execute when the application is run, when they will check for patches and if necessary, download and install them.

All automatic update systems are host-based: they execute as part of the software system, either as separate software agents or as software mechanisms built into applications. As a result, these systems are tied to a specific OS and are thus not OS-agnostic. In addition, the cloud provider must rely on the cloud user to properly install and configure host-based systems since the cloud provider does not have administrative access to the VMs. Automatic update systems can be accurate or continuous depending on whether they are OS-wide or application-specific. OS-wide updaters periodically compare a database of installed patches on the host with an online repository of patches to determine if patches need to be applied. They are unaware of whether the components they maintain are executed or not, making them inaccurate. In addition, because the check for patches is periodic, they are not continuous. Application-specific updaters that exist as separate applications from the application they maintain have the same properties as OS-wide updaters. However, many application-specific updaters are part of the application they maintain and only run when the application they are patching is run, making them accurate as a result. However, they are only partially continuous. These systems will check for patches when the application first starts up, and then periodically afterwards. Thus, if a patch exists at startup, they will apply it before the application actually runs. However, if a patch becomes available while the application is running, it will not be detected and applied until the next time the update system checks.

An alternative, OS-agnostic approach for detecting the presence of unpatched software is a network vulnerability scanner such as Nessus [21]. These scanners attempt to detect unpatched code by scanning all hosts and ports in a range of IP addresses and trying to identify the version of any network facing services installed on

Solution	Continuous	Accurate	OS-agnostic
OS-wide update	No	No	No
Application update	On startup	Yes	No
Network scanning	No	No	Yes
P2	**Yes**	**Yes**	**Yes**

Table 1. Comparison of P2 and current solutions.

those hosts. The identification of the software version can be done simplistically by examining the banner returned by the service, or in a more sophisticated way by sending requests to the hosts and observing the responses. After such scans, the network vulnerability scanner informs system administrators of the presence of software with known vulnerabilities on machines connected to the network. A network vulnerability scanner does not rely on any software agent being present on the host and thus is attractive in IaaS clouds where there is a diversity of OSs running and the cloud administrator has no administrative access to install and manage any host-based solution.

Unfortunately, a network-based approach has limited coverage. It can only be used to detect unpatched server software for which the version can be determined via network queries. As a result, it will not protect a user that uses an out-of-date browser to visit a malicious website. In addition, network scanning has limited precision and may not be able to definitively determine if a service is vulnerable or not. For example, if the network scanner relies on banners, it will fail if the server is configured not to return a banner. More sophisticated scanners that attempt exploits may fail if the exploit's success depends on random factors. Thus, network scanners have limited accuracy. Network scanning is also non-continuous since the scans only happen periodically. In a cloud environment, this problem is compounded by the fact that software executing on VMs that are only powered on occasionally could remain out-of-date for long periods of time.

In this work, we show how P2 takes advantage of a virtualized infrastructure to implement a patch audit system that has the combined advantages of existing systems. Table 1 summarizes the advantages of P2 over host-based automatic update systems and network-based vulnerability scanning. In a virtualized environment, a ubiquitous layer of software executes below OSs which now interact with virtualized hardware. Because it facilitates management and allows system administrators to retain some control over desktop machines, such a setup has become popular and is offered by VMware View [32] and Citrix XenClient [8], amongst others. This is also increasingly the architecture used in clouds, be they public or private.

2.2 Assumptions and Guarantees

We make several assumptions about the OS that will be installed in the VMs. First, we assume that an efficient, commodity OS is installed. P2 relies on the efficiency assumption because it uses information flow tracking to infer whether a non-binary file is read by an application that can interpret the non-binary file and expose the vulnerability. Specifically, we assume that when a process requests data from a file, data is copied directly from the disk into a buffer cache using Direct Memory Access (DMA), and then copied from there into the address space of the requesting process. If the OS inefficiently makes extra copies of the data before returning it, this will confuse the information flow tracking of P2. Similarly, DMA transfers are much faster than Programmed I/O (PIO) transfers. OSs will generally use DMA if it is available and only fall back to PIO if DMA is not available. We have confirmed that this assumption holds for all flavors of Linux and Windows. Memory copying is expensive and we believe that all OS implementations that consider performance important will try to avoid unnecessary copying of disk data.

Second, P2 assumes that application code is stored on the local virtual disk of the VM. Applications that are executed over a web browser as JavaScript or stored on networked storage are not monitored by P2 because the files containing these applications are not read from the local disk. This assumption is reasonable for a cloud environment where VMs are usually self-contained, and contain all code necessary to execute.

Third, P2 assumes the system it is monitoring is not trying to covertly execute unpatched code. For example, for efficiency, P2 assumes that the flow of information from the disk to a process terminates at the process that reads the file. However, if the adversary wants to avoid detection, she can create a "scrubbing" application that reads an unpatched script, and then passes it to the `stdin` of the interpreter via a pipe. Thus, P2's patch auditing is restricted to VMs that are not under the control of a malicious adversary.

Finally, P2 inspects disk contents to identify blocks that correspond to unpatched files. Currently, if the disk or files on the disk are encrypted or compressed, P2 will not be able to monitor the content of these files. P2 can be extended to support encrypted or compressed files, by adding functionality that would allow it to decrypt or decompress files for analysis.

Given that these assumptions hold, P2 provides two guarantees. First, P2 will identify the execution of all unpatched applications, regardless of whether they are composed of native binary code, non-binary code, or some combination of the above. Non-binary code includes all forms of byte code or scripts, regardless of whether they are executed by an interpreter or a JIT compiler. P2 infers non-binary code execution anytime a script or byte code file is loaded into the memory of a matching interpreter or JIT compiler, i.e. if the Perl interpreter loads a valid Perl script or a JAVA virtual machine loads a JAVA class file. P2 can also detect unpatched resource and configuration files, provided that they match the unpatched, vulnerable version exactly – P2 does not identify vulnerabilities caused by custom configurations. Second, P2 works for any commodity OS. We have validated P2 on two widely used OS environments, Windows XP and Fedora Core Linux.

3. Patch Survey

To be OS-agnostic, P2 uses architectural introspection, which restricts hypervisor monitoring to the interaction between the guest VM and virtual hardware. As a result, P2 must make a distinction between applications implemented in native binary code and applications implemented in an interpreted language. On the one hand, binary code must execute directly on the processor and thus is observable through interactions with the memory management unit (MMU). On the other hand, interpreted code can only execute when it is read and executed by the appropriate interpreter or JIT, which itself is usually a native binary. To understand the importance of these two forms of monitoring, we conducted a survey of historical data to characterize the composition of security-sensitive patches.

