Virtics: A System for Privilege Separation of Legacy Desktop Applications

Matt Piotrowski
Anthony D. Joseph

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-70
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-70.html

May 13, 2010

Virtics: A System for Privilege Separation of Legacy Desktop Applications

Matt Piotrowski
University of California, Berkeley

Anthony D. Joseph
Intel Labs Berkeley
University of California, Berkeley

Abstract

Legacy desktop applications – the applications in use on most desktops today – often process data from multiple untrusted sources. If an application makes a mistake when processing this data, the integrity of the application, and potentially the entire system, can be compromised. We introduce a new operating system primitive that enables an application running on a legacy OS to efficiently create unprivileged virtual machines when dealing with untrusted data. These virtual machines can then perform all of the complex operations needed to process and render the application's data. The resulting window content is transparently mapped into the window space of the application. Using this primitive, we built an evince-based PDF viewer that limits PDF exploits to controlling an unprivileged virtual machine with file access only to the PDF itself. We also built a WebKit-based web browser which limits browser exploits to controlling an unprivileged virtual machine with access solely to the contents of the tab in which the exploit occurred. We further show how a whole suite of desktop applications can use our new primitive to separate privileges when dealing with untrusted data. Moving recursively upwards, we can view the operating system itself as an application that needs privilege separation when dealing with untrusted data from multiple sources (i.e. the different applications it runs). We describe a prototype implementation of an operating system that manages its applications in this way.

1 Introduction

Personal computers can be compromised using a variety of approaches, including malicious applications unknowingly installed on a system (e.g., spyware or other malware), malicious documents (e.g., documents that exploit document processing errors such as buffer overruns or counter overflows), and network processing errors (e.g., carefully crafted network streams that when processed by the recipient exploit vulnerabilities in a manner similar to malicious documents). Preventing malicious applications, documents, and network flows from reaching PC's is challenging, given the large number of applications and document types and ever increasing application code complexities. Even an application thought to previously be "secure" becomes an attack vector when a vulnerability is identified. A security hole in the system exists until the application is patched or removed from the system. In an in-depth scan of 20,000 personal computers, Secunia found that only 2% of the PC's were fully patched, while 30% of PC's had 1-5 insecure programs, 25% had 6-10 insecure programs, and 46% had 11 or more insecure programs [1]. Even well-managed enterprise PC's are at risk: referring to the recent targeted attacks against high-profile US companies (Google, et al.), a security investigator remarked, "All of the victims we've worked with had perfectly installed antivirus. They all had intrusion detection systems and several had Web proxies scan content" [12].

The traditional approach to security and isolation in operating systems has two serious flaws: a compromised or malicious application can access any data at the user's privilege level, and more sophisticated attacks can use the wide OS interface exposed to applications to perform privilege-elevation and compromise the entire system. Attackers are aware of these flaws and exploit them to gain administrative access to machines, install rootkits, access files, and collect sensitive information. Clearly, a stronger isolation model is needed to prevent these attacks.

One simple way to address these problems would be for a user to have two laptops, one for "work" and one for "play". This would allow the user to do potentially dangerous things on the "play" laptop and not worry that the data on the "work" laptop could be affected. However, on each laptop the user will likely use many applications and many documents, each of which has varying levels of sensitivity. If any one of these is malicious, the whole laptop could be compromised. As a

paper exercise, consider the following *infinite laptop* scenario: assume that you have an infinite number of laptops and you can switch your hands and eyes between them quickly and seamlessly. You could use this environment to greatly reduce the threats from malware and spyware as follows.

First, run each application on a separate laptop. That way, if an application is spyware or malware, it won't be able to interact with files or intercept keystrokes on other laptops. Furthermore, if a trusted application on one laptop becomes infected when handling a malicious document, the damage will be limited to that laptop, not any other laptops running other applications.

Second, since we have an infinite number of laptops, we can apply this approach recursively to applications themselves: when working with a new document, we give it its own dedicated laptop. So instead of a laptop that runs application A opening documents X and Y, we now have two laptops running application A, with one dedicated to working with document X and the other dedicated to working with document Y. Thus, the damage from a malicious document X will be limited to the laptop dedicated to working with malicious document X and not any other laptops. This perfect isolation is a huge increase in security.

However, since some documents need to be accessed by multiple applications and some applications need to access multiple documents simultaneously, the third step is to augment the infinite laptop model to allow us to explicitly run a cord between two laptops to allow them to share documents. This added connectivity is a double-edged sword, since it enables damage to propagate from one laptop to others, limited by the set of laptops reachable from that laptop by cords. Note that for a single malicious document shared between multiple applications, any damage caused by the document is still limited to just that document, maintaining perfect isolation. For a single laptop accessing multiple documents (i.e., some form of aggregation application), the damage from a malicious document is limited to the set of documents being aggregated, which is a significant increase in security.

While the infinite laptop model is interesting, it is obviously infeasible. However, recent advances in high-performance, hardware-based virtualization technology enable a large number of virtual machines to run simultaneously on a single machine, allowing us to approximate the infinite laptop model. We take advantage of this fact and the fact that VM's offer a simple interface for backwards compatibility. This simple interface is amenable to security while the backwards compatibility allows us to leverage an enormous body of existing code.

1.1 Our Contributions

- We introduce a new primitive for privilege separation that enables applications to create efficient, high-performance unprivileged virtual machines, perform complex operations on untrusted data in these virtual machines, and then have the results transparently mapped into the application's windows. The ability for an application to spawn a subapplication and have it render in part of its windows can be achieved using session capabilities in the EROS Trusted Window System [20]; however, EROS does not support legacy codebases. Since we implement our primitive using virtual machines that run Linux and X11, legacy code runs unmodified in our unprivileged environment.

- We design "armored" versions of existing applications that use our new primitive, and we describe the implementation of two of these, a PDF reader and a web browser

- We describe how to build a high-performance OS as a privilege separating application and discuss our prototype implementation of such a system, which has been in daily use for over a year. This prototype can run a large number of existing applications without modification. Although our prototype is built on Linux for convenience of implementation, the concepts would allow other commodity operating systems (e.g. Windows and Mac OS X) to protect themselves without changing existing applications.

1.2 Roadmap

The core of our work is based on the new primitive we have developed. After discussing our threat model and related work, we describe this new primitive. At first, we only consider applications running as normal processes that use our new primitive to spawn VM's to protect themselves. Later, we look at an operating system that spawns VM's whenever it runs applications; in this system, no application runs as a normal process outside a VM. These applications running in a VM could protect themselves recursively by spawning their own VM's, but there is no requirement that they do so. After describing our designs and implementations, we examine their performance and show that although there is overhead associated with our system, it is not unreasonable given the advances in security, and furthermore, increased efforts to optimize (in particular, memory sharing) will make it even smaller. We discuss how our system is very usable and doesn't place a heavy burden on the user, relying on familiar concepts and utilizing existing actions that a user normally performs in order to make security deci-

sions. We wrap up with a discussion of limitations and some conclusions.

