# Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems

Shaya Potter    Jason Nieh
*Computer Science Department*
*Columbia University*
{spotter, nieh}@cs.columbia.edu

## Abstract

Desktop computers are often compromised by the interaction of untrusted data and buggy software. To address this problem, we present Apiary, a system that transparently contains application faults while retaining the usage metaphors of a traditional desktop environment. Apiary accomplishes this with three key mechanisms. It isolates applications in containers that integrate in a controlled manner at the display and file system. It introduces ephemeral containers that are quickly instantiated for single application execution, to prevent any exploit that occurs from persisting and to protect user privacy. It introduces the Virtual Layered File System to make instantiating containers fast and space efficient, and to make managing many containers no more complex than a single traditional desktop. We have implemented Apiary on Linux without any application or operating system kernel changes. Our results with real applications, known exploits, and a 24-person user study show that Apiary has modest performance overhead, is effective in limiting the damage from real vulnerabilities, and is as easy for users to use as a traditional desktop.

## 1 Introduction

In today's world of highly connected computers, desktop security and privacy are major issues. Desktop users interact constantly with untrusted data they receive from the Internet by visiting new web sites, downloading files, and emailing strangers. All these activities use information whose safety the user cannot verify. Data can be constructed maliciously to exploit bugs and vulnerabilities in applications, enabling attackers to take control of users' desktops. For example, a major flaw was recently discovered in Adobe Acrobat products that enables an attacker to take control of a desktop when a maliciously constructed PDF file is viewed [2].

The prevalence of untrusted data and buggy software makes application fault containment increasingly important. Many approaches have been proposed to isolate applications from one another using mechanisms such as process containers [24, 28, 32] or virtual machines [39]. Faults are confined so that if an application is compromised, only that application and the data it can access are available to an attacker.

However, existing approaches suffer from an unresolved tension between ease of use and degree of fault containment. Some approaches [20, 26] provide an integrated desktop feel but only provide partial isolation. They maintain traditional usage metaphors, but do not prevent vulnerable applications from compromising the system itself. Other approaches [34, 39] have less of an integrated desktop feel but fully isolate applications into distinct environments, typically by using separate virtual machines. These approaches effectively limit the impact of compromised applications, but are harder to use because users are forced to learn new ways to use these systems as well as having to manage many environments.

To address these problems, we introduce Apiary, a system that provides strong isolation for robust application fault containment while retaining the integrated look, feel, and ease of use of a traditional desktop environment. Apiary accomplishes this using three key mechanisms that combine well-understood technologies like thin clients, operating system containers, and unioning file systems in novel ways.

First, it decomposes a desktop's applications into isolated containers. Each container is an independent appliance that provides all services an application needs to execute. This prevents an application exploit from compromising other applications. To retain traditional desktop semantics, Apiary integrates these containers in a controlled manner at the display and file system.

Second, it introduces the concept of ephemeral containers. Ephemeral containers are execution environments with no access to the user's data that are quickly instantiated from a clean state for only a single application execution. When the application terminates, the container is archived and never used again. Ephemeral containers have three benefits. First, they prevent compromises, because exploits, even if triggered, cannot persist. Second, they protect users from compromised applications. Even when an application has been compromised, a new ephemeral container running that application in parallel will remain uncompromised. Third, they help protect user privacy when using the Internet. Apiary uses ephemeral containers as a fundamental building block of the integrated desktop experience.

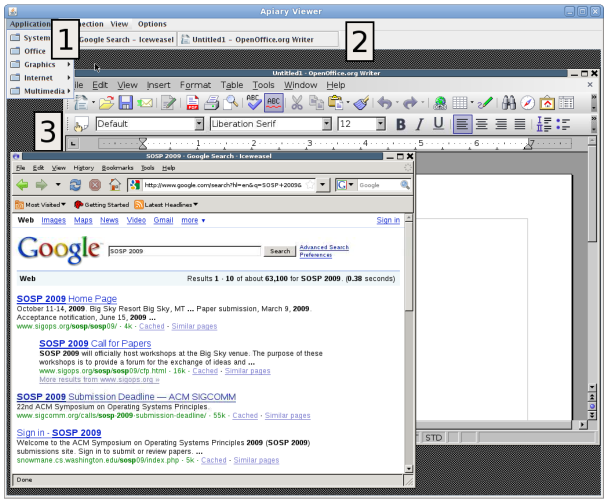
Third, Apiary introduces the Virtual Layered File System (VLFS). Apiary introduces the VLFS to efficiently store and instantiate containers. Each software package or application is stored as a read-only software file system layer. Layers are analogous to software packages in current systems. A VLFS dynamically composes together a set of shared software layers into a single file system view. In Apiary, each container has its own independent VLFS. Since each container's VLFS will share the layers that are common to them, Apiary's storage requirements are the same as a traditional desktop. Similarly, since no data has to be copied to create a new VLFS instance, Apiary is able to quickly instantiate ephemeral containers for a single application execution.

We have implemented an Apiary Linux prototype without any application or operating system kernel changes. We evaluated its effectiveness by conducting various experiments with real applications, vulnerabilities, and users in a user study. Our results show that Apiary can instantiate application containers in under a second, can upgrade a set of containers in under a seconds, has scalable storage requirements, and has only modest file system performance overhead. Our results show that Apiary is effective at containing real exploits. It quickly returns the desktop to a clean uncompromised state in cases where the exploit forces a complete reinstall when it occurs on a traditional desktop system. Finally, our results from a blind user study show that users find Apiary as easy to use as a traditional desktop.

## 2 Apiary Usage Model

Figure 1 shows the Apiary desktop. It looks and feels like a regular desktop. Users launch programs from a menu or from within other programs, switch among launched programs using a taskbar, interact with running programs using the keyboard and mouse, and have a single display with an integrated window system and clipboard functionality that contains all running programs.

Although Apiary works similarly to a regular desktop, it provides fault containment by isolating applications into separate containers. Containers provide all the resources an application needs to run. This includes an isolated execution context, independent display driver and complete file system. As each container's file system is independent, each container has its own isolated home directory to store files created by the user in that container and to isolate them from every other container. For example, if one had a web browsing container and a word processing container, each application would store their contents in the container's version of the user's home


Apiary Viewer
Applications
Connection View Options
System
Office
Graphics
Internet
Multimedia
1
2
3
Google Search - Iceweasel
Untitled1 - OpenOffice.org Writer
File Edit View Insert Format Table Tools Window Help
Default
Liberation Serif
12
STD
SOSP 2009 - Google Search - Iceweasel
File Edit View History Bookmarks Tools Help
Most Visited
Getting Started
Latest Headlines
Web Images Maps News Video Gmail more
Sign in
SOSP 2009
Search
Advanced Search
Preferences
Done

http://www.google.com/search?hl=en&q=SOSP+2009&

Web Results **1** - **10** of about **63,100** for **SOSP 2009**. (**0.38** seconds)

**SOSP 2009** Home Page
October 11-14, **2009**. Big Sky Resort Big Sky, MT **...** Paper submission, March 9, **2009**. Acceptance notification, June 15, **2009** **...**
www.sigops.org/**sosp/sosp**09/ - 4k - Cached - Similar pages

**SOSP 2009** Call for Papers
**SOSP 2009** will officially host workshops at the Big Sky venue. The purpose of these workshops is to provide a forum for the exchange of ideas and **...**
www.sigops.org/**sosp/sosp**09/cfp.html - 16k - Cached - Similar pages
More results from www.sigops.org »

**SOSP 2009** Submission Deadline — ACM SIGCOMM
22nd ACM Symposium on Operating Systems Principles.
www.sigcomm.org/calls/**sosp-2009**-submission-deadline/ - 55k - Cached - Similar pages

Sign in - **SOSP 2009**
Welcome to the ACM Symposium on Operating Systems Principles **2009** (**SOSP 2009**) submissions site. Sign in to submit or review papers. **...**
snowmane.cs.washington.edu/**sosp**09/index.php - 5k - Cached - Similar pages

*Figure 1:* Apiary desktop session: (1) application menu, (2) window list, (3) composited display

directory. This enables containers to enforce isolation without the creation of any isolation rules.