We collected all security patches for Fedora Core 10 and 11 from the Fedora Project's admin website [1]. The Fedora Core 10 updates consist of security patches between its release date on November 25, 2008 until October 1, 2009 when we conducted the study. The Fedora 11 updates were for the same period starting from its release date on June 6, 2009. To identify which patches contain security fixes, we used the Fedora Project's update classification scheme, which classifies patches as pending, testing, stable or security. To be classified as security, the patch must contain at least one security critical fix.

[1] `https://admin.fedoraproject.org/updates/`

OS	Changed files	Documents	Binaries	Non-binaries	Top 10 non-binary file extensions
FC 10	73800	61037	4979	7784	pyo, pyc, php, js, tmpl, desktop, elc, info, inc, jar
FC 11	29019	24331	1307	3281	php, pyo, pyc, jar, js, inc, zip, gif, <none>, py

Table 2. Summary of Fedora Core security patch RPMs.

To characterize the types of files patched, we compared each security patch with the previous version of the application that the patch replaced. Fedora uses Red Hat's RPM package format to distribute patches. RPMs contain entire binaries, which will overwrite the binaries of the previous version when the patch is installed. In addition, patch RPMs can also be installed onto a system where no previous version of the application exists. Thus, patch RPMs also contain files that did not change from the previous version. Thus, to identify which files are patched, we compare the files in each patch RPM with the RPM of the previous version. From this, we can characterize the makeup of files that changed in response to a security vulnerability. We note that while a security patch must fix at least one security vulnerability, the patch may also contain fixes for non-security sensitive bugs. Unfortunately, neither the Fedora Project page nor the RPM package format contains sufficient information to remove files that contain non-security sensitive bug fixes from our study, so we assume that any file that changed due to a security patch contains a security fix.

Table 2 summarizes the data obtained from this survey, including the top 10 most common non-binary file types by extension. An extension of <none> denotes a file with no extension. We analyzed 446 out of the 495 security patches in Fedora Core 10 and 130 out of the 134 security patches in Fedora Core 11. Not all patches could be analyzed because we could no longer obtain the matching previous RPM packages in some cases. For each file in an RPM that was changed due to a security patch, we determined if it was a binary file or a non-binary file using the `file` utility, which classifies any ELF file as a shared object or as an executable, and any other file as a non-binary file. To get a fair assessment of the percentage of non-binary files that were modified by security updates, we filtered out `man` pages, document files stored in `/usr/share/doc`, `/usr/share/info`,... as well as `locale` files that contain localization data. We further classified the remaining non-binary files based on their extension. The results illustrate two important findings. First, 64% of the updates across both distributions were to non-binary files. This indicates that P2's ability to monitor the execution of non-binary code in an OS-agnostic way is critical to its success as a patch audit solution. Second, a significant number of the top 10 non-binary files are executable scripts and byte code: `pyo` and `pyc` are compiled python code, `php` and `inc` contain php code, which is often used to implement websites, `jar` and `js` are for JAVA and JavaScript respectively, and `elc` files contain Emacs Lisp code. The top 10 non-binary file types make up 77% and 84% of all non-binary, non-document files in Fedora Core 10 and 11 respectively, forming a significant portion of the patched non-binaries. Thus, many of the non-binary updates are to executables that are run with the aid of an interpreter or a JIT compiler. In particular, much of this code appears to be used to implement web applications. While P2 also has the ability to identify unpatched configuration or application resource files, patches to these types of files are not common in practice.

4. System Architecture

We now describe the design and architecture of P2, which provides accurate, continuous and OS-agnostic detection of unpatched code in a VM. P2 is most naturally implemented in the virtualization infrastructure that manages the cloud VMs or a company's VMs. As a result, its operation is administered by the cloud provider in the public cloud setting, or by the system administrators in charge of the computing infrastructure in an enterprise private cloud setting. The architecture of P2 is illustrated in Figure 1. While P2 leverages Patagonix as a component, all other components in the figure are new contributions of P2. We begin by describing the database of software information that P2 needs and then describe how P2 uses these databases to detect the execution of unpatched binary and non-binary software.

4.1 Software databases

To detect and report the execution of vulnerable, unpatched applications, P2 requires a database of information with which it can identify these applications. Such an *unpatched database* can be derived from the files that make up the application. A list of files that need to be patched can be easily obtained by monitoring the updates applied by automatic update systems, or manually monitoring security vulnerability distribution lists. P2 requires the files that each patch modifies, which can be obtained by comparing the files in the patch with the software version it was patching. The patch survey described in Section 3 was performed using scripts written over a course of 6 days by one of the authors of this paper. Based on this experience, we believe that building and maintaining a database of unpatched software would require only modest effort and time. P2 also requires an *interpreter database*, which lists applications that may load and interpret unpatched non-binary files. We use the term interpreter to denote all interpreters and JIT compilers on the system, as well as any application that can load unpatched configuration files. Each non-binary file in the unpatched database must be associated with at least one binary in the interpreter database, which can load the non-binary file and be subject to the vulnerability.

P2 also requires a *patched database* of files that do not need to be patched. Such a database would be considerably larger than the database of unpatched files since there are many more of such files. Maintaining such a database in a homogeneous enterprise setting where a single entity controls the software environment represents a tractable undertaking. In a more open setting, building a list of patched applications is a more difficult task, but not intractable. For example, anti-virus vendors already maintain large databases that keep track of information about malicious software. Some anti-virus vendors have suggested that it may actually be easier to maintain a white list of software instead of the current industry practice of maintaining a black list of malicious software [24]. The administrator can also leverage existing databases of software that currently exist. For example, NIST maintains a database of hashes for a large number of applications [23] and VersionTracker [31] keeps track of available software updates. Linux vendors also maintain large repositories of software and track updates to this software.