2 Environment

2.1 Threat Model

We assume that the applications installed on the system are not malicious and want to protect themselves from compromise. Later, when considering the operating system itself as a privilege-separating application, we relax this assumption and consider malicious applications.

The adversary we are trying to protect against is remote across the Internet and can supply arbitrary malicious data to applications. This may take the form of a malicious document (e.g., a malicious PDF [13]) or a malicious network flow (e.g. a malicious HTTP attribute [14]).

What we are trying to protect is the integrity of the application and the integrity of the system. In protecting the integrity of the application, we want to prevent malicious code from running with all of the privileges of the application and instead have it run inside a virtual machine with only network access and a subset of the application's data (ideally, this would just be the document or network flow the malicious code was attached to). In protecting the integrity of the system, we similarly want to prevent malicious code from running with all of the privileges of the system and the user of the system.

2.2 Related Work

Previous mechanisms for privilege separation [4] [16] focus on non-graphical applications. We provide a privilege separation mechanism that at its very core expects an application to use it to divide itself up graphically. The closest work in this respect is a full capability system such as the EROS Trusted Window System [20]. However, as we mentioned earlier, the EROS system does not support legacy desktop applications.

SELinux [10] is an alternate sandbox mechanism which allows resources and executing code to be assigned labels. Rules exist for which labels can access which resources and how to transition from one label to another. One problem with SELinux is that it does not support dynamic rulesets or label creation. The set of rules and labels are static once a system boots. So our mechanism for protecting user files where applications are given access dynamically based on user file dialogs is not directly expressible in SELinux. Another problem with SELinux is that application policies are expected to be created by a knowledgeable system administrator. For example, should the application be able to write to /usr/lib, or /var/run/foo? This approach works fine when you have a knowledgeable administrator to make the decision, but in our scenario ordinary users are installing arbitrary applications from the Internet. The application developer could write the SELinux policy needed for the application to run, but we do not trust the developer. The best one could hope for in this case would be to allow the application to write to any part of the system files it needs to but make these changes visible only to the application; this would require machinery outside of SELinux to replicate the functionality provided by our private disks model (Section 6.2.2). A third problem with SELinux is that all applications still run on the same kernel and share the X server. So a vulnerability in, for example, TCP would compromise the entire system. Exposing the entire X server to a potentially malicious application is also unwise. Recent efforts by the developers of SELinux to bring limited sandboxing to graphical applications [26] have run each application with its own nested X server. However, this nested X server still has access to the entire trusted X server. Because the nested X server must be considered compromisable due to its complexity, this approach does not provide significant extra security. To improve the security substantially, the interface between the nested X server and the trusted X server must be greatly narrowed, which is exactly the approach our graphics system takes. Furthermore, by duplicating functionality like the X server in order to run desktop applications securely, SELinux loses its appeal as a lightweight sandboxing mechanism.

Xax [6] and Native Client [33] are interesting technologies for running native code in a browser; however, they are not well-suited for code that requires "direct file system access or unrestricted access to the network", which is a focus of our work.

Terra [8] was a system that ran applications in VM's for trusted computing purposes. Their focus was not on weaving the different VM's into a coherent desktop as they do not provide a secure graphics subsystem or an easy way to securely share files between VM's.

Polaris [21] is a system that attempts to isolate applications on Windows by running them in separate user accounts and requiring file dialogs for the user's files. However, each application is still running on Windows with the wide system interface that entails. Also, Polaris does not provide a mechanism for an application to protect itself when dealing with a potentially malicious document.

The Tahoma [5] secure web browser utilizes virtual machines for isolation of different sites. In many ways, Tahoma is a subset of Virtics and has isolation properties similar to our Armored WebKit (Section **??**). The Tahoma browser is an example of one of many secure applications that can be built using the primitive our system provides.

Chromium [3] uses processes to partition the web browser, separating out render processes from the main browser process for fault tolerance and security.

Chromium runs parts of WebKit in the trusted browser process, which causes it to only be immune to 70% of the arbitrary code execution vulnerabilities in WebKit instead of being at 100%. Chromium also does not provide a high level of inter-render process isolation, so if there is a compromise in a render process, the browser process may be protected, but one render process could execute a shatter attack against another. The ability to access arbitrary cookies is also an avenue of attack available to a corrupted render process. Chromium's site instances [17] divide up the web browser in a similar way to our Armored WebKit's tabs. Gazelle [27] takes the isolation in Chromium a step further by isolating separate domains within a site instance into separate processes, solving the cookie problem mentioned above. It also provides protection within a tab, unlike our Armored WebKit.

The Qubes [18] operating system utilizes virtual machines to isolate components, similar to Virtics. Qubes does a great job of narrowing the VM interface even further by moving components like the network subsystem and the storage subsystem into their own VM's. In contrast, Virtics does not provide VM isolation of OS components, instead focusing on VM isolation of applications and subapplications. Qubes also carries out VM isolation of applications, but uses a different approach. In Qubes, a user can create multiple VM's and label them with different purposes. These VM's can each run multiple applications internally but the label that has been applied to their VM is clearly visible when the user is interacting with one of their windows. As a basic example, consider a user who has created two VM's and labeled them with the purposes "sensitive" and "insensitive". The user has a PDF reader available in both VM's. In the "sensitive" VM they open a PDF on corporate strategy. In the "insensitive" VM they open a random PDF they have received from an untrusted source. If the PDF they opened in the "insensitive" VM causes corruption of the PDF reader, it is isolated to the "insensitive" VM and cannot access the contents of the corporate strategy document in the "sensitive" VM. This type of isolation provides strong protection similar to what would be achieved by opening both PDF's in our Armored PDF Reader. The difference is that in Qubes the user has to be very proactive about creating VM's with different purposes and has to be careful to work with data only in the appropriate VM; in Virtics, the user is automatically working with a separate VM when working with separate applications and separate documents. As a further example of this, assume that the corporate strategy PDF has a hyperlink embedded in it that points to an external resource. If the user opens this hyperlink in the browser of the "sensitive" VM, a compromise of the browser will allow access to the corporate strategy PDF as well; the content at the external resource does not even have to be malicious for this to occur; it could be the case that the content is at an HTTP url instead of an HTTPS url and someone on the network could lie about the DNS translation and use this to return malicious content instead of the legitimate content. On the other hand, the user could take the initiative to open the hyperlink in the browser of the "insensitive" VM; this would prevent any browser exploit from having access to the corporate strategy document; however, it is worth noting that if the content in the external resource is semi-sensitive, we may have just exposed it to all of the malcode that is running in the "insensitive" VM from previous actions. In contrast, in Virtics the hyperlink would be handled by the Armored Web Browser and it would be opened in its own virtual machine corresponding to its own tab. If an exploit happens when processing this external content, the malcode will not have access to the corporate strategy PDF. Another difference between Qubes and Virtics is how files are shared across VM's. In Qubes, the user must explicitly open up a separate explorer-like tool to copy files from one VM to another. In Virtics, we use the open/save dialogs that a user normally interacts with in GUI applications to access files from multiple VM's.