Apiary isolates individual applications, not individual programs. An application in Apiary is a software appliance made up of multiple programs that are used together in a single environment to accomplish a specific task. For instance, a user's web browser and word processor are separate applications and isolated from one another. This software appliance model means that users can install separate isolated applications that contain many or all of the same programs, but used for different purposes. For example, a banking application contains a web browser for accessing a bank's website, while a web surfing application also contains a web browser, but for general web browsing. Both appliances make use of the same web browser program, but are listed as different applications in the application menu. This model can be extended to the point where individual containers are provided for many individual sites, such as Amazon and eBay. While this model differs from a regular desktop, it is similar to what users experience on mobile devices, such as iPhone and Android phones, where they install website specific applications to gain more efficient access to those sites.

Apiary provides two types of containers: ephemeral and persistent. Ephemeral containers are created fresh for each application execution. Persistent containers maintain their state across application executions. Apiary lets users select whether an application should launch within an ephemeral or a persistent container.

Ephemeral containers provide a powerful mechanism for protecting desktop security and user privacy. Users will typically run multiple ephemeral containers, even for the same application, at the same time. They provide important benefits for a wide range of uses.

Ephemeral containers prevent compromises because exploits cannot persist. For example, a malicious PDF document that exploits an ephemeral PDF viewer will have no persistent effect on the system because the exploit is isolated in the container and will disappear when the container finishes executing.

Ephemeral containers protect user privacy when using the Internet. For example, many websites require cookies to function, but also store advertisers' cookies that can compromise a user's privacy. Apiary makes it easy to use multiple ephemeral web browser containers simultaneously, each with separate cookies, and prevents the cookies from persisting.

Ephemeral containers protect users from compromises that may have already occurred on their desktop. If a web browser has been compromised, parallel and future uses of the web browser will allow an attacker to steal sensitive information when the user accesses important websites. Ephemeral containers are guaranteed to launch from a clean slate. For example, by using a separate ephemeral web browser container for accessing a banking site, Apiary ensures that an already exploited web browser installation cannot compromise user privacy.

Ephemeral containers allow applications to launch other applications safely. For example, users often receive viewable email attachments such as PDF documents. To avoid compromising an email container, Apiary creates an ephemeral PDF viewer container for the PDF. Even if it is malicious, it cannot effect the user's desktop, as it only affects the ephemeral container.

Persistent containers are necessary for applications that maintain state across executions to prevent a single application compromise from affecting the entire system. Unlike ephemeral containers, users use only one persistent container per application. Some applications only use one type of container, while others use both. For example, email is typically used in a persistent container to maintain email state across executions. On the other hand, a web browser may be used both in a persistent container, to access a user's trusted websites, and in an ephemeral container, to view untrusted websites.

Apiary's containers work together to provide a security system that differs fundamentally from common security schemes that attempt to lock down applications within a restricted-privilege environment. In Apiary, each application container is an independent entity that is entirely isolated from every other application container on the Apiary desktop. One does not have to apply any security analysis or complex isolation rules to determine which files a specific application should be able to access. Also, in most other schemes, an application, once exploited, will continue to be exploited, even if the exploited application is restricted from accessing other applications' data. Apiary's ephemeral containers, however, prevent an exploit from persisting between application execution instances.

Apiary provides every desktop with two ways to share files between containers. First, containers can use standard file system share concepts to create directories that can be seen by multiple containers. This has the benefit of any data stored in the shared directory being automatically available to the other containers that have access to the share. Second, Apiary supplies every desktop with a special persistent container with a file explorer. The explorer has access to all of the user's containers and can manage all of the user's files, including copying them between containers. This is useful if the user wants to preserve a file from an ephemeral container, or move a file from one persistent container to another, as, for instance, when emailing a set of files. The file explorer container cannot be used in an ephemeral manner. Its functionality cannot be invoked by any other application on the system and no other application is allowed to execute within it. This prevents an exploited container from using the file explorer container to corrupt others.

It should be noted that both of these mechanism break, to some degree, the container's isolation. File system shares can be used by an exploited container as a vector to infect other containers by tricking a user into moving a malicious file between containers. However, this is a tension that will always exist in security systems that are meant to be usable to a diverse crowd of users. To mitigate this, Apiary lets documents stored in a persistent manner be viewable, by default, in an ephemeral container. For example, PDF files can be stored persistently, but always viewed in ephemeral containers. However, to persistently edit a PDF file, it would still have to be opened within a persistent container such that a malicious PDF would have a persistent effect.

## 3 Architecture

To support its container model, Apiary must have four capabilities. First, Apiary must be able to run applications within secure containers to provide application isolation. Second, Apiary must provide a single integrated display view of contains all running applications. Third, Apiary must be able to instantiate individual containers quickly and efficiently. Third, for a cohesive desktop experience, Apiary must allow applications in different containers to interact in a controlled manner.

Apiary does this by using a virtualization architecture that consists of three main components: an operating system container that provides a virtual execution environment, a virtual display system that provides a virtual display server and viewer, and the VLFS. Apiary also provides a desktop daemon that runs on the host to instantiate containers, manage their lifetimes, and ensure that they are correctly integrated.

## 3.1 Process Container

Apiary's containers enable applications to be isolated from one another. Individual applications can run in parallel within separate containers, and have no conception that there are other applications running. This enforces fault containment, as an exploited process only has access to files available within its own container.

Apiary's containers leverage commodity operating system features such as Solaris's zones [32], FreeBSD's jails [21], and Linux's containers [24] to create isolated and independent execution environments. Each container has its own private kernel namespace, file system, and display server, providing total isolation at the process, file system, and display levels. Programs within separate containers can only interact using normal network communication mechanisms. Each container is provided with an application control daemon that enables the virtual display viewer to query the container for its contents and interact with it.

## 3.2 Display

Apiary's virtual display system is crucial to complete process isolation and a cohesive desktop experience. In Apiary, each container has a virtual display similar to existing systems [4,11,37,38]. This virtual display operates by decoupling the display state from the underlying hardware and enabling the display output to be redirected anywhere. This is necessary, since if containers were to share a single display directly, malicious applications could leverage built-in mechanisms in commodity display architectures [13,27] to insert events and messages into other applications that share the display, enabling the malicious application to remotely control the others, effectively exploiting them as well. Many existing commodity security systems do not isolate applications at the display level, providing an easy avenue for attackers to further exploit applications on the desktop.

However, if each container's display is independent, they will not provide a single cohesive display. Apiary provides a cohesive display in two ways. First, it integrates the displays views into a single view. While a regular remote framework provides all the information needed to display each desktop, it assumes that there is no other display in use, and therefore expects to be able to draw the entire display area. In Apiary, where multiple containers are in use, this assumption does not hold. Therefore, to enable multiple displays to be integrated into a single view, the Apiary viewer does Porter-Duff compositing [30] of the displays using the *over* compositing operation.