In a public cloud setting, the cost of the work required from the cloud provider can be spread amongst cloud customers. The task of maintaining the database can also be outsourced to third parties. The exact details of how such databases would be managed are

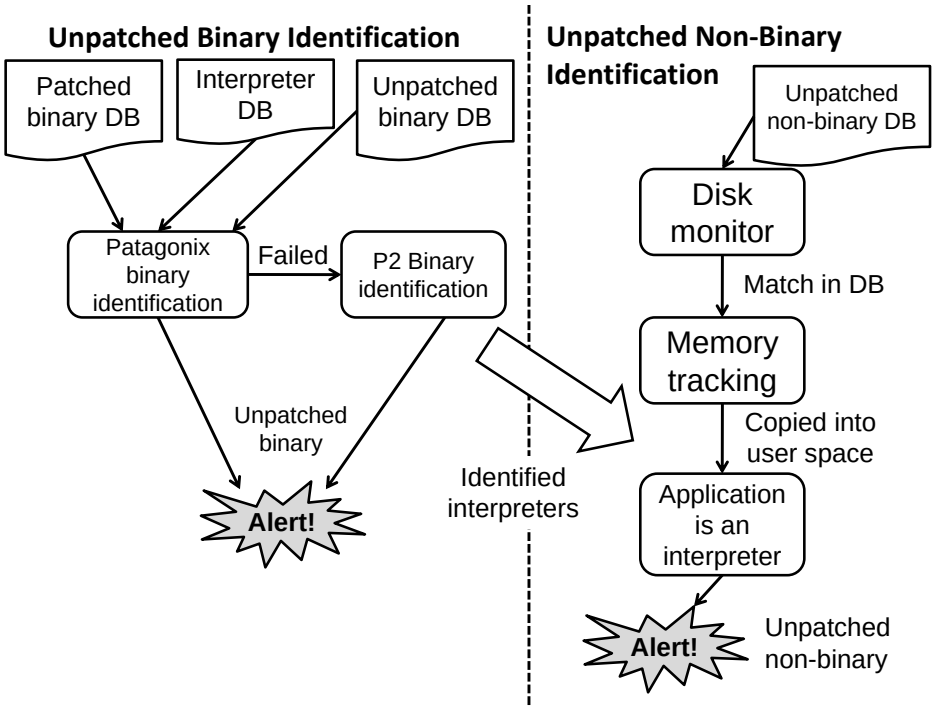

Figure 1. P2 architecture. P2's binary identification is only invoked if Patagonix's binary identification fails. P2's non-binary identification relies on information from the binary identification to find out which processes are running interpreter binaries.

outside the scope of this work. For the rest of this paper, we assume that the cloud provider has access to the files that correspond to the software that need to be monitored.

4.2 Monitoring

Patagonix [17] introduced a mechanism for identifying executing binaries that is accurate, continuous and OS-agnostic. However, Patagonix is not suitable for patch monitoring in a cloud environment for three reasons. First, Patagonix must be invoked before the application to be identified is run. Patagonix operates by tracking the mapping between regions in memory and binary files. For Windows PE binaries, this is problematic because these binaries may be modified at run time to relocate them to a different virtual address. To address this, Patagonix must observe the first instruction executed for any binary, which it uses to identify the binary using an entry point database. Second, Patagonix is only applicable to binaries and cannot identify interpreted or JIT-ed code. P2 extends Patagonix's binary monitoring so that it can identify applications in mid-execution. In addition, P2 adds a completely new monitoring mechanism for non-binary code based on coarse-grain information flow tracking.

The database of unpatched files is divided into two databases: one containing binary files and the other containing non-binary files. The database of unpatched binary files is then combined with a database of interpreters and the database of patched binaries. This combined database is then used for identifying executing binaries.

When P2 detects an executing binary, it first invokes Patagonix to identify the binary. If Patagonix is able to identify the binary it takes actions depending on which database the binary is identified in. If the binary is unpatched, P2 raises an alert and, if running in reporting mode, reports the alert to the cloud administrator. If P2 is running in prevention mode, it will also prevent the unpatched code from running using the technique described in Section 5.3. If P2 matches the executing binary with an entry in the database of interpreters, then it notes the address space the interpreter occupies for use in detecting unpatched non-binary file use. If Patagonix fails to identify the binary, P2 uses the address inference based binary identification algorithm described in Section 5.1 to identify the binary.

P2's mechanism for identifying unpatched non-binary files starts with a virtual disk monitor that monitors disk blocks being read by the guest OS. Each block is compared with the database of unpatched non-binary files. If there is a match, P2 invokes a coarse-grain, page-based memory tracking mechanism to track the use of the data read from the file. Here, P2 leverages information from its binary execution tracking – if it detects that data from the unpatched non-binary file flows into the memory space of a matching interpreter for the file, then it raises an alert and takes action depending on its operating mode.

It is crucial that P2 detects that an unpatched non-binary file is actually loaded into the appropriate interpreter before raising an alert. For example, consider a simpler system where P2 does not track data flow through memory, but instead raises an alert whenever an unpatched file was read off disk. Such a naïve approach will raise alerts even when the file is accessed in a way that will not exercise the vulnerability in the unpatched file. Execution of a file by an interpreter or as a configuration file constitutes *vulnerable accesses*, as it has the potential to exercise the vulnerability in the file. On the other hand, access by applications such as anti-virus software performing a scan, desktop search software that is indexing the system, or backup applications constitute *non-vulnerable accesses* as the accessing applications do not interpret the file data in a way that can exercise the unpatched vulnerability. Without memory information flow tracking, P2 will lose accuracy and report non-vulnerable accesses as alerts. Unpatched files may be present on a system and accessed by such applications for a vari-

Instruction	Encoding
jnz 0x46	7544
push [ebp+0x8]	ff7508
call [0x76381004]	ff15**04103876**
push [0x76382018]	68**18203876**
mov [0x76382018],0x94	c705**18203876**94000000

(a) Example instruction sequence. Preferred address = 0x76380000.

Instruction	Encoding
jnz 0x46	7544
push [ebp+0x8]	ff7508
call [0x76382004]	ff15**04203876**
push [0x76383018]	68**18303876**
mov [0x76383018],0x94	c705**18303876**94000000

(b) Relocated instruction sequence at address = 0x76381000.

Figure 2. PE binary relocation.

ety of reasons. For example, Windows keeps copies of old libraries after an update to allow software rollback should the update fail or result in unforeseen problems.

5. Implementation

5.1 Binary file monitoring

5.1.1 Patagonix background

We use Patagonix [17] as a component to build P2. Patagonix is able to detect executing binary code and identify it using a database of known binaries. Patagonix detects code execution by manipulating page table entries in the hypervisor. Page table entries on x86 processors provide three permission bits for each memory page: a readable bit, a writable bit and a non-executable (NX) bit. Patagonix initially sets the NX bit on all memory pages in the guest VM. As a result, whenever the CPU executes code on a page for the first time, a trap is generated and control is transferred to Patagonix. Patagonix then suspends the guest VM and identifies the page that caused the fault. After identifying the binary, Patagonix enables execution on the page by clearing the NX bit and clearing the writable bit so that the page cannot be modified without its knowledge. If the page is subsequently modified, Patagonix makes the page writable and non-executable again. If the new page is executed, Patagonix is again invoked via a trap and will identify the new contents of the page.