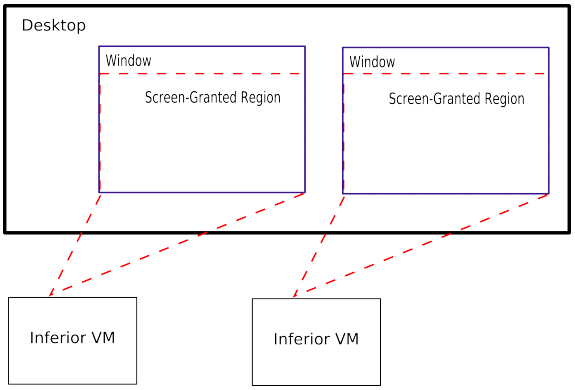

Figure 1: An example of an application that has spawned two inferiors and screen-granted them to two different windows

3 A New VM Primitive

3.1 Design

In this section, we describe the design of our new primitive for privilege separation of desktop applications. The design centers around the ability to create unprivileged VM's (called inferior VM's) and then screen grant and input grant them the ability to render to and interact with a portion of the application's windows.

3.1.1 Inferior VM

An application can ask the system for an inferior VM to be created. This VM uses a previously installed disk im-

age (see section 3.2.1 for installation details). This disk image is immutable to the running inferior VM. The immutable nature of the inferior VM's disk allows multiple instances to run concurrently using the same image. An inferior VM is unprivileged in terms of its access to the rest of the system. In particular, it has no access to any of the user's documents. It has only network access and a set of pipes to the application that created it. The set of pipes serve as a secure communication channel over which a custom application/inferior protocol can be defined.

3.1.2 Screen Grant

An application can give an inferior VM the ability to display in a portion of one of its windows by issuing a screen grant to the system. The screen grant specifies the application window to use and the location and dimensions of a rectangle within that window that is to be given to the inferior. The system transparently maps the output of the inferior VM to this portion of the application's window. Figure 1 shows an application that has started two inferiors and screen-granted them the lower majority of two windows. This could represent a PDF viewer that is displaying two documents and is reserving the upper portion of the windows for a menu bar that it controls while leaving the complex output generated from handling the PDF's themselves to the inferior VM's. If the inferior VM generates any popup windows, these are restricted to displaying in the area that has been granted.

3.1.3 Input Grant

Inferior windows that are being displayed receive input events just like normal windows. That is, when a mouse event occurs on the screen-granted region, it is delivered to the inferior; and when the screen-granted region has received the keyboard focus by being clicked, keyboard events are delivered to the inferior. However, at any time, the application that started the inferior can issue an InputGrant event to transfer the keyboard focus to the inferior or remove it.

3.1.4 Copy and Paste

It is important that an inferior VM does not have *carte blanche* access to the clipboard, both for copying and for pasting. Copying is a concern because malicious code could copy data to the clipboard with the hopes that it will be pasted into a vulnerable environment. Pasting is a concern because there may be sensitive information in the clipboard that malicious code would like to obtain access to. Instead of blanket access, the system traps the <ctrl-c> and <ctrl-v> key sequences to discern the user's copy and paste intent. When the user types <ctrl-c>, the system will take note of this sequence and the inferior VM it is intended for; when that inferior VM then asks to copy data to the clipboard, the request will succeed; similarly, when the user types <ctrl-v>, the system will take note of this sequence and the inferior VM it is intended for; when that inferior VM then asks to paste data from the clipboard, the request will succeed. This model for copy and paste has two main drawbacks: (1) applications that use those key sequences for operations other than copy and paste will gain access to the clipboard at times the user didn't intend them to, and (2) other actions that normally would result in a copy operation, such as selecting edit->copy from a menu, won't succeed unless preceded by a <ctrl-c>.

3.2 Implementation

We implement our new primitive using Linux and KVM virtualization technology. The inferior VM's run a full Linux and X11 stack. Our focus in providing this type of environment for inferior VM's was on backwards compatibility. That is, there is a large amount of code that knows how to run on Linux and how to display to an X11 window system. We can run this code unmodified.

3.2.1 Installing an Inferior

All inferior disk images are overlaid on top of a system base image containing gigabytes of common libraries. This allows an application to specify an inferior's disk image by providing essentially a file diff between the system base image and the complete set of files needed to run the inferior code. Once the inferior disk image is created, it is immutable. An application is free to create multiple inferior images if it wants.

3.2.2 Running an Inferior

An application starts an inferior VM using the ID it received when installing the inferior image. An application can start multiple instances that use the same inferior image. Each instance gets its own VM with access to the inferior image that is overlaid internally with a RAM disk to allow writes while the instance is running. An application that starts an inferior VM gets back a set of pipes that it can use to send and receive data to/from the inferior.

3.2.3 Screen Grants and Input Grants

When an application issues a screen grant or input grant, it goes to a program called the virtual window manager server. There is a corresponding program called the virtual window manager client that runs in the inferior VM. This client acts as a window manager for the nested X11 system running in the inferior VM. When window events happen in the inferior, they are forwarded along by the client to the virtual window manager server, which can act on them based on the current status of screen grants and input grants. Window contents are communicated through shared memory. This memory can be anywhere in the memory owned by the inferior VM. The server simply maps it, reads the raw pixel data, and pushes it

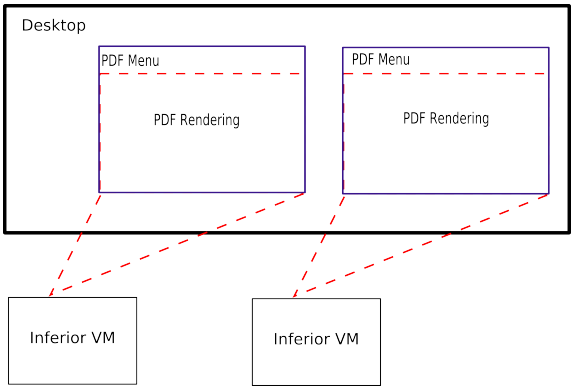

Figure 2: An armored PDF viewer with two PDF's open.

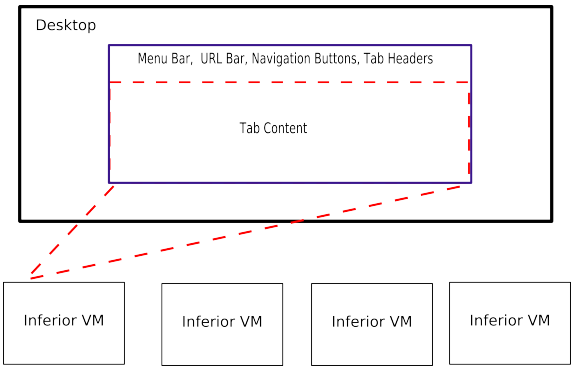

Figure 3: An armored web browser with four tabs.

to the screen in the proper location. It is worth repeating that the inferior does not carry out the full X protocol with the trusted X server running in the host. Instead, it carries out a greatly limited protocol via the virtual window manager client. The general idea of rendering in a VM for security was discussed in [7].