Second, Apiary's display viewer provides the normal desktop metaphors that users expect, including a single menu structure for launching applications and an integrated task switcher that allows the user to switch among all running applications. Apiary leverages the application control daemon running within each container to enumerate all the available applications within the container, much like a regular menu application does in a traditional desktop. Instead of providing the menu directly in the screen, however, it transmits the collected data back to the viewer, which then integrates this information into its own menu, associating the menu entry with the container it came from. When a user selects a program from the viewer's menu, the viewer instructs the correct daemon to execute it within its container.

Similarly, to manage running applications effectively, Apiary provides a single taskbar with which the user can switch between all applications running within the integrated desktop. Apiary leverages the system's ability to enumerate windows and switch applications [15] to have the daemon enumerate all the windows provided by its container and transmit this information to the viewer. The viewer then integrates this information into a single taskbar with buttons corresponding to application windows. When the user switches windows using the taskbar, the viewer communicates with the daemon and instructs it to bring the correct window to the foreground.

## 3.3 Virtual Layered File System

Apiary requires containers to have file systems that are efficient in storage space, instantiating time, and management costs. A container's file system has to be efficient in storage space to enable regular desktops to support the large number of application containers that will be used within the Apiary desktop. A container's file system has to be efficient to instantiate to provide fast interactive response time, especially for launching ephemeral containers. Finally, a container's file system has to be efficient to manage as Apiary increases the number of file systems that are in use.

There are many existing file system approaches that could be used for Apiary, but they all suffer drawbacks. Package management [12, 35] is useful for managing a file system, however, it does not help provision a file system quickly nor is it space efficient if each independent container's file system has its own copy of the package. This also impacts management as each file system would have to be updated independently. File systems that support a branching semantic [7, 29] can be used to instantiate an ephemeral container quickly from a template file system. However, each template is independent and is therefore inefficient in space and in its ability to be maintained. Finally, even a single template file system with all the programs desired for every container does not help since it reduces isolation between programs.

Apiary introduces the concept of a VLFS to meet these requirements. The VLFS extends the package management concept to enable file systems to be created by composing shareable layers together into a single file system namespace view. VLFSs are built by combining a set of shared software layers together in a read-only manner with a per container private read-write layer. The VLFS's software layers are analogous to packages in a traditional system, and just like a file system will have hundreds of packages installed into it, a VLFS can be composed of hundreds of layers as well. Similar to a regular file system, where package management tools are used to update and install packages and their dependencies into the system, in Apiary, the same type of tools are used to create VLFSs and keep them up to date.

Unlike multiple regular file systems that will each need their own copy of a file, multiple VLFSs providing multiple applications are as efficient as a single regular file system as all files that are common among them will be stored once in the set of shared layers. Therefore, Apiary is able to store the file systems needed by its containers in an efficient manner. This also enables Apiary to manage its containers easily, as all one has to do is replace the single layer that contains the files that have to be updated to update each VLFS that uses it. The VLFS also enables Apiary to efficiently instantiate each container's file system. As no data has to be copied into place and each of the software layers is shared in a read-only manner, instantiating a file system is nearly instantaneous, and occurs transparently to the end user.

Layers are the primary building block of a VLFS. Layers are composed of three elements: the metadata files that describe the layers, configuration scripts that enable the layer to be added and removed from the VLFS correctly, and the primary component, its file system namespace. The layer's file system namespace is a self-contained set of files providing a specific set of functionality. The files are the individual items in the layer that are composed into a larger VLFS. There are no restrictions on the type of files. They can be regular files, symbolic links, hard links or device nodes. The layer's file system namespace can be viewed as a directory stored on the shared file system that contains the same file and directory structure that would be created if the individual items were installed into a traditional file system. On a traditional UNIX system, the directory structure would typically contain directories such as `/usr`, `/bin` and `/etc`. Symbolic links work as expected between layers since they work on path names, but a limitation is that hard links cannot exist between layers.

To support the VLFS, Apiary must solve a number of file system related problems. First, to enable quick instantiation, the VLFS must be able to quickly compose numerous distinct file system layers into a single static view. Second, as users expect to be able to interact with the VLFS as a normal file system, such as by creating and modifying files, Apiary has to enable an instantiated VLFS to be fully modifiable, while enforcing the read-only semantics for the software layers. Finally, Apiary has to support the ability to dynamically add and remove layers without taking the file system off-line. This is equivalent to installing, removing or upgrading a software package while a monolithic file system is online.

To solve these problems, Apiary leverages and expands upon unioning file systems [41]. Unioning file systems enable Apiary to solve the first problem as they allow the system to join multiple distinct file system namespaces into a single namespace view. These directories are unioned by layering directories on top of one another, joining all the files provide by all the layers into a single file system namespace view. As unioning requires no copying, it occurs quickly, enabling Apiary to be efficient in terms of provisioning.

To solve the second problem, union semantics are extended [41] to enable the assignment of properties to the layers, defining some layers to be read only, while others are read-write. This results in a model that borrows from copy-on-write (COW) file systems, where modifying a file on a lower read-only layer will cause it to be copied to the topmost writable layer in a COW fashion. The VLFS leverages this property to enable multiple VLFSs to share a set of software layers in a read-only manner, while providing each instantiated VLFS with its own read-write private layer to store file system modifications. This enables Apiary to be efficient in terms of storage.

This layering model also provides semantics that directory entries located at higher layers in the stack obscure the equivalent directory entries at lower levels. To provide a consistent semantic, if a file is deleted, a white-out mark is also created to ensure that files existing on a lower layer are not revealed. The white-out mechanism enables obscuring files on the read only lower layers, by just creating the white-out file on the topmost read-write private layer.

However, this creates a problem where a file deleted from a read-only share will never be able to be recreated. In a traditional file system, a deleted system file can be recovered by simply reinstalling the package that provided that file. In a VLFS, if the white-outs are stored in the private layer, they will persist, and even if the layer containing the file is replaced, the file will remain obscure. The VLFS solves this problem by associating individual private writable layers with each of its shared read-only layer for the storage of white-outs. When a file in a shared read-only layer is deleted, instead of writing a white-out file to the top-most layer, the white-out is stored in the shared layer's associated white-out layer. When a layer is replaced, its associated white-out layer will be replaced with an empty white-out layer as well, enabling any obscured file to be revealed.

Similarly, the VLFS has to handle the case where a file belonging to a shared read-only layer was modified and therefore copied up to the VLFS's private read-write layer. Apiary provides a *revert* command that enables the owner of a file that has been modified to revert the file's state to its original pristine state. While a regular VLFS *unlink* operation would remove the modified file from the private layer and create a white-out mark to obscure the original file, *revert* only removes the copy in the private layer thereby revealing the original copy below it.

Finally, VLFSs also have to support being managed while they are in use. In a traditional file system, an administrator can remove a package containing files in use, as deleting a file does not remove its contents from the file system until the file is no longer in use. However, if a layer is removed from a union, the data is effectively removed as well as unions only operate on system namespaces and not on the data the underlying files contain. If an administrator wanted to modify the VLFS by removing a layer due to deletion or upgrade maintenance, one would be forced to perform the maintenance off-line due to not being able to remove layers that are in use.

The VLFS solves this problem by emulating what the `unlink` operation does on a single file and applies it to layer removal. `unlink` operates in two steps. It first deletes the file name from the file system's namespace, while only freeing up the space taken up by the file's contents when it's no longer in use. Traditional package management systems rely on these semantics to enable them to upgrade packages, even if files are in use, by unlinking and then recreating them instead of directly overwriting the files. Apiary applies these semantics to layers. When a layer is removed from a VLFS, Apiary marks the layer as `unlinked`, removing it from the file system namespace. While this layer is no longer part of the file system namespace and therefore cannot be used by any operations that work on the file system namespace, such as `open`, it remains part of the VLFS enabling data operations, such as `read` and `write`, to continue to work correctly for files that were previously opened.