To identify the binary from which a page originates, Patagonix uses *identity oracles*. Identity oracles are binary format specific. Patagonix contains oracles for the PE format [19], which is used by all versions of Windows since Windows NT 3.1 and an oracle for the ELF file format [30] used by many UNIX OSs, including Linux. In the PE case, dynamically loaded libraries (DLLs) may be *relocated* at load time, where the OS adjusts absolute addresses in the binary depending on the virtual address the library is loaded at. Consider the example instruction sequence in Figure 2(a). As we can see, the last 3 instructions have absolute addresses, which appear in the encoded format of the instructions (note that x86 byte order causes the bytes to appear in reverse order). Every DLL has a *preferred address*, which is the default address that all such absolute references assume the binary is loaded at. If it is loaded at any address other than the default address, the loader must adjust all absolute addresses in the binary by the difference. In Figure 2(b), the binary was loaded at address 0x76380000, which is 0x1000 more than the preferred address. As a result, all absolute addresses are also increased by 0x1000.

Since relocation causes the image of the binary in memory to be different from the image on disk, the Patagonix identity oracle must undo the relocations that were applied when the binary was loaded. To do this, Patagonix leverages the insight that all DLLs have a small number of entry points, which must be executed before any other code in the library. The offsets from the start of a memory page of these entry points are invariant under relocation and thus can be used to serve as identifier hints for the DLL. These hints enable Patagonix to narrow down the number of possible DLLs to just a small number, at which point Patagonix can identify the bytes in the page that correspond to relocated absolute addresses, undo the relocations and then check the true identity of the memory page.

Unfortunately, the entry point method only works if Patagonix is active at the time that the entry point is executed. If Patagonix is applied to a VM after resuming from a checkpoint, some applications will be in mid-execution. As a result, Patagonix will have missed the execution of the entry points for these applications and thus fail at identifying the binary. When Patagonix does fail in this way, P2 invokes its identification method based on *address inference* to identify the locations of absolute addresses in the page and undo the relocations on them, thus allowing the page to be identified.

5.1.2 P2 binary identification

To undo relocations in a page of binary code, P2 must infer which bytes in the page may be absolute addresses. To do this, we make the observation that absolute addresses used in a binary must fall within the binary itself. Let S be the size of the binary, and a, the address at which the code execution is detected. Since we know that a points to valid code and a is somewhere within a binary of size S, then all absolute addresses in the code segment of the binary must fall within the range $[a - S, a + S]$.

While we do not know S, we can use the size of the largest binary in the database, s, as an upper-bound for S (i.e. $S \leq s$). P2 then identifies any sequence of 4 bytes that fall within the range $[a - s, a + s]$ as a *candidate address*. When a RISC design is used, instructions are aligned and have fixed sized op-codes. However, there is no alignment restriction for x86 instructions, and instruction op-codes have a variable length. As a result, it is not possible to identify which part of the page consists of operands and which part of the page consists of instructions. Thus, P2 considers all overlapping 4-byte sequences on the page. For example, in the example in Figure 2(a), if s is 4MB, this would give a range of addresses from 0x75c00000 to 0x76400000. This would cause the three absolute addresses, as well as the first 4 bytes (0x75ff4475) to be identified as candidate addresses in the sequence of bytes. While the last three are true addresses, the first 4 bytes is a *false inference* because it just happens to fall within the right range, but is not really an address. To get an idea of the likelihood of such a false inference in general, assume that the byte values in the page are randomly distributed. The probability that a 4-byte candidate address will happen to fall within the range $[a - s, a + s]$ is given by:

$$p = \frac{2 * s}{2^{32}}$$

We remark that this probability increases linearly with s and that this is only an estimate, as code bytes are not randomly distributed. For example, if we use 4 MB for s, we get a 0.002 probability that a candidate address is not actually an absolute address.

To identify the binary from which the code originates, P2 makes a copy of the byte sequence and sets all candidate addresses to zero. A hash of this sequence is then computed and checked against the database of binaries, which contains hashes of sequences where the absolute addresses have also been set to zero. To address false

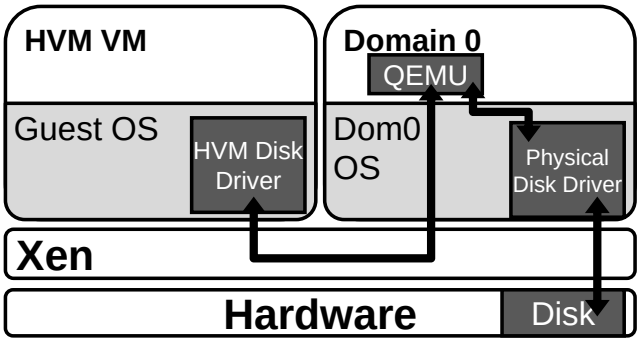

Figure 3. Basic virtual disk architecture for Xen HVM VMs.

inferences, P2 considers all subsets of the candidate addresses (i.e., the power set $\mathcal{P}$ of candidate addresses). This results in 2^k possibilities, where k is the number of candidate addresses. P2 starts by zeroing out all candidate addresses and searching for a match in the binary database. If no match occurs, it then tries all combinations where one candidate address is not zeroed out, then two candidate addresses and so on. The search stops as soon as the sequence is matched with a binary in the database. However, in the worst case, 2^k combinations need to be tested.

Since a page consists of 4096 bytes, k could be as large as 4096, causing 2^k to be extremely large. This would make the algorithm intractable. Rather than consider the entire page, P2 considers only substrings of length l within the page. Only considering substrings also enables P2 to handle pages that contain both code and data. l should be chosen so that strings will have a small number of candidates but be long enough to occur only in at most a few binaries. In our implementation of P2, we used l=64 bytes.

The l-length substrings should be taken at several offsets to provide a spread of offsets throughout the page so that at least one will fall in a code region should a page contain a mix of code and data. At the same time, none of the substrings should straddle a page boundary. In our implementation, we use offsets o in the following order: 0x3; 0x1000 - 0x3 - l; 0x103 + i*0x100, $i \in [0, 14]$. The code database is augmented with an index that contains hashes of the first l-length string at offset o in each binary. The database construction procedure searches for this string by trying different o's in the order given above.