4 Armored Applications Design

In this section, we discuss how a suite of desktop applications might "armor" themselves using our new primitive. Each armored app consists of two parts, an application part and an inferior part. The application part runs with the full privileges of the application, while the inferior part runs with the limited privileges that an inferior VM gets plus any privileges given to it by the application-specific app/inferior protocol. We try to keep this app/inferior protocol as simple as possible. We also try to push as much complexity as possible into the inferior VM, leaving the application part to deal with simple user interactions (e.g. menus) and simple data management (e.g. reading and writing raw bytes from disk). The idea here is that the complex code that operates on untrusted data is the most likely code to be exploited and we want this code to be running in an inferior so that when an exploit occurs, it is isolated.

4.1 Armored PDF Viewer

The main job of the application part of our armored PDF viewer is to control the menu bar (i.e. File, Edit, View). When a PDF is opened, we create a new window and a new inferior VM. We screen grant everything below the menu bar to the inferior VM. We use the pipe we obtained when we created the inferior to send it an OpenFile event followed by a stream of OpenFileBytes events specifying the content of the PDF that is to be displayed. The inferior VM performs whatever complex operations it needs to do to render the PDF to the screen. Because of the transparent delivery of input events, the user can then interact with the content of the PDF without intervention from the application. If the user selects a menu item from the View or Edit menus (i.e. Find, Find Next, Copy, Zoom In, or Zoom Out), the appropriate event is sent from the application to the inferior. If the user chooses to open a different file, a new window and a new inferior VM are created. Figure 2 shows an armored PDF viewer that is displaying two files.

4.1.1 Compromise of an Inferior VM

If one of the PDF's is malicious and successfully exploits the rendering engine, it will find itself in an unprivileged VM. From there it could try to attack the application over the communication pipe; however, the protocol defined over the pipe is designed to be simple and in this case is actually unidirectional from the application to the inferior, so the application will not even be processing events from the inferior. Another avenue of attack is to go after the virtual machine environment it is running in. This is the more likely avenue of attack but the idea is that this interface can be made simpler than an OS interface and significant effort can be focused on securing it.

4.2 Armored Web Browser

Our model for the armored web browser centers around tabs and registry-controlled domain names (e.g. a.com, b.com, etc.). Each tab in the browser is associated with a different inferior VM. The application part of the browser contains the menu bar, the URL bar, the navigation buttons, and the tab headers. The area below the tab headers is screen-granted to the inferior VM whose tab is currently selected. Figure 3 shows an armored web browser currently displaying four tabs.

4.2.1 Registry-Controlled Domain Labeling of Inferior VM's

The armored web browser labels each inferior VM with a protocol (e.g. http, ftp, https) and a registry-controlled domain. This labeling is immutable for the lifetime of the inferior. If a tab wants to be labeled with a different protocol or a different registry-controlled domain, it must

make a URLRequest to the application, which will create a new inferior, label it with the new protocol and registry-controlled domain, and assign it to the tab, discarding the old inferior. Note that for purposes of implementing a "back" button we could keep old inferiors around or dump the state needed to recreate them. We don't further consider the design of the back button here.

The labeling of an inferior is significant because it determines not only what is displayed in parts of the URL bar but also the cookies that the inferior can ask to receive or set (see Section 4.2.2). To illustrate the flow of inferiors assigned to a tab, consider a user that starts browsing http://a.com. A new inferior will be created and labeled with (http, a.com). This inferior will be assigned to the tab and given the screen grant for the area below the tab headers. The inferior will fetch the document, process it, and render the results to the screen. The user interacts with the contents as normal. If the user clicks a link to http://a.com/some_inner_path.html or http://innerdomain.a.com, the inferior VM will request that the application change the URL visually and will then fetch the new document, render it, and allow the user to interact with it. Since in both cases the registry-controlled domain was not changed, the label applied to the tab is not changed. If the user instead clicks a link to go to http://b.com, the inferior VM will request that the application fetch the requested URL and send any form data that goes along with the request. The application will create a new inferior, label it with (http, b.com), assign it to the tab, and give it the screen grant. The old inferior that was assigned to the tab will be discarded. The new inferior will fetch the document at http://b.com and perform all of the operations needed to render the document to the screen. The user will then be able to interact with the contents as normal and the process repeats. This tab model is similar to the process-based site instance model of the Chromium web browser [17].

4.2.2 Cookies

An inferior VM may ask the application to get/set cookies for the registry-controlled domain it is labeled with. It does not have access to the cookies for any other registry-controlled domain. This is true even if the document it is rendering requires the inferior to fetch objects from other domains. However, an inferior VM will maintain a temporary set of cookies it receives from other domains. This set of temporary cookies will be used by the inferior as long as it is running. After it is discarded, only the cookies that were set for its labeled registry-controlled domain will be accessible by other inferiors. An alternative to our cookie model would be for the application to perform the HTTP fetches; however, this is more complexity than we wish to add to the application part.

4.2.3 Secure Cookies

An inferior may only get/set the secure cookies for its registry-controlled domain if it is labeled with the https protocol. Recall that to switch from http to https, a new inferior must be used, and similarly, to switch from https to http, a new inferior must be used. This does not prevent an https document from including http objects or objects from another domain, which, if malicious, gain access to the secure cookies with a successful compromise of the inferior VM. However, it does prevent a compromised inferior VM that is labeled with (http, a.com) from accessing the secure cookies for a.com .

4.2.4 URLRequest Parsing Code

In order to protect the security of cookies, we need to be able to properly handle a URLRequest. That is, we need to be able to parse a URL and any associated form data into its components. We note that we do not perform request parsing in the application itself. Instead, we take advantage of the fact that a new inferior is in a known uncompromised state. Therefore, we can have the first action of a new inferior be to calculate its own label from an arbitrary URLRequest made by another inferior and inform us of the result. We can trust this result as much as we could trust the application doing its own calculation of the label. The importance of proper request parsing becomes clear with an example. Suppose that there were a bug in the parsing code that resulted in arbitrary code execution for specially formatted paths. If this were the case, then a malicious website could ask us to navigate to http://t.com/malicious_path where t.com is an arbitrary registry-controlled domain the malicious website wants to steal the cookies of. We will create a new inferior and give it the URL to navigate to; however, the inferior will mishandle the path information, potentially running arbitrary code; this arbitrary code could then perform operations while being labeled with (http, t.com). The best we can do to combat this problem in our model is to make the parsing code as simple and secure as possible.