## 3.4 Inter-Application Integration

Apiary's isolated containers provide effective fault containment. However, isolated containers can hinder effective use of the desktop. For instance, if one's web browser is totally isolated from the PDF viewer, how does one view a downloaded PDF file? If the PDF viewer is included within the web browser container the isolation that should exist between web browser and an application viewing untrusted content is violated. Users could copy the file from the web browser container to the PDF viewer container, but this is not the integrated feel that users expect.

Apiary solves this problem by enabling applications in one container to execute specific applications in ephemeral containers. Every container is preconfigured with a list of programs that it enables other applications to use in an ephemeral manner. Apiary refers to these as *global* programs. For instance, a Firefox container can specify `/usr/bin/firefox` and a Xpdf container can specify `/usr/bin/xpdf` as global programs. Program paths marked global exist in all containers. Apiary accomplishes this by populating a single global layer, shared by all the container's VLFSs, with a wrapper program for each global program. This wrapper program is used to instantiate a new ephemeral container and execute the requested process within it.

When executed, the wrapper program determines how it was executed and what options were passed to it. It connects via network mechanisms to the Apiary desktop daemon on the same host and passes this information to it. The daemon maintains a mapping of global programs to containers and determines which container is being requested to be instantiated ephemerally. This ensures that only the specified global programs' containers will be instantiated, preventing an attacker from instantiating and executing arbitrary programs. Apiary is then able to instantiate the correct fresh ephemeral container, along with all the required desktop services, including a display server. The display server is then automatically connected to the viewer. Finally, the daemon executes the program as it was initially called in the new container.

To ensure that ephemeral containers are discarded when no longer needed, Apiary's monitors the process executed within the container. When it terminates, Apiary terminates the container. Similarly, as the Apiary viewer knows which containers are providing windows to it, if it determines that no more windows are being provided by the container, it instructs the desktop daemon to terminate the container. This ensures that an exploited process does not continue running in the background.

However, running a new program in a fresh container is not enough to integrate applications correctly. When Firefox downloads a PDF and executes a PDF viewer, it must enable the viewer to view the file. This will fail because Firefox and an ephemeral PDF viewer containers do not share the same file system. To support this functionality, Apiary enables small private read-only file shares between a parent container and the child ephemeral container it instantiates. Because well-behaved applications such as Firefox, Thunderbird, and OpenOffice only use the system's temporary directory to pass files among them, Apiary restricts this automatic file sharing ability to files located under `/tmp`. To ensure that there are no namespace conflicts between containers, Apiary provides containers with their own private directory under `/tmp` to use for temporary files, and they are preconfigured to use that directory as their temp directory.

But providing a fully shared temporary file directory allows an exploited container to access private files that are placed there when passed to an ephemeral container. For instance, if a user downloads a malicious PDF and a bank statement in close succession, they will both exist in the temp directory at the same time. To prevent this, Apiary provides a special file system that enhances the read-only shares with an access control list (ACL) that determines which containers can access which files. By default, these directories will appear empty to the rest of the containers, as they do not have access to any of the files. This prevents an exploited container from accessing data not explicitly given to it. A file will only be visible within the directories if the Apiary desktop daemon instructs the file system to reveal that file by adding the container to the file's ACL. This occurs when a global program's wrapper is executed and the daemon determines that a file was passed to it as an option. The daemon then adds the ephemeral container to the file's ACL. Because the directory structure is consistent between containers, simply executing the requested program in the new ephemeral container with the same options is sufficient.

Situations can conceivably exist where the ephemeral application would need to access multiple files located within the temporary directory, such as a web page with images where the entire web page is saved. In these cases, Apiary's sharing will fail to permit access to all the files. However, in practice, these situations are uncommon and Apiary's scheme works well. In situations where this can occur, one can construct the application containers to contain all the programs needed. For instance, in a web development container, one will provide a web browser to preview one's content, instead of instantiating an external ephemeral container, thereby preventing this problem from occurring.

Apiary enables the file explorer container discussed in Section 2 in a similar way. The file explorer container is similar to Apiary's other containers. It is fully isolated from the rest of the containers and users interact with it via the regular display viewer. However, the other containers are not fully isolated from it. This is necessary as users can store their files in multiple locations in each container, most notably, the `/tmp` directory and the user's home directory. Apiary's file explorer provides read-write access to each of these areas as file shares within the file explorer's file system namespace. Apiary prevents any executable located within these file systems from executing with the file explorer container to prevent malicious programs from exploiting it. Users are able to use normal copy/paste semantics to move files among containers. While this is more involved than a normal desktop with only a single namespace, users generally do not have to move files among containers.

The primary situation in which users might desire to move files between containers is when interacting with an ephemeral container, as a user might want to preserve a file from there. For instance, users can run web browsers in an ephemeral containers to maintain privacy, but also download files they want to keep. While the ephemeral container is active, a user can just use the file explorer to view all active containers. To avoid situations where users only remembers after terminating the ephemeral container that it had files they wanted to keep, Apiary archives all newly created or modified non hidden files that are accessible to the file explorer when the ephemeral container terminates. This allows a user to gain access to them even after the ephemeral container has terminated. Apiary automatically trims this archive if no visible data was stored within the ephemeral container, such as in the case of an ephemeral web browser that the user only used to view a web page, not download and save a specific file. Similarly, Apiary provides users the ability to trim the archive to remove ephemeral container archives that do not contain data they need.

Apiary also turns the desktop viewer into an inter-process communication (IPC) proxy that can enable IPC state to be shared among containers in a controlled and secure manner. This means that only explicitly allowed IPC state is shared. For example, one of the most basic ways desktop applications share state is via the shared desktop clipboard. To handle the clipboard, each container's desktop daemon monitors the clipboard for changes. Whenever a change is made to one container's clipboard, this update is sent to the Apiary viewer, and then propagated to all the other containers. The Apiary viewer also keeps a copy of the clipboard so that any future container can be initialized with the current clipboard state. This enables users to continue to use the clipboard with applications in different containers in a manner that is consistent with a traditional desktop. However, by allowing the clipboard of an ephemeral container to read from the shared clipboard, Apiary does allow information to be leaked. This can be handled by only allowing ephemeral containers to write to the shared clipboard, if the decreased functionality is acceptable.

## 4 Experimental Results

We have implemented a remote desktop Apiary prototype system for the Linux desktop without any application, library, window system, or base kernel changes. The prototype consists of a virtual display driver for the X window system based on MetaVNC [37], a set of user space utilities that enable container integration, and a loadable kernel module for the Linux 2.6 kernel that provides the ability to create and mount VLFSs. Apiary uses Zap [28], a predecessor to Linux containers [24], to provide the isolated containers.

For our prototype, we created 211 software layers by converting the set of Debian packages needed by the set of applications we tested into individual layers. Each Debian package can be viewed as providing three sets of items, a set of files that is extracted into a file system, a set of metadata that determines the dependency relationship among packages, and configuration scripts that are executed on installation and removal to ensure the packages are installed correctly. For the VLFS, we first extract the set of files into a directory that will be used for composition. Second, we extract the metadata that determines dependency relationships between the packages and associate it with the newly created layers. Finally, we associate the configuration scripts from each package with the layers which are used each time the layer is added or removed from an application appliance. Using these layers, we are able to create per application appliances for each individual application by simply selecting which high-level applications we want within the appliance, such as Firefox, with the dependencies between the layers ensuring that all the required layers are included. Using these appliances, we are able to instantly provision persistent and ephemeral containers for the applications as needed.