During identification, all combinations of candidate addresses are searched. If a match occurs, then all relocations in the page are undone and a hash over the entire page is computed and checked to verify that the match is indeed correct. Because the hashes on the substrings are taken frequently, our implementation uses the non-cryptographic, fast murmur2 hash [1] for the hash on the substring and a sha256 hash on the whole page to verify the match. The complete process for trying the candidate addresses is summarized in Algorithm 1.

Algorithm 1 The P2 binary identification algorithm.

```
function identify_code(string code, int a, int l, int s)
offset_list = [0x3, 0x1000 - 0x3 -l, 0x103, ..., 0xf03]
for offset ∈ offset_list do
  candidates ← ∅
  for i ∈ [0, l + 3] do
    value = integer value of 4-byte sequence of code from
    offset+i − 3 to offset+i
    if value ∈ [a − s, a + s] then
      candidates ← i
    end if
  end for
  for candidate_set ∈ P(candidates) do
    candidate_string ← code[offset..offset+l[ with sequences
    in candidate_set zeroed out
    hash ← hash of candidate_string
    if hash is in database then
      candidate_pages ← pages in index that match hash
      for candidate_page ∈ candidate_pages do
        relocated_code ← code with relocations undone ac-
        cording to candidate_page relocation information.
        if hash of candidate_page stored in database == hash
        of relocated_code then
          return binary
        end if
      end for
    end if
  end for
end for
```

5.2 Non-binary file monitoring

P2's file execution monitor has two components. The first component is a disk monitor that listens on requests to the virtual disk and detects accesses to unpatched non-binary files. The second component is the memory tracking component, which tracks the flow of data from unpatched files through the guest OS. We give some background on Xen's virtual disk architecture and then describe the two components in turn.

5.2.1 Xen virtual disk background

In a virtualized environment, persistent storage is provided by virtual block devices. Figure 3 illustrates the Xen virtual disk architecture. In Xen Hardware Virtual Machines (HVM), virtual block devices are emulated using a user space QEMU process that runs in domain 0, which is a special VM that has access to the physical devices on the machine. The QEMU process faithfully emulates a physical disk in software. When the guest OS device driver sends DMA requests to the virtual disk, it is intercepted by Xen and forwarded to the QEMU disk emulator in domain 0. QEMU services the requests by accessing the physical disk and responding to the guest OS via Xen. It is possible to improve virtual I/O performance by inserting a custom paravirtualized (PV) disk driver into the guest OS. PV disk drivers are Xen-aware, thus saving domain 0 from having to emulate a physical disk and allowing direct communication between the guest OS driver and the Domain 0 physical disk driver. We could have implemented P2 to support both HVM and PV, but for simplicity, our prototype currently only supports HVM drivers.

The P2 disk monitor assumes that the file system block size is the same as the memory page size (4096 bytes), which is true for all major file systems today. Hard drives currently use a sector size of 512 bytes, which is the smallest addressable chunk of data that can be read from the disk. The memory page size for the x86 architecture is 4096 bytes. Since the sector size is smaller than the memory page size, it is possible that a single memory page could contain data from several different files. However, no file system we know of uses this capability. Instead, file systems match their minimum addressable block size with that of the CPU memory system for simplicity and efficiency. In response to this trend, hard drive manufacturers have agreed on a new standard [7], which specifies that all disks will eventually use a sector size of 4096 bytes instead of 512 bytes.

5.2.2 Disk monitor

The architecture of P2's file monitoring system is summarized in Figure 4, which shows the disk monitor, memory tracking component and binary file monitor from Section 5.1. The purpose of the disk monitor is to detect when an unpatched non-binary file is read and convey the address of the access, and the identity and associated interpreters of the file to the memory tracking component. As a result, the most natural place to implement disk monitoring is in the domain 0 QEMU disk emulator. The unpatched non-binary database contains the identity of each unpatched file and the associated interpreter(s) that can make vulnerable accesses to the file. These entries are indexed by hash values computed over each block-sized chunk of the file. On each disk access, the P2 disk monitor computes a hash of the block being read and searches the index in the unpatched non-binary database. If there is a hit, the disk monitor informs the memory tracking component of the physical address that the disk data will be read to. The disk monitor is able to get the destination address from the DMA request protocol, which includes this information. The disk monitor also conveys the identity of the file as well as the identity of the associated interpreters.

Two cases deserve special attention in the disk monitor: files may be smaller than a single 4096 block and some blocks appear in more than one file. For each unpatched non-binary file, the database stores a hash of a sub-block prefix of each block of the file along with the amount of space occupied by the file in this block in a *block prefix index*. The disk monitor first computes a non-cryptographic hash of this prefix and searches the block prefix index for a match. If there is a hit in the block prefix index, either a hash of the remainder of the block is taken if the file occupies the entire block, or a hash of the portion of the block that is occupied by the file is taken if the file does not occupy the entire block. This serves to verify that the file is indeed a match. If multiple files exist for a given prefix, then each candidate needs to be tested individually until a match is found. The length of the prefix is chosen to minimize the chances of a collision while retaining the ability to handle even very small files. P2 uses a prefix length of 64 bytes. As a result, only files smaller than 64 bytes cannot be handled. In our patch study, there were 32 files out of a combined 17351 files across both Fedora Core 10 and 11 that were less than 64 bytes. Due to their short length and simplicity, we also believe they are unlikely to contain security vulnerabilities. Use of the prefix index means that for the majority of blocks read from the disk which do not contain data from unpatched, non-binary files only a short prefix of the block needs to be hashed before P2 can establish that the block does not contain data that needs to be tracked.

To handle cases where a file block appears in multiple files, the disk monitor conveys the set of non-binary files that could have matched and the monitor component will apply the union of the associated interpreters in its checking. While rare, we have observed that different files do occasionally have exactly matching 4096 byte chunks.

5.2.3 Memory tracking component

The memory tracking component is implemented in the Xen hypervisor. When the memory tracking component is notified of an access to an unpatched non-binary file it clears the "present" bit of the corresponding page table entries in the shadow page tables, thus causing the processor to fault anytime the page containing unpatched file data is accessed. It also marks the page as *tracked* so that when a fault occurs on that page, the Xen hypervisor knows to invoke the P2 memory tracking component. In addition, if the guest OS creates new mappings to a tracked page, the Xen hypervisor will also clear the present bit on the new mappings in the shadow page table.