4.2.5 Uploading and Downloading

When an inferior VM wants to upload a file, it sends an OpenFile event to the application. If the inferior VM represents the currently selected tab, the application displays an open file dialog to the user. The inferior VM is informed of the selected file and can access its contents through a series of OpenFileGetBytes events to the application. Similarly, if an inferior VM wants to save a file, it sends a SaveFile event to the application. If the inferior VM represents the currently selected tab, the application displays a save file dialog to the user. The inferior VM is informed of the selected file and can set its contents through a series of SaveFileSetBytes events to the appli-

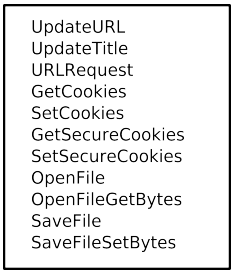

Figure 4: The events in the application/inferior protocol for the armored web browser.

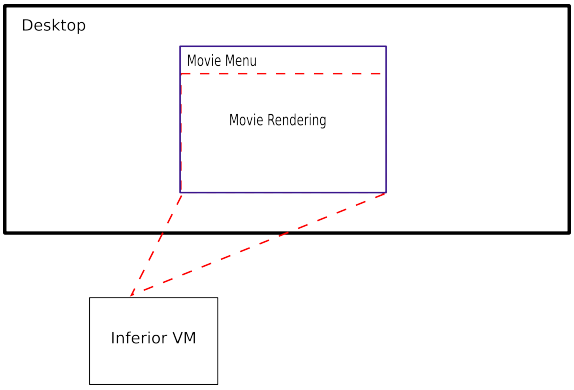

Figure 5: An armored movie player playing a movie.

cation.

4.2.6 Popup Windows

An inferior VM is free to pop up windows within the screen-granted region. If it wants to open a URL in a new window outside of its region or in a new tab, it must indicate this in a URLRequest to the application and thus cede any control over that content.

4.2.7 Plugins and Extensions

Browser plugins lend themselves well to our model. Plugins that exist to handle new types of objects (e.g., a Flash plugin, a PDF plugin, or Java plugin) work well. These plugins just run inside the inferior VM as one would expect. Browser extensions, on the other hand, need to hook themselves into the browser in deeper ways (e.g., an extension that adds a toolbar). In our model, browser extensions become part of the code base running in the application part, and therefore need to be trusted.

4.2.8 Compromise of an Inferior VM

If one of the inferior VM's gets compromised, malicious code will find itself not running with the privileges of the browser but rather as an unprivileged VM with access to the contents of the tab, access to the cookies of the registry-controlled domain the tab is labeled with, and network access. As in the case of the armored PDF viewer, the malicious code could try to elevate its privileges by attacking the application/inferior protocol or the virtual machine environment. Although the application/inferior protocol is richer than in the armored PDF viewer, it is still kept relatively simple. Figure 4 provides a recap of the events in the application/inferior protocol.

4.3 Armored Movie Player

The main job of the application part of our armored movie player is to control the menu bar. When a movie is opened, we create a new window and a new inferior VM. We screen grant everything below the menu bar to the inferior VM. The inferior is sent a PlayMovie event informing it of the size (in bytes) of the movie it is about to play. The inferior can then issue GetFileBytes events to read pieces of the file it is playing. The inferior performs all of the complex decoding of the video and audio and displays the result to the screen. Since the inferior has no way of producing sound, it must send OutputSound events to the application. These events contain raw sound data that can be pushed directly to the native sound system. Figure 5 shows an armored movie player that is displaying a movie.

4.3.1 Fullscreen Viewing

If the user selects to view the movie in full screen, the window is resized and a screen grant for the whole window is given to the inferior. This action carries the risk of a compromised inferior VM pretending to exit, remaining in fullscreen mode, and then emulating the user interface of the system. The inferior VM could then hope to trick the user into entering sensitive data. This attack could be combatted using various methods, including a small border around even full screen windows.

4.3.2 Compromise of an Inferior VM

If a malicious movie compromises an inferior VM, in addition to the attack mentioned above, it could try to attack the application/inferior protocol, which is very simple, or, as with the other armored applications, the virtual machine environment.

4.4 Armored Email Client

Our model for the armored email client utilizes inferiors in multiple situations. First of all, the preview pane (see Figure 6) is screen-granted to an inferior, while the message summary pane and folder/contact pane is controlled by the app. Whenever a new email is being previewed, a new inferior is created to preview that email. If an email is opened for reading in its own window, a new window is created along with a new inferior that is screen-granted everything below the menu bar. The inferior VM is informed of the email with a DisplayEmail event and proceeds to obtain the bytes of the email by is-

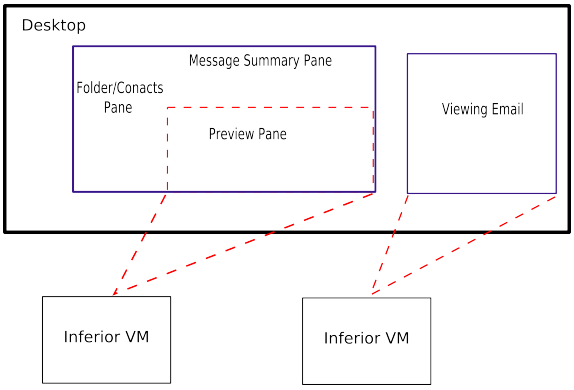

Figure 6: An armored email client previewing one email and viewing another.

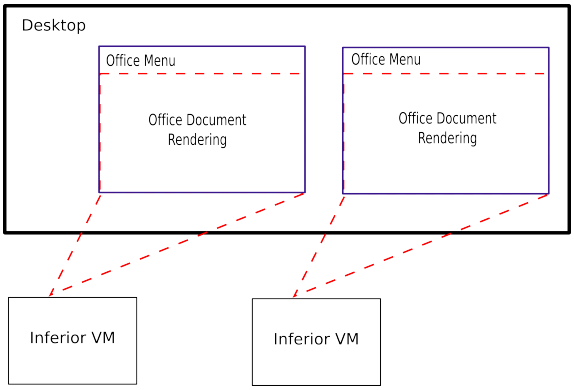

Figure 7: An armored office application working on two office documents.

suing EmailGetBytes events to the app. It then performs whatever complex rendering is necessary to display the email. If the user goes to compose an email, a new window and a new inferior are created for the composition. Everything below the menu bar is screen-granted to the inferior. When the user chooses to send the mail, the app issues a series of MailGetBytes events to obtain the message to send. Because carrying out popular email protocols (IMAP, POP, SMTP) is more complex than any operation we want to run in the app part, we also create protocol inferior VM's. There is one protocol inferior VM for each combination of protocol and server we communicate with. So in the standard setup of one server with IMAP and SMTP, there would be one inferior VM that carries out the IMAP protocol with that server and another that carries out SMTP with that server. These inferior VM's would have no on-screen representation but would allow us to carry out the protocols without the privileges of the app.