Using this prototype, we used real exploits to evaluate Apiary's ability to contain and recover from attacks. We conducted a user study to evaluate Apiary's ease of use compared to a traditional desktop. We also measured Apiary's performance with real applications in terms of runtime overhead, startup time, and storage efficiency. For our experiments, we compared a plain Linux desktop with common applications installed to an Apiary desktop that has applications available for use in persistent and ephemeral containers. The applications we used are the Pidgin 2.4.3 instant messenger, the Firefox 3.0.3 web browser, the Thunderbird 2.0 email client, the OpenOffice.org 2.4.1 office suite, the Xpdf 3.02 PDF viewing program, and the MPlayer 1.0-rc2 media player. Experiments were conducted on an IBM HS20 eServer blade with dual 3.06 GHz Intel Xeon CPUs and 2.5 GB RAM. All desktop application execution occurred on the blade. Participants in the usage study connected to the blade via a Thinkpad T42p laptop, with a 1.8 GHz Intel Pentium-M CPU and 2GB of RAM running the MetaVNC viewer.

## 4.1 Handling Exploits

We tested two scenarios that illustrate Apiary's ability to contain and recover from a desktop application exploit, as well as explore how different decisions can affect the security of Apiary's containers.

### 4.1.1 Malicious Files

Many desktop applications have been shown to be vulnerable to maliciously created files that enable an attacker to subvert the target machine and destroy data. These attacks are prevalent on the Internet, as many users will download and view whatever files are sent to them. To demonstrate this problem, we use 2 malicious files [14, 16] that exploit Xpdf 1.01 and mpg123 pre0.59s. We installed the older Xpdf version in the Xpdf container and mpg123 in the MPlayer container. The mpg123 exploit works by creating an invalid mp3 file that triggers a buffer overflow in old versions of mpg123, enabling the exploit to execute any program it desires. The Xpdf exploit works by exploiting a behavior of how Xpdf launched helper programs, that is, by passing a string to `sh -c`. By including a back-tick (\` \`) string within a URL embedded in the PDF file, an attacker could get Xpdf to launch unknown programs. Both of these exploits are able to leverage sudo to perform privileged tasks, in this case, deleting the entire file system. Sudo is exploited because popular distributions require users to use it to gain root privileges and have it configured to run any applications. Additionally, sudo, by default, caches the user's credentials to avoid needing to authenticate the user each time it needs to perform a privileged action. However, this enables local exploits to leverage the cached credentials to gain root privileges.

In the plain Linux system, recovering from these exploits required us to spend a significant amount of time reinstalling the system from scratch, as we had to install many individual programs, not just the one that was exploited. Additionally, we had to recover a user's 23 GB home directory from backup. Reinstalling a basic Debian installation took 19 minutes. However, reinstalling the complete desktop environment took a total of 50 minutes. Recovering the user's home directory, which included multimedia files, research papers, email, and many other assorted files, took an additional 88 minutes when transferred over a Gbps LAN.

Apiary protected the desktop and enabled easier recovery. It protected the desktop by letting the malicious files be viewed within ephemeral containers. Even though the exploits proceeded as expected and deleted the container's entire file system, the damage caused is invisible to the user, because that ephemeral container was never used again. Even when we permitted the exploits to execute within persistent containers, Apiary enabled significantly easier recovery from the exploits. As shown in Table 2, Apiary can provision a file system in just a few milliseconds. This is nearly 6 orders of magnitude faster than the traditional method of recovering a system by reinstallation. Furthermore, Apiary's persistent containers divide up home directory content between them. For instance, a web browser container's home directory will contain the web browser's configuration, browser cache and downloaded files, while a word processing container will contain the documents one has created or edited. This eliminates the need to recover all of a user's data if only one application is exploited.

This also shows how persistent containers can be constructed in a more secure manner to prevent exploits from harming the user. As a large amount of the above user's data, such as media files, is only accessed in a read-only manner, the data can be stored on file system shares. This enables the user to allow the different containers to have different levels of access to the share. The file explorer container can access it in a read-write manner, enabling a user to manage the contents of the file system share, while the actual applications that view these files can be restricted to accessing them in a read-only manner, protecting the files from being damaged by exploits.

### 4.1.2 Malicious Plugins

Applications are also exploited via malware that users are tricked into downloading and installing. This can be an independent program or a plugin that integrates with an already-installed application. For example, malware have tried to convince users to download a "codec" they need to view a video. Recently, a malicious Firefox extension was discovered [6] that leverages Firefox's extension and plugin mechanism to extract a user's banking credentials from the browser when the user visits a bank's website. These attacks are common because users are badly conditioned to always allow a browser to install what it claims is needed. When installed into a traditional environment, this malicious extension persists until the user, or the user's anti-virus software, discovers and removes it. As it does not affect regular use of the browser, there is little to alert users that they have been attacked. As this exploit is not publicly available, we simulated its presence with the non-malicious Greasemonkey Firefox extension [18]. Like with the malicious file, ephemeral containers prevented the extension from persisting.

However, this exploit poses a significant risk to persistent web browser containers. While one might expect Firefox extensions to be uninstallable through Firefox's extension manager, this is only true of extensions that are installed through it. If an extension is installed directly into the file system, it can only be disabled this way, but not uninstalled. This applies equally to Apiary and traditional machines. While Apiary users can quickly recreate the entire persistent Firefox container, that requires knowing that the installation was exploited. Apiary handles this situation more elegantly by allowing the user to use Firefox in multiple web browsing containers. In this case, we created a general-purpose web browsing container for regular use, as well as a financial web browsing container for the bank website only. Apiary configured the financial web browser container to refuse to install any addons within it's browser, keeping it isolated and secure even when the general-purpose web browsing container was compromised.

Apiary enables the creation of multiple independent application containers, each containing the same application, but performing different tasks, such as visiting a bank website. Because the great majority of the VLFS's layers are shared, the user incurs very little cost for these multiple independent containers. This approach can be extended to other related but independent tasks, for instance, using a media player to listen to one's personal collection of music, as opposed to listening to Internet radio from an untrusted source.

This scenario also reveals a problem with how plugins and other extensions are currently handled. When the browser provides its own package management interface independent of the system's built-in package manager, this impacts Apiary, because certain application extensions might be needed in an ephemeral container, but if they are not known to the package manager, they cannot be easily included. However, many plugins and browser extensions are globally installable and manageable via the package manager itself in systems like Debian. In these systems, this yields the benefit that when multiple users wish to use an extension, it only has to be installed once. In Apiary, it additionally provides the benefit that it can become part of the application container's definition, making it available to the ephemeral container without requiring it to be manually installed by the user on each ephemeral execution.

## 4.2 Usage Study

We performed a 24-person usage study that evaluated the ability of users to use Apiary's containerized application model based on our prototype environment, focusing on their ability to execute applications from within other programs. Participants were mostly recruited from within our local university, including faculty, staff and students. All of the users were computer-literate, though a significant number were not power users and included business and humanities students.