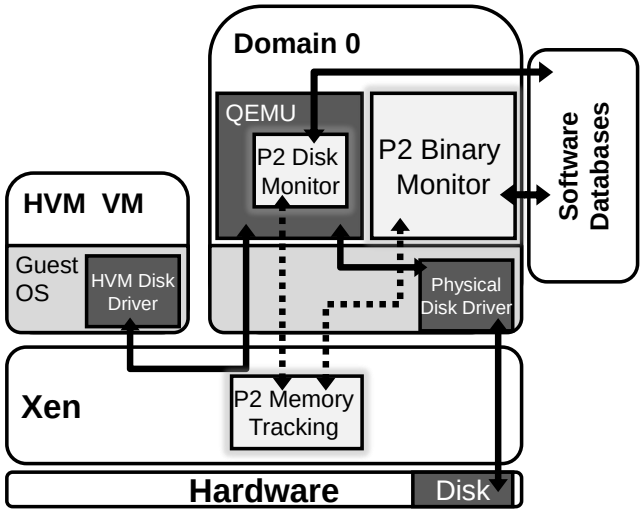

Figure 4. P2 non-binary file monitoring additions to Xen.

If and when a fault to a tracked page occurs, the P2 memory tracking component is invoked by Xen. P2 will inspect the instruction that caused the trap. The instruction can be either in the guest kernel or in a user space process in the guest VM. If the instruction is in the guest kernel, then the file data is being accessed by a process via a system call such as `read`. In this case, P2 inspects the instruction and verifies that the instruction is copying data from the tracked page into the user space process. If the trapping instruction is in a user-space process, then the file data has been mapped into the address space of the user space process. Thus, P2 can infer that the file data has flowed into the address space of a particular process. In either case, the memory tracking component has inferred that an unpatched non-binary file has been read into a user space process.

At this time, the memory tracking component queries the P2 binary file monitor, which tracks executing interpreters. If the current context (read from the `CR3 register`) matches that of a currently executing interpreter that can make a vulnerable access, P2 raises an alert. If running in reporting mode, P2 simply reports the vulnerable access to the cloud administrator. On the other hand, if P2 is running in prevention mode, then it will terminate the interpreter as described in Section 5.3.

If the accessing process is not a vulnerable interpreter, or is allowed to proceed because P2 is operating in reporting mode, then P2 allows the access to the tracked page to proceed by marking the page as present. It maintains this present marking until the content of the `CR3` is changed by the guest OS, indicating a context switch. At this point, the memory tracking component must clear the present bit so that it can detect accesses by other processes. Since multiple traps may take place between two context switches, the tracking component maintains a list of page table entries that need to have their "present" bit cleared upon the next context switch.

In addition to detecting when a tracked page is being accessed, P2 needs to detect when the memory page no longer contains unpatched file data and no longer needs to be tracked. This may occur as a result of two events: the guest VM modified the content of the memory page, or it requested that a virtual device performs a DMA transfer to the memory page. To detect the first scenario, the tracking component clears the "writable" bit on any tracked page. As a result, when the page is written to, either because the page is being reused by the kernel for a different purpose or because

the page content is being modified, a page fault will occur upon access. At this point, P2 clears the tracked flag from the page and stops tracking it.

To detect DMA transfers, we extend the disk monitor to inform the memory tracking component on every DMA transfer, not just the ones to unpatched files. DMA transfers of regular patched files are not tracked, but if they overwrite a tracked page, this will cause P2 to stop tracking the overwritten page.

5.3 Prevention mode

When running in prevention mode, P2 is in a position to prevent any unpatched code from executing. P2 does this with as few side-effects as possible. If the unpatched code is a binary, then P2 replaces the instruction at the faulting address with an illegal instruction. When the guest VM is resumed, the application will execute the illegal instruction causing an OS fault, and the guest OS will terminate the application cleanly.

If the unpatched code is a non-binary file, P2 must terminate the associated interpreter that is making a vulnerable access to the file. If the access occurred from user space because the file is mapped into the interpreter address space, P2 inserts an illegal instruction at the address that caused the trap and restarts the process just like above. However, if the accessed occurred from within the kernel, P2 cannot insert illegal instructions into the kernel as this would terminate the entire VM. Instead, P2 allows the access to complete but sets the entire user space address range to non-executable. When the guest OS returns to user space, a fault will be generated, at which time P2 can inject an illegal instruction and have the guest OS terminate the interpreter.

Setting	Vanilla	P2	P2, non-binary disabled
Linux	436.9s	455.8s (4.3%)	460.9s (5.5%)
Windows	581.3s	605.2s (4.1%)	617.7s (6.3%)

Table 3. Compilation time for Apache (overhead).

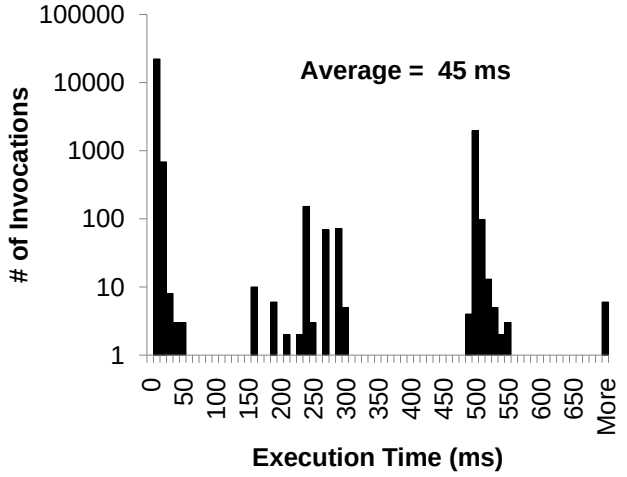

Figure 5. Histogram of P2 binary identification execution times.

6. Evaluation

We evaluated P2's effectiveness at detecting unpatched applications and the performance overhead P2's monitoring introduces. All experiments were conducted on an AMD Athlon 64 X2 Dual Core 3800+ processor running at 2GHz, with 2GB of RAM. We used the Xen 3.3.0 VMM and allocated 512MB of RAM to the monitored VM and 1GB to the `domain 0` VM. We pinned the `domain 0` VM to the first core and the monitored VM to the second core to minimize VM scheduling effects. To demonstrate P2's OS-agnostic quality, unless otherwise stated, all tests were run on both Windows and Linux VMs. The Windows VM runs Windows XP SP2. The Linux VM is a Fedora Core 9 distribution with a 2.6.27.25 Linux kernel. Timing was recorded using an external time server to eliminate clock skew introduced by the hypervisor.