4.4.1 Compromise of an Inferior VM

If a malicious email were to compromise the rendering engine, it would find itself in an unprivileged VM with access to the network and access to the email it used as a vector. Compromise of an inferior VM used for composing an email (e.g. through rendering of quoted text) is more serious. The compromised inferior VM could lie about the recipients and the body of the email and attempt to send spam. This is mitigated by the fact that the app part must be told to send the message by the user (thus greatly reducing the rate of propagation). It is worth noting also that the user must be replying to an email with malicious content in order for this attack to work. The most serious of the inferior VM compromises would be a compromise of a protocol VM. This could be used to read all of the user's email from a particular server or send out volumes of spam. On the bright side, email from other servers is not automatically compromised and the integrity of the system remains intact.

4.5 Armored Office

The main job of the application part of armored office is to control the menu bar. When a document is opened, we create a new window and a new inferior VM. We screen grant everything below the menu bar to the inferior VM. We then send the inferior an OpenFile or NewFile event informing it of its purpose (i.e. to work on an existing document or create a new one). For an OpenFile event, the inferior VM can issue OpenFileGetBytes events to read the contents of the file. It then performs whatever complex operations it needs to do to display the file and is responsible for the creation of any new content. When the inferior VM wants to save the contents of the document it is working on, it issues SaveFileSetBytes events to the application. A feature like autosave can be supported in a similar manner. If a document wants to embed another document inside of it, the inferior VM can issue an IncludeFile event to the application. The application will then display a dialog to the user, asking them which file to embed. The inferior VM is informed of the selected file and can access the file's contents with OpenFileGetBytes events. To keep the included file in sync, the application would monitor its most recent modification time and send an OpenFileChanged event to the inferior VM that has included the file if the file changes. Figure 7 shows armored office working with two office documents.

4.5.1 Compromise of an Inferior VM

If malicious code compromised an inferior VM, it would have access to the document it used as a vector and any documents that the user chose to embed in that document. Other documents opened by armored office and the system itself would be safe provided that the application/inferior protocol and the virtual machine environment cannot be compromised.

armored evince superior	4425	armored webkit superior	4970
armored evince inferior	1362	armored webkit inferior	1618

Figure 8: Lines of C code in the armored app implementations as counted by cloc.

5 Armored Applications Implementation

We carried out the implementation of an armored PDF viewer and an armored web browser.

5.1 Armored Evince

The Evince PDF viewer nicely encapsulates all of its complex rendering in the library libevview. Therefore, with a small amount of wrapper code to shuttle events between the application part and the inferior running libevview, we are able to isolate all of the complex code involved in rendering a PDF to running inside an inferior VM. No libevview code runs in the application VM. The number of lines of code added is given in Figure 8.

5.2 Armored WebKit

The WebKit library nicely encapsulates all of the complex code needed to handle modern webpages. It provides hooks allowing the code using it to make decisions about what to do before loading a new page and how cookies should be accessed. We created an Armored WebKit where no WebKit code runs outside of inferiors. In addition to the glue code needed to shuttle events back and forth, we made two slight modifications to the WebKit library; one modification was to contact the application when cookies for the labeled registry-controlled domain were being accessed; another modification was to stream uploads and downloads to/from the application instead of the local file system. The number of lines of code added is given in Figure 8.

6 OS as a Privilege-Separating Application Design

So far, we have assumed that applications themselves are not malicious and that they will actively try to protect themselves using our new security primitive. However, if we want to address malicious applications or an application that gets compromised, we can view the operating system itself as a privilege-separating application that uses our new primitive. Other work has explored running applications in VM's for isolation, notably [23], [8], and [7] using L4Linux. However, these systems did not have our primitive at their disposal.

In our model for the OS as a privilege-separating application, each application is run in its own inferior VM. This ensures that no application runs with the full privileges of the system and furthermore no application runs with the full privileges of the user. To distinguish an inferior VM created by the operating system to run an application from an inferior VM created by an application to protect itself, we'll call the former an app VM. We refer to our operating system design as Virtics.

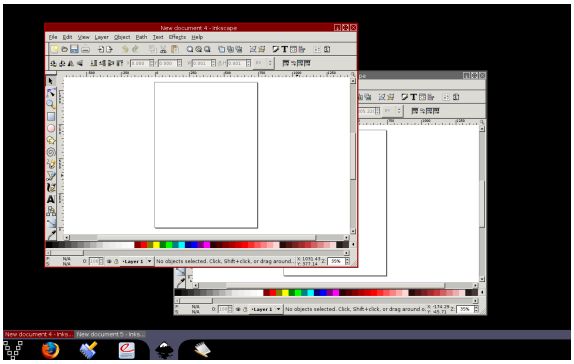

Figure 9: Screenshot of the user interface.

6.1 Screen Grants to App VM's

To implement windowing, we do the following: when an application wants to display a new window, it sends an event to the OS; if the application currently has the focus, a new window with the standard title bar, "minimize", "maximize", and "close" window decoration is created and the area below this window decoration is screen-granted to the application. Just as with the inferior VM's, the app VM is free to render whatever content it wants within this screen-granted region and can pop up windows within this region. A windowing system implemented like this has many of the same properties of prior secure GUI's described in [20] [7]. This includes:

- Keyboard input cannot be sniffed or spoofed; all input flows through the operating system, which makes sure it only comes from input devices and is only directed to the application with the current focus
- Screen content cannot be scraped; an application cannot read the content of a window that has been screen-granted to another application
- Window non-interference; an application cannot cause a window to pop up when another application has the focus; if it does so, the request will be delayed until the application gains the focus and a small visual indicator will be placed next to the application's icon in the application bar (see next bullet)
- The application that currently has the focus is clearly indicated to the user; we accomplish this

with two bars at the bottom of the screen (see Figure 9). The lower bar is called the application bar and contains one entry for each currently running application. The upper bar is called the window bar. There is one window bar per running application, and only windows associated with a given application appear in its window bar. A visual link is made between an application in the application bar and its window bar. As can be seen in Figure 9, when an application's window bar is showing, its icon is surrounded by the same color as the window bar and the two bars flow together.

6.2 Persistent Storage

6.2.1 VM Image

Unlike a typical inferior VM whose image is immutable, an app VM must be allowed to make persistent changes to its image. To accomplish this, we allow an app VM to issue WriteBlock events in addition to ReadBlock events. Because an app VM's image is not immutable, Virtics only allows one instance of each app VM to be running at any time.

6.2.2 User's Files

In addition to its own image, an app VM must be able to access the user's files. Instead of allowing an app VM to have access to all of the user's files, by default it has access to none of these files. If it wants to gain access, it must carry out the standard open/save dialog window that user's are used to. Instead of the file dialog coming from the app VM, which we do not trust, it is displayed by the operating system and made to appear as if it came from the application so that the user knows which application to associate the dialog with. This is similar to the Powerbox [19] used to grant access in other operating systems [34] [21]. When a user selects a file or folder, the following access control decisions are made:

1. If the dialog is an OpenFile(s) dialog, the application is given read and write access to each of the chosen files, modulo file extensions.

2. If the dialog is an OpenDirectory(s) dialog, the application is given read and write access to the chosen directories and read and write access to all files and directories within those directories; it is worth noting that the root directory holding all of the user's documents is not something that can be chosen; at the highest level, the user only sees the files and directories within this root directory; there is no way to navigate up to its parent and then select it; this means that giving access to all files requires a concerted effort.