For our study, we created three distinct environments. The first was a plain Linux environment running the Xfce4 desktop. It provided a normal desktop Linux experience with a background of icons for files and programs and a full-fledged panel application with a menu, task switcher, clock and other assorted applets. The second was a full Apiary environment. It provided a much sparser experience, as the current Apiary prototype only provides a set of applications and not a full desktop envi-

| Test | Description |
|---|---|
| Untar | Extract Linux 2.6.19 kernel source archive |
| Gzip | Compress a 250 MB Linux kernel source archive |
| Octave | Octave 3.0.1 numerical benchmark [19] |
| Kernel | Build the 2.6.19 kernel |

*Table 1:* Application Benchmarks

ronment. The third was a neutered Apiary environment that differs from the full environment in not launching any child applications within ephemeral containers.

The three environments enable us to compare the participants' experience along two axes. First, we can compare the plain Linux environment, where each application is only installed once and always run from the same environment, to the neutered Apiary environment, where each application is also only installed once and run from the same environment. This allows us to measure the cost of using the Apiary viewer, with its built-in taskbar and application menu, against plain Linux, where the taskbar and application menu are regular applications within the environment. Second, the full and neutered Apiary desktops enable us to isolate the actual and perceived cost to the participants of instantiating ephemeral containers for application execution. We presented the environments to the participants in random order and iterated through all 6 permutations equally.

We timed the participants as they performed a number of specific multi-step tasks in each environment that were designed to measure the overhead of using multiple interacting applications. In summary, the tasks were: (1) download and view a PDF file with Firefox and Xpdf and follow a link embedded in the PDF back to the web; (2) read an email in Thunderbird that contains an attachment that is to be edited in OpenOffice and returned to the sender; (3) create a document in OpenOffice that contains text copied and pasted from the web and sent by e-mail as a PDF file; (4) create a "Hello World" web page in OpenOffice and preview it in Firefox; and (5) launch a link received in the Pidgin IM client in Firefox.

As Figure 2 shows, the average time to complete each task only differed by a few seconds for all tasks in all environments. Figure 2 shows that even in tasks where users were creating multiple new ephemeral containers, that overhead imposed in creating these containers is minimal and generally unnoticeable to the user. Therefore, users were able to complete the tasks using Apiary with the same efficiency as on a regular system.

The participants also rated their perceived ease of use of each environment on a scale of 1 to 5. The average rating, of both the plain Linux environment and Apiary, was a 3.9 with a standard deviation of 0.9 and 1.1 respectively. The participants were asked if they could imagine using Apiary full time and whether they would prefer to do so if it would keep their desktop more secure. All of the participants expressed a willingness to use this environment full-time, and a large majority indicated that they would prefer to use Apiary over the plain Linux environment if it would keep their data more secure.

## 4.3 Performance Measurements

### 4.3.1 Application Performance

To measure the performance overhead of Apiary on real applications, we compared the runtime performance of a number of applications within the Apiary environment against their performance in a traditional environment. To provide a conservative measure of Apiary performance, we used a container with all of the applications from all of our experiments to maximize the number of layers installed.

Table 1 lists our application tests. We focus mostly on file system benchmarks, as we have shown [4, 28] that display and operating system virtualization have little overhead. Untar tests file creation and throughput. Gzip tests file system throughput and computation. Octave is a pure computation benchmark. The kernel build tests computation and stresses the file system, because of the large number of lookups that occur due to the large size of the kernel source tree and the repeated execution of the preprocessor, compiler, and linker. To stress the system with many containers and provide a conservative performance measure, each test was run in parallel with 25 instances. To avoid out-of-memory (OOM) conditions, as the Octave benchmark requires 100 to 200 MB of memory at various points during its execution, we ran the benchmarks staggered 5 seconds apart to ensure they kept their high memory usage areas isolated to avoid the benchmarks being killed by Linux's OOM handler. Figure 3 shows that Apiary imposes almost no overhead in most cases, with about 10% overhead in the kernel build case due to the VLFS's constant need to perform lookups on the file system incurring an extra cost. This demonstrates that Apiary is able to scale to a large number of concurrent containers with minimal overhead.

### 4.3.2 Container Creation

For ephemeral containers to be useful, container instantiation must be quick and impose little overhead on application startup time. Although our user study already indicates that Apiary container instantiation overhead is not noticeable to users, we measure the overhead in two more ways. We measure both how long it takes to instantiate a VLFS and how long the application takes to start up within the container. First, we compare how long it takes to setup a VLFS against three other potential approaches to setting up the same container file system: using traditional Debian bootstrapping tools (**Create**), ex-

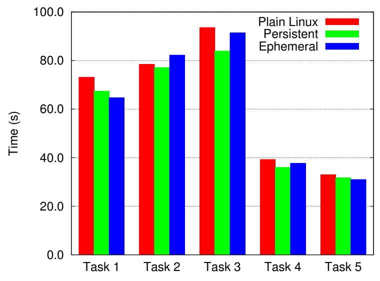


*Figure 2:* Usage Study Task Times

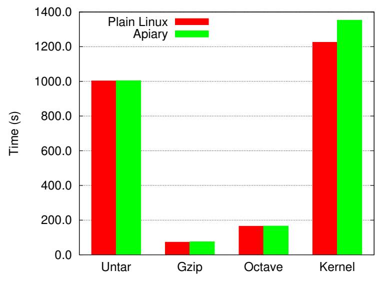


*Figure 3:* Overhead at Scale

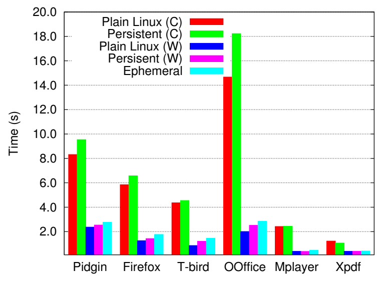


*Figure 4:* Application Startup Time

| | P | F | T | O | X | M |
|---|---|---|---|---|---|---|
| **Create (s)** | 317 | 276 | 294 | 365 | 291 | 294 |
| **Extract (s)** | 82 | 86 | 87 | 150 | 81 | 81 |
| **FS-Snap (s)** | .016 | .015 | .016 | .020 | .009 | .010 |
| **Apiary (s)** | .005 | .005 | .005 | .005 | .005 | .005 |

*Table 2:* File System Instantiating Times for (P)idgin, (F)irefox, (T)hunderbird, (O)penOffice, (X)pdf and (M)Player

tracting the file system from a tar archive (**Extract**), and using Btrfs [9], a file system with a snapshot operation, to create a new snapshot and branch of a preexisting file system namespace (**FS-Snap**). To minimize network effects with the bootstrapping tools, we used a local Debian mirror on the local 100 Mbps campus network, and were able to saturate the connection while fetching the packages to be installed.

Table 2 shows that Apiary instantiates containers with a VLFS composed of nearly 200 layers nearly instantaneously. This compares very positively with traditional ways of setting up a system. Table 2 show that it takes a significant amount of time to create a file system for the application container using either Debian's bootstrapping tool or extracting it from a tar archive. Therefore, these methods are not usable for ephemeral application containers, as users will not want to wait minutes for their applications to start. Tar archives also suffer from their need be actively maintained and rebuilt whenever they need fixes. Therefore, the amount of administrative work increases linearly with the number of applications in use. As Apiary creates the file system nearly instantaneously, it is able to support the creation of ephemeral application containers with no noticeable overhead to the users. While Table 2 shows that file systems with a snapshot and branch operation can also perform quickly, the user would have to manage each of the application's independent file systems separately.

Second we quantify startup time by measuring how long it takes for the application to open and then be automatically closed using ephemeral containers, persistent containers, and plain Linux. In the case of Firefox, Xpdf, and OpenOffice.org, this includes the time it takes to display the initial page of a document, while Pidgin, MPlayer and Thunderbird are only loading the program. For ephemeral containers, we measure the total time it takes to set up the container and execute the application within it, while for persistent containers and plain Linux, we only measure application execution time as these environments are persistent and therefore require no setup time. We compare ephemeral container application startup time to cold (C) and warm (W) cache application startup times for both plain Linux and Apiary's persistent containers. We include cold cache results for benchmarking purposes and warm cache results to demonstrate the results users would normally see.