6.1 Effectiveness

We evaluated P2's ability to detect unpatched binary and non-binary code. To evaluate the ability of P2 to detect the execution of binary code, we ran P2 and compared the binaries it reports as running with that returned by the Task Manager. We also suspended and resumed the VMs several times and inspected the reported binaries. In all cases, P2 accurately listed the exact same executing binaries as the respective Linux and Windows tools. This shows that P2 can accurately identify the exact binary that is executing. We also installed both patched and unpatched versions of Apache in both Windows and Linux and executed both. P2 was able to differentiate the execution of the patched version from the unpatched one.

Next, we evaluated the effectiveness of P2 at detecting the execution of non-binary code. To do this, P2 must properly detect access to unpatched files and correctly attribute these accesses to the appropriate interpreter or JIT compiler. We placed a unique ruby script in each directory on the file system (4161 ruby scripts on Windows and 10268 ruby scripts on Linux). To see if P2 correctly attributes the accesses to the correct application, we simultaneously run an anti-virus scanner (ClamAV 0.95.2 on Linux and Symantec 10.1.5 on Windows) and a specially crafted application that would randomly execute one of the ruby scripts every second. In both cases, P2 was able to identify exactly which scripts were accessed by which application and in what order, demonstrating its accuracy. In addition P2, does not miss any access, showing that it is continuous.

To test whether stress on the OS's buffer cache might affect P2's accuracy, we generated load on the system by creating an extension that would cause the Firefox web browser on our test system to randomly crawl websites, thus creating churn in its web cache. We did this at the same time as running the test above. Again, P2 attributed all accesses correctly. In addition, no access to any script was recorded for Firefox or any other application running on the system.

6.2 Performance

P2 is invoked when new code is executed, when data is read from disk into the OS buffer cache and when data in the buffer cache is read by a process. To evaluate the overall performance impact of P2, we measured the time required to compile the Apache web server on both Windows and Linux. In each case, we ran P2 with an unpatched database containing 20,532 entries and comprehensive patched and interpreter databases. We ran P2 twice, once auditing both unpatched binary and non-binary files and again with non-binary file auditing disabled. The results are tabulated in Table 3. In both the Linux and the Windows cases, the overall overhead of P2 was 4% (mean of 4 runs, standard deviation under 1% of the mean). Performance of P2 was actually slightly worse when monitoring only unpatched binaries, but the difference is comparable to measurement error in our system. Thus, we surmise that the majority of the overhead is imposed by the auditing and identification of executing binaries.

P2 binary identification is only invoked when the Patagonix identity oracle fails to identify an executing binary. To evaluate its overhead separately from the overhead of the Patagonix identity oracle, we instrumented the P2 binary identification component to measure both the number of times it is invoked after a VM resumes and the amount of time it takes to identify a binary each time it is invoked. We then perform the Apache compile and suspend the execution of a Windows VM compiling Apache at 1 minute and 2 minute intervals. We found that on average, the P2 algorithm was invoked an average of 46 times with a standard deviation of 2.5 when using a 1 minute interval, and an average of 48 times with a standard deviation of 3 when using a 2 minute interval. This illustrates that few invocations of the P2 algorithm are necessary when resuming a suspended machine. This is because at most one invocation is necessary per binary file that was in the process of being executed when the machine was suspended. The execution time of the P2 binary identification is essentially determined by the number of iterations of the two nested for loops in the algorithm, which are bounded by the number of candidate addresses that are wrongly inferred and the number of offsets that must be attempted. We graph a histogram of execution times in Figure 5. Note that the y-axis is logarithmic. While the execution times vary widely, there are two dominant cases, illustrated by the two peaks centered around 31 ms and 501 ms. The overall average execution time of the P2 binary identification algorithm is 45 ± 143 ms. As a result, combining the average run time of the P2 binary identification algorithm and the average number of times it needs to be invoked, we surmise that P2 adds an average of 3 seconds of overhead every time a VM is resumed.

To more accurately measure the overhead of non-binary file auditing, we evaluate it in isolation. We ran a micro-benchmark that sequentially read 2 GB of data from the disk to measure the disk transfer rate. The vanilla transfer rate was 79 MB/s (mean of 10 runs, standard deviation under 1% of mean) and 78 MB/s with auditing. To assess the impact of the software database size on the file monitoring overhead, we reran the benchmark with a larger database containing 1,252,977 entries. The transfer rate in this scenario dropped slightly to 63 MB/s since more time was required to query the database on each disk access. The modest overhead incurred by file monitoring on transfer rates explains why no noticeable overhead is introduced on a macro-benchmark such as compiling Apache.

7. Related Work

Virtual machine introspection is used to bridge the *semantic gap* [6], which exists between the hypervisor and software running in guest VMs. Many systems have used introspection in the past to bridge this gap. One of the first, Livewire [10], implements intrusion detection from the hypervisor, but required detailed, implementation-specific information on the location of code and data within the guest VM software. Introvirt [16] extends the concept to be able to detect intrusions that have occurred in the past. VMWatcher [14], uses introspection to enable standard anti-virus tools to be run on VMs from within the hypervisor. SBCFI [25] takes a different approach and uses hypervisor privileges to enforce control-flow integrity on the guest OS kernel. Finally, Patagonix [17], on which P2 is based, identifies covertly-executing binaries in guest VMs in a way that is OS-agnostic. P2 is different from these previous systems in two ways. First, all previous approaches are restricted to monitoring binary code, while P2 is capable of monitoring non-binary code as well. Second, all previous approaches focus on detecting intrusions or malicious code. P2 takes a pro-active approach and tries to detect and prevent unpatched software with known vulnerabilities from executing and exposing the system to attack.

Hypervisors have been used for security in general for a number of systems. Hypervisors offer a smaller trusted computing base, and higher privileges, making them an attractive platform for implementing security. Terra [12] used a hypervisor to perform attestation for individual VMs, thus isolating attested high integrity code from unattested low-integrity code. Proxos [29], extended this idea to permit isolated applications to communicate with other VMs in a controlled way. There has also been recent research in using hypervisors to monitor and control what code executes in the kernel. SecVisor [27] is a small hypervisor with fewer than 2000 lines of code, which can verify the integrity of code executing in the kernel.