3. If the dialog is a SaveFile dialog, the application is given read and write access to the chosen filename modulo file extensions.

6.3 Sound and Microphone

To output sound, an app VM must send an event containing the raw contents of the sound to output. To read from the microphone, an app VM must request access. An app VM must have the focus when it requests access and a small window asking to confirm the request appears under the cursor; if the user confirms, access to the microphone is given; this access persists when the application no longer has the focus.

6.4 Printer

To print a document, an app VM must send events specifying a document to be passed directly to the printer.

6.5 USB Storage, CD-ROM's, Network Drives

File systems exterior to the main hard drive can be made available to app VM's as mount points in the file dialog hierarchy. This allows the same access control to apply to these files as files on the main hard drive. If the file systems themselves may be malicious [32], the operating system could consider running them in their own file system inferior VM's.

6.6 Other Hardware

Gaining access to other hardware (e.g. write access to the CD-ROM for burning) could be accomplished using the same guiding principles we used to share the hardware above; that is, don't expose the hardware directly and make the application do the hard work of converting data into a raw form that can be output directly or the hard work of reading a raw form and converting it into something intelligible.

6.7 Installation

In the Virtics system, the fundamental unit is the application. There is no notion of installing a library or part of an application. A Virtics package contains exactly one application. A system base containing gigabytes of common libraries is provided on a read-only basis to all applications. A Virtics package, then, is essentially the diff between the files in the system base and the files needed to run the application. This type of self-contained packaging system has some nice properties:

1. the user knows that everything needed to run the application is contained in the package; if an application needs a library that is not in the system base or a newer or older version of a library in the system base, it must include this library in its package; thus, "dependency hell" is avoided

2. a programmer knows exactly what to expect from the software environment they are programming

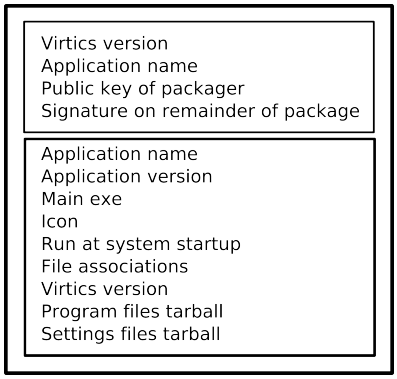

Figure 10: A Virtics package.

for; thus, if it works on their system, it'll work on the end user's

3. maintaining backwards-compatibility is easier since Virtics 2.0 can provide the system base from Virtics 1.0 to allow applications packaged for that system to be run on the newer one

In Virtics, only the user can install an application. Another application has no way of doing so. The system can distinguish between the actions of a user and the actions of an application because all user input flows through the system and there is no way for an application VM to spoof this input.

A more detailed look at a Virtics package can be found in Figure 10. The first four fields are known as the verification header. Its purpose is to let Virtics know if the package is an upgrade of an existing application already installed on the system. A package is an upgrade if it has the same name and was packaged by the same public key. Otherwise, the package represents a new application. It is important to distinguish between an upgrade and a new application (and to verify this cryptographically) because an upgrade inherits all of the access the user has given the old version (e.g. see Section 6.2.2).

Once the package is verified, the signed part of the package is used to perform the installation. For a new application, the user is presented with a simple set of (up to) four dialogs. The first displays the name of the application, its version, and its icon and asks the user to either continue or cancel. If the user decides to continue and the application has requested to be run at startup, the user is asked if they want the application to be run whenever the system starts. After making their choice and choosing to continue, the user is presented with a third dialog asking if they want to associate the application with the file types the application has requested. For each file type, the currently associated application is displayed in order for the user to make an informed decision. After making their choices and choosing to continue, the user is presented with a fourth and final dialog showing the progress of the installation. For an upgrade, the user is presented with a simple set of two dialogs. The first displays the name of the application, the old version, the new version, and the icon and asks the user to either continue or cancel. If the user decides to continue, they are presented with a dialog showing the progress of the installation. It is worth noting that for both a new install and an upgrade, if the user decides to cancel before reaching the dialog showing the progress of the installation, no changes are made to the system.

The files in the package's tarballs are installed to a private disk specific to the application. This disk is overlaid on top of the read-only system base. The application is free to make any changes it wants to its private disk, as it is the only one that will see these changes. For the purposes of installing a new application, there is no difference between the files in the program files tarball and the files in the settings files tarball; both are untarred to the private disk. For an upgrade, however, the two sets of files are treated differently: files that appear in the settings files tarball but are already on the private disk are ignored. The idea is that the upgrade should not clobber any configuration changes the user made to the old version.

6.8 File Associations

During installation, an application can request that it be associated with specific file extensions or URL protocols. If an application is granted association with a file extension, it is not given *carte blanche* access to files with that extension; rather, it simply becomes the application that is called upon when a file with that extension needs to be handled. A file needs to be handled in two circumstances:

1. the user launches a document while browsing through their documents in the documents explorer

2. another application asks for a file to be handled and the user confirms this need through a dialog

Handling a URL protocol is similar, except a URL protocol will never be encountered in the documents explorer.

If the application needed to handle a file or URL protocol is not currently running, it is started. Once started, the application is informed of the file or URL it is to handle. If the item needing handling is a file and it exists in the user's documents, the application is given access to this file. If it is a file but is located on the private disk of another application, it is transparently mapped to a temporary documents directory on the private disk of the handling application. If the item needing handling is a URL, the handling application need only be informed of what the URL is.

6.9 Running Applications

An application can only be run through an action of the user. There are four such actions:

1. explicitly selecting an application to be run
2. allowing an application to be run at system startup (see Section 6.7)
3. explicitly selecting a document to open, causing the application associated with that type of file to be run (see Section 6.8)
4. allowing one application to ask another application to handle a file of a specific type (see Section 6.8)

Each running application runs in a separate virtual machine. This virtual machine is isolated from the other applications running in their own virtual machines and from the system by the hypervisor. In order for the application to be useful to the user, however, it must be allowed some communication with the overall system. This is done with narrow protocols between the application VM and the system:

Disk a running application VM is given the ability to read and write blocks on its private disk; this is done through a driver whose frontend runs in the application VM and communicates with the corresponding backend that runs in the system; an application can also work with the user's documents by communicating with the documents server running in the system; this server does not provide *carte blanche* access to the user's documents, but rather tightly controlled access based on the application making the request (Section 6.2.2)

Network an application VM is given the ability to read and write packets on its private network adapter; this is done through a driver whose frontend runs in the application VM and communicates with the corresponding backend that runs in the system; the network adapter is given a local IP address and is NAT'ed to multiplex it over the real connection; a small amount of filtering is done to ensure that the IP address on packets coming out of the private network adapter is the same as that assigned for its use and to ensure that an application VM cannot directly send packets to the local IP of another application VM

Screen an application VM is responsible for all of the rendering that must be done; in the end, the system only cares about bitmaps representing the contents of windows; the application VM informs the system of the memory location of these bitmaps and the system simply pushes the raw pixel contents to the screen; informing the system of the bitmaps as well as negotiating the placement of windows and receiving user input takes place through the virtual window manager server (Section 7.1.4)

7 OS as a Privilege-Separating Application Implementation

Figure 11 gives an overview of the interactions between an app VM and the system in our prototype. Most of these interactions follow directly from the fact that an app VM is an inferior VM. The virtual window manager client/server interaction was described in Section 3.2.3. The disk front/back driver is the virtio_blk driver in the Linux kernel, providing block-level access to the VM image. The net front/back driver is the virtio_net driver in the Linux kernel, providing packet-level network access. The network adapter has a local IP address and NAT is used to multiplex onto the real connection.