As Figure 4, shows, the startup time overhead of running within a container versus plain Linux with no containers is generally under 25% in both cold and warm cache scenarios. This overhead is mostly due to the added overhead of opening the many files needed by today's complex applications. The most complex application, OpenOffice, imposes the most, while the least complex application, Xpdf, has negligible overhead. While the maximum absolute extra time spent in the cold cache case was nearly 5 seconds for OpenOffice, in the warm cache case it dropped to under .5 seconds. Ephemeral containers provide an interesting result. Even though they have a fresh new file system and would be thought to be equivalent to a cold cache startup, they are nearly equivalent to the warm cache case. This is because their underlying layers are already cached by the system. The ephemeral case has a slightly higher overhead due to the need to create the container and execute a display server inside of it in addition to regular application startup. However, as this takes under 10 milliseconds, it adds only a minimal amount to the ephemeral application startup time.

## 4.4 File System Efficiency

To support a large number of containers, Apiary must store and manage its file system efficiently. This means that storage space should not significantly increase with an increasing number of instantiated containers and should be easily manageable in terms of application updates. For each application's VLFS, Table 3 shows its size, its number of layers, the amount of state shared with the other application VLFSs, and the amount of state unique to it. For instance, the 129 layers that make up Firefox's VLFS require 353 MB, of which 330 MB are

| | **P** | **F** | **T** | **O** | **X** | **M** |
|---|---|---|---|---|---|---|
| **Size** (MB) | 394 | 353 | 367 | 645 | 339 | 355 |
| **# Layers** | 147 | 129 | 125 | 186 | 130 | 162 |
| **Shared** (MB) | 322 | 330 | 335 | 329 | 330 | 326 |
| **Unique** (MB) | 72 | 23 | 32 | 316 | 9 | 29 |

*Table 3:* VLFS Layer Storage Breakdown for (P)idgin, (F)irefox, (T)hunderbird, (O)penOffice, (X)pdf and (M)Player

| | | **FS Size** | **Update Time** |
|---|---|---|---|
| **1 Container** | **Plain Linux** | 815 MB | 18 s |
| | **Apiary** | 815 MB | 124 ms |
| **6 Containers** | **Plain Linux** | 2453 MB | 108 s |
| | **Apiary** | 815 MB | 745 ms |

*Table 4:* Apiary vs Traditional File System Efficiency

shared with other applications and 23 MB are unique to the Firefox VLFS. Table 3 shows that the majority of files in each container are shared with other containers.

Table 4 compares the storage requirements of a plain Linux desktop versus Apiary when using different numbers of containers to store the six applications listed in Table 3. When all the applications are installed within a single container, plain Linux and Apiary require the same amount of storage. However, when each application is installed within its own container, Apiary's VLFS imposes no additional storage requirements while the traditional Linux method of provisioning an independent file system for each container requires more than three times more disk space due to the duplication of files amongst the containers. If instead of using local desktops, multiple remote desktops are provided on a server, the VLFS usage would remain constant with the total size of all layers, while the plain Linux case would grow linearly with the number of desktops.

Table 4 demonstrates how Apiary improves the ability of users to maintain their many containers. We measured the time it took to apply a security update common to all the containers. Table 4 shows the time it took to update a single container containing all the applications, as well as all six application containers. The plain Linux case is two order of magnitude longer due its need to extract files from a package archive and copy them into the container's file system. In Apiary, no copying has to be performed. While Table 4 demonstrates that an individual update by itself does not take too long, the total time to apply common updates to many containers rises linearly with the number of containers.

## 5 Related Work

Isolation mechanisms such as VMs [39] and operating system containers [24, 32] have long been used to increase the security of applications. However, this results in applications not being integrated into the user's desktop experience. Each application is totally independent and cannot leverage another one. Products like VMware's Unity [39] attempt to solve part of this issue by combining the applications from multiple VMs into a single display with a single menu and taskbar, as well as providing file system sharing between host and VMs. While VMs provide superior isolation, they suffer higher overhead due to running independent operating systems. This impacts performance and makes them unusable for ephemeral usage on account of their long startup times. In contrast, Apiary provides lightweight containers that can support ephemeral execution.

Tahoma [34] is similar to Apiary in that it creates fully isolated application environments that remain part of a single desktop. Tahoma creates browser applications that are limited to certain resources, such as specific URLs, and that are fully isolated from each other. However, it only provides these isolated application environments for web browsers. It does not provide any way to integrate these isolated environments and does not provide ephemeral application environments. Google's Chrome web browser [17] builds upon some of these ideas to isolate web browser pages within a single browser. But the browser as a whole does not offer any isolation from the system. While its multiple-process model uses operating system mechanisms to isolate separate web pages that are concurrently viewed, it does not provide any isolation from the system itself. For instance, any plugin that is executed has the same access to the underlying system as does the user running the browser.

Modern web browsers improve privacy by providing private browsing modes that prevent browser state from being committed to disk. While they serve a similar purpose to ephemeral containers, private browsing is fundamentally different. First, it has to be written into the program itself. Many different types of programs have privacy modes to prevent them from recording state and this model requires them to implement it independently. Second, it only provides a basic level of privacy. It cannot prevent a plugin from writing state to disk. Furthermore, it makes the entire browser and any helper program or plugin that it executes part of the trusted computing base (TCB). This means that the user's entire desktop becomes part of the TCB. If any of those elements gets exploited, no privacy guarantees can be enforced. Apiary's ephemeral containers make the entire execution private and support any application with a state a user desires to remain private without any application modifications. It also keeps the TCB much smaller, by only requiring that the underlying operating system kernel and the minimal environment of Apiary's system daemon be trusted.

Apiary's ability to run multiple applications in parallel resembles Lampson's Red/Green isolation [22] and WindowBox [3]. These schemes involve users running two or more separate environments, for instance, a red environment for regular usage and a green environment for actions requiring a higher level of trust. However, unlike Apiary's ephemeral containers, if an exploit enters the green container, it will persist. Furthermore, by requiring two separate virtual machines, one increases the amount of work a user has to do to manage their machines. Apiary, by leveraging the VLFS, minimizes the overhead required required to manage multiple machines. Storage Capsules [8] also attempts to mitigate this problem by securely running the applications requiring trust in the same operating system environment as the untrusted applications, while keeping their data isolated from one another. However, this involves significant startup and teardown costs for each execution.

File systems and block devices with branching or COW semantics [7, 29, 36] can be used to create a fresh file system namespace for a new container quickly. However, these file systems do not help to manage the large number of containers that exist within Apiary. Because each container has a unique file system with different sets of applications, administrators must create individual file systems tailored to each application. They cannot create a single template file system with all applications because applications can have conflicting dependency requirements or desire to use the same file system path locations. Furthermore, if all applications are in a single file system, they are not isolated from each other. This results in a set of space-inefficient file systems, as each file system has an independent copy of many common files. This inefficiency also makes management harder. When security holes are discovered and fixed, each individual file system must be updated independently.