Dynamic information flow tracking (DIFT) and related taint tracking techniques have been extensively used to improve security. Suh et al. [28] and Taintcheck [22] both use taint tracking to detect corruption of critical pointers with external data. Suh's method requires specialized hardware but has low runtime overhead. Taintcheck uses a software-only method, but suffers from overheads of 30x or greater. Recently, Chang et al. [5] applied compiler analysis to optimize away some of the instrumentation required to perform taint tracking, achieving runtime overheads less than 13% without any specialized hardware. However, their compiler pass requires source code to perform the optimizations and instrument the program. Dalton et al. [9] introduce more complex security policies and a pointer identification algorithm to improve the accuracy of taint tracking. However, their technique requires specialized hardware to be added to the processor to be practical. P2 differs from previous uses of taint tracking for security in two respects. First, P2 uses taint tracking to determine if a non-binary file is accessed in an unsafe way while previous systems used taint tracking to detect attacks. Second, because P2's goals are different, P2 only needs to track taint at a page granularity, which allows it to leverage the processor MMU to monitor accesses to tainted data. This makes P2 far more efficient, allowing it to achieve low overheads without source code changes or special hardware.

Recently, information flow tracking was leveraged by Ho et al. [13] in a virtualization context to track data originating from the network in a virtualized environment. This allows the proposed system to determine if a VM ever attempts to execute data originating from the network. Like P2, page table manipulations are used to track tainted data at the page granularity. However, once tainted pages are accessed, the system switches to byte-level granularity and starts running VMs in an emulator, resulting in significant performance overhead. Because P2 uses coarse taint tracking, code can execute natively on the processor, resulting in modest performance overheads.

Finally, in a non-security context, others have also leveraged hypervisor virtualization of the MMU and disk to monitor systems. monitor the interaction between VMs and storage. Geiger [15] manipulates page table entries to detect evictions from the OS's buffer cache. This information allows optimizing memory allocations to VMs and helps implement a second-level buffer cache maintained by the VMM. Satori [20] hashes all disk content accessed by VMs to identify sharing opportunities between VMs to reduce memory consumption.

8. Conclusion

We have demonstrated that while virtualization has the potential to exacerbate the problem of patch management, it also offers capabilities to improve it. By monitoring only the virtual hardware interface, P2 is able to efficiently, accurately and continuously audit the use of unpatched code in both Linux and Windows systems. While previous systems that implement information flow tracking at the byte or word granularity have prohibitive overheads, P2 takes advantage of the fact that it does not need such fine-granularity. We find that by tracking information flow at a page-level granularity

that matches that of the hardware MMU, P2 is able to infer the execution of non-binary code with very little overhead. This allows P2 to accurately detect unsafe uses of unpatched non-binary files.

We also find that to overcome Patagonix's inability to identify binaries in mid-execution, P2 must adopt a more expensive sampling-based approach to infer addresses. Our evaluation concludes that while P2's binary identification requires several milliseconds on average, it can have wide range of execution times. We are able to mitigate this shortcoming by combining P2 and Patagonix into a single system that achieves good performance equivalent to Patagonix, but does not suffer from the limitation of not being able to identify binaries that have already started execution. With its applicability across common commodity OSs, low overhead and high accuracy, we believe P2 is a practical solution to help IaaS cloud providers audit the patch level of VMs running on their infrastructure.

Acknowledgements

Support for the work in this paper was provided by the NSERC ISSNet Strategic Network and an Ontario MRI Early Researcher Award.

References

[1] A. Appleby. MurmurHash 2.0, 2010. `http://murmurhash.googlepages.com/`.

[2] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack. Timing the application of security patches for optimal uptime. In *Proceedings of the 15th Large Installation Systems Administration Conference (LISA)*, pages 233–242, Nov. 2002.

[3] A. Bellissimo, J. Burgess, and K. Fu. Secure software updates: Disappointments and new challenges. In *Proceedings of the 1st Usenix Workshop on Hot Topics in Security (HOTSEC)*, July 2006.

[4] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, May 2008.

[5] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, Oct. 2008.

[6] P. M. Chen and B. D. Noble. When virtual is better than real. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS 2001)*, pages 133–138, May 2001.

[7] P. Chicoine, M. Hassner, M. Noblitt, G. Silvus, B. Weber, and E. Grochowski. Hard disk drive long data sector white paper. Technical report, The International Disk Drive Equipment and Materials Association (IDEMA), Apr. 2007.

[8] Citrix XenClient, 2010. `http://www.citrix.com/xenclient`.

[9] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 17th USENIX Security Symposium*, pages 395–410, July 2008.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Symposium on Network and Distributed System Security (NDSS)*, pages 191–206, Feb. 2003.

[11] T. Garfinkel and M. Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *The 10th Workshop on Hot Topics in Operating Systems (HotOS 2005)*, May 2005.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206, Oct. 2003.

[13] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection. In *Proceedings of the First European Conference on Systems (EuroSys)*, Apr. 2006.

[14] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 128–138, Oct. 2007.

[15] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–24, Oct. 2006.

[16] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 91–104, Oct. 2005.

[17] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th USENIX Security Symposium*, July 2008.

[18] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Computer meteorology: Monitoring compute clouds. In *The 12th Workshop on Hot Topics in Operating Systems (HotOS 2009)*, May 2009.

[19] Microsoft. Visual Studio, Microsoft Portable Executable and Common Object File Format specification, May 2006. URL `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`. Rev. 8.0.

[20] G. Miłoś, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 Annual Usenix Technical Conference*, July 2009.

[21] Nessus, Tenable Network Security, 2010. `http://www.nessus.org`.

[22] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Symposium on Network and Distributed System Security (NDSS)*, Feb. 2005.

[23] NIST. National software reference library, 2010. `http://www.nsrl.nist.gov/`.

[24] P. Nowak. Internet security moving toward "white list", Sept. 2007. Available at `http://www.cbc.ca/news/background/tech/privacy/white-list.html`.

[25] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 103–115, Oct. 2007.

[26] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 199–212, Nov. 2009.

[27] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[28] G. E. Suh, J.-W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004.

[29] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 279–292, Nov. 2006.

[30] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) specification, May 1995. V1.2.

[31] VersionTracker. VersionTracker, 2010. `http://versiontracker.com/`.

[32] VMware View, 2010. `http://www.vmware.com/products/view`.