Unique to app VM's is the ability to access the user's files. This takes place through the documents client and documents server. Together they act as a network file system between the app VM's and the system. The documents server maintains all the prior access control decisions of the user and only allows access to the appropriate app VM's. The documents client is implemented as a FUSE file system that communicates with the documents server. For the user's files that an app VM already has access to, they will automatically appear under a mount point on the app VM's private disk and will be accessible to the app VM without interaction with the user. For the user's files that an app VM doesn't have access to, they will not appear under the mount point and it will be as if they didn't exist. To gain access to these files, the application must initiate a system-mediated file dialog. This is implemented as a patch to common widget libraries such as GTK and Qt. The patch transparently turns a request to display a file dialog locally into a request to the documents server to display a file dialog on behalf of the application. Since the documents server owns this window and not the application, it can observe the user's choices and update the access control on files accordingly. These files will then appear under a mount point on the app VM's private disk. Since the vast majority of graphical applications rely on popular widget libraries to display file dialogs, we are able to modify a few libraries while leaving application code untouched. The size (in terms of lines of code) of various components in the prototype are listed in Figure 12.

Since our prototype provides a standard Linux and X Window stack to applications running in an app VM, it can run many popular applications without modification. This includes (but is not limited to), Firefox, Open Office, Mplayer, Evince, The Gimp, Evolution, Pidgin, Gedit, Kate, and Inkscape.

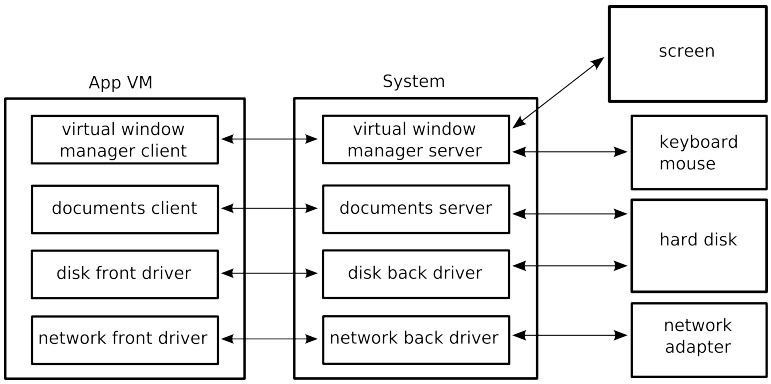

Figure 11: Interactions between an app VM and the system

installation client	2700	installation server	4600
run client	1500	run server	9000
vwm client	5700	vwm server	9800
		virtics ui	14000
documents client	8300	documents server	14000
file dialog client	3200	file dialog server	8000
	21200		59400

Figure 12: Lines of C code in the OS implementation as counted by cloc.

7.1 Virtics Clients and Servers

Most of Virtics is implemented as client and server processes that run in userspace. The clients usually run inside the application VM's while the servers run in the privileged system domain, although sometimes the servers running in the privileged system domain can be clients of each other.

7.1.1 Application Database Server

An application database server runs in the system domain and keeps track of all the installed applications and inferiors and their properties.

7.1.2 Installation Client and Server

The installation server runs in the system domain and has two main jobs: the installation of applications and the installation of inferiors. It is worth noting that a Virtics package is not parsed inside the system domain in order to minimize the attack surface for this piece of untrusted data. The most important step in installing the package is signature verification. To do this, a separate signature-checking VM is created with access only to the package to install. The sole goal of this VM is to read the name and public key from the verification header and verify that the signature on the package is correct. Although this operation runs in a separate VM, it is important for the security of the system that it not be compromised. Otherwise, a package could pretend to be the upgrade of an existing installed application and gain access to its private disk and any files the application may have been given access to. Once signature verification is complete, a different VM is created with access to the application's private disk. It is the job of this VM to parse the rest of the package and populate the private disk. It is worth noting that this VM is completely untrusted and corruption of its operation (for example, providing a malicious tar file) would not really gain the attacker anything. The second job of the installation server is installing inferiors. This job is considerably easier since we know which application the request comes from and do not have to perform any signature checking. In this case, we proceed directly to the second step of launching a VM to populate the private disk.

7.1.3 Run Client and Server

The run server runs in the system domain and is responsible for creating and keeping track of all application and inferior VM's. The Virtics UI can make requests to start applications. Application VM's can request that inferiors be run or that files be handled.

7.1.4 Virtual Window Manager Client and Server

Each application and inferior VM runs its own copy of the X server. Inside these VM's, a program known as the virtual window manager client acts as a window manager for the dummy X server. This client listens for window events (creations, configures, mappings, property changes, etc.) and passes them on to the virtual window manager server running in the system VM. The client utilizes the Composite and Damage X extensions to learn when the the contents of windows change and conveys this information to the server. Because we want all application VM to system protocols to be as simple as possible for security purposes, communicating the contents of a window is a matter of specifying memory pages. These memory pages are mapped by the virtual

window manager server and are considered to represent a raw rectangular texture. This texture is mapped onto the appropriate window using OpenGL. If the client has specified garbage as the content of the window, garbage will appear as the content of the window; we place no interpretation on this data.

7.1.5 Virtics UI

The virtual window manager server does not itself display the UI of Virtics. Instead, it makes requests to map windows on behalf of virtual window manager clients, and it is the Virtics UI that decides whether these windows get mapped and where. These decisions are made based on which application currently has the focus. The virtual window manager server labels the windows it creates, and this allows the Virtics UI to place them appropriately in the window bar/application bar hierarchy shown previously in Figure 9.

7.1.6 Documents Client and Server

Files outside of an application's private disk (i.e. the user's documents) are accessed from the documents client, which is a FUSE file system running inside the application VM that forwards operations on to the documents server, running in the system domain. The documents server makes a decision to carry out the operation or deny it based on the permissions it has for the application making the request. The documents server itself overlays these permissions on a normal Linux file system.

7.1.7 Documents Dialog Client and Server

To change its permissions on the user's documents, an application sends a request to the documents dialog server. This server displays an open/save dialog that appears to come from the requesting application. Once the user makes their choice, the documents dialog server informs the