Many systems have been created that attempt to provide security through isolation mechanisms [1, 5, 10, 23, 25, 31, 33, 40]. All these systems differ from Apiary in that they try to isolate the many different components that make up a standard fully-integrated single system using sets of rules to determine which of the machine's resources the application should be able to access. This often results in one of two outcomes. First, a policy is created that is too strict and does not let the application run correctly. Second, a policy is created that is too lenient and lets an exploited application interact with data and applications it should not be able to access. Apiary, on the other hand, forces each components to be fully isolated within its own container before determining on which levels it should be integrated. As container setup leverages regular installation utilities to ensure all the required components are installed, it is much easier to ensure the container is setup correctly and provides all the resources that the application needs to execute. As the container is independent from all other containers on the system, no complicated rule sets have to be created to determine what it needs access to. Furthermore, rule based systems do not provide ephemeral execution and therefore if an application gets exploited, it will remain exploited, even if the exploit is confined.

Solitude [20] provides isolation via its Isolation File System (IFS), which a user can throw away. This is similar to Apiary's ephemeral containers. However, the IFSs are not fully isolated. First, Solitude does not create a new IFS for each application execution. Second, the IFS is built on top of a base file system with which it can share data, breaking the isolation. To handle this, Solitude implements taint tracking on files shared with the underlying base file system. This helps determine post facto what other applications may have been corrupted. Similarly, Solitude only provides isolation at the file system level. Because each application still shares a single display, malicious and exploited applications can leverage built-in mechanisms in commodity display architectures [13, 27] to insert events and messages into other applications sharing the display.

## 6 Conclusions

Apiary introduces a new compartmentalized application desktop paradigm. Instead of running one's applications in a single environment with complex rules to isolate the applications from each other, Apiary enables them to be easily and completely isolated while retaining the integrated feel users expect from their desktop computer. The key innovations that make this possible are the introduction of the Virtual Layered File System and the ephemeral containers they enable. The Virtual Layered File System enables the multiple containers to be stored as efficiently as a single regular desktop, while also allowing containers to be instantiated almost instantly. This functionality enables the creation of the ephemeral containers that provide an always fresh and clean environment for applications to run in. Ephemeral containers prevent malicious data from having any persistent effect on the system and isolate faults to a single application instance.

We have implemented Apiary on Linux without requiring any operating system kernel or application changes. Our results demonstrate that Apiary's containerized desktop severely reduces the threat posed by malicious files and plugins by isolating them in ephemeral containers and enabling users to quickly recover if they penetrate a persistent container. Our 24-person usage study demonstrates that Apiary is as easy to use as a regular Linux desktop by both measuring the time it took users to perform their tasks and their subjective opinions. Furthermore, we demonstrate that Apiary adds minimal overhead to application performance, is as efficient as a regular desktop in its use of storage space, and instantiates ephemeral containers in less than .5 s.

## 7 Acknowledgments

This work was supported in part by NSF grants CNS-0717544, CNS-0914845 and CNS-0905246, and AFOSR MURI grant FA9550-07-1-0527.

## References

[1] A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Applications. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, Aug. 2000.

[2] Adobe Systems Incorporated. Buffer Overflow Issue in Versions 9.0 and Earlier of Adobe Reader and Acrobat. `http://www.adobe.com/support/security/advisories/apsa09-01.html`, Feb. 2009.

[3] D. Balfanz and D. R. Simon. WindowBox: A Simple Security Model for the Connected Desktop. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle, WA, Aug. 2000.

[4] R. A. Baratto, L. N. Kim, and J. Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, Oct. 2005.

[5] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific File Protection for the UNIX Operating System. In *Proceedings of the 1995 USENIX Winter Technical Conference*, New Orleans, LA, Jan. 1995.

[6] bitdefender. Trojan.pws.chromeinject.b. `http://www.bitdefender.com/VIRUS-1000451-en--Trojan.PWS.ChromeInject.B.html`, Nov. 2008.

[7] J. Bonwick and B. Moore. ZFS: The Last Word In File Systems. `http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf`, Nov. 2005.

[8] K. Borders, E. V. Weele, B. Lau, and A. Prakash. Protecting Confidential Data on Personal Computers with Storage Capsules. In *Proceedings of the 18th USENIX Security Symposium*, Montreal. Canada, Aug. 2009.

[9] Btrfs. `https://btrfs.wiki.kernel.org`.

[10] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th USENIX Systems Administration Conference*, New Orleans, LA, Dec. 2000.

[11] B. Cumberland, G. Carius, and A. Muir. *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference*. Microsoft Press, Aug. 1999.

[12] J. Fernandez-Sanguino. Debian GNU/Linux FAQ - Chapter 8 - The Debian Package Management Tools. `http://www.debian.org/doc/FAQ/ch-pkgtools.en.html`.

[13] J. Gettys and R. W. Scheifler. *Xlib - C Language X Interface*. X Consortium, Inc., 1996.

[14] M. Gilmore. 10Day CERT Advisory on PDF Files. `http://seclists.org/fulldisclosure/2003/Jun/0463.html`, June 2003.

[15] Gnome.org. Libwnck Reference Manual. `http://library.gnome.org/devel/libwnck/`.

[16] GOBBLES Security. Local/Remote Mpg123 Exploit. `http://www.opennet.ru/base/exploits/1042565884_668.txt.html`, Jan. 2003.

[17] Google. Google Chrome - Features. `http://www.google.com/chrome/intl/en/features.html`.

[18] Greasemonkey. `http://www.greasespot.net/`.

[19] P. Grosjean. Speed Comparison of Various Number Crunching Packages (Version 2). `http://www.sciviews.org/benchmark/`, Mar. 2003.

[20] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-level Isolation and Recovery with Solitude. In *Proceedings of the 3rd ACM European Conference on Computer Systems*, Glasgow, Scotland, Apr. 2008.

[21] P.-H. Kamp and R. N. M. Watson. Jails: Confining the Omnipotent Root. In *Proceedings of the 2nd International SANE Conference*, MECC, Maastricht, The Netherlands, May 2000.

[22] B. Lampson. Accountability and Freedom. `http://research.microsoft.com/en-us/um/people/blampson/slides/accountabilityandfreedom.ppt`, Sept. 2005.

[23] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, NV, Dec. 2003.

[24] Linux Containers. `http://lxc.sourceforge.net/`.

[25] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the 2001 USENIX Annual Technical Conference: FREENIX Track*, Boston, MA, June 2001.

[26] Microsoft Application Virtualization. `http://www.microsoft.com/systemcenter/appv/`.

[27] Microsoft Corp. SendMessage Function. `http://msdn.microsoft.com/en-us/library/ms644950.aspx`.

[28] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[29] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *Proceedings of the 3rd Symposium of Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[30] T. Porter and T. Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[31] S. Potter, J. Nieh, and M. Selsky. Secure Isolation of Untrusted Legacy Applications. In *Proceedings of the 21st Conference on Large Installation System Administration*, Dallas, TX, Nov. 2007.

[32] D. Price and A. Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th Large Installation System Administration Conference*, Atlanta, GA, Nov. 2004.

[33] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, Aug. 2003.

[34] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems*, Nuremberg, Germany, Mar. 2009.

[35] RPM Package Manager. `http://www.rpm.org/`.

[36] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[37] S. Uchino. MetaVNC - A Window Aware VNC. `http://metavnc.sourceforge.net/`.

[38] Virtual Network Computing. `http://www.realvnc.com/`.

[39] VMware Inc. VMware Worksation 6.5 Release Notes. `http://www.vmware.com/support/ws65/doc/releasenotes_ws65.html`, Oct. 2008.

[40] D. Wagner. Janus: An Approach for Confinement of Untrusted Applications. Master's thesis, University of California, Berkeley, Aug. 1999.

[41] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage*, 2(1):1–32, Feb. 2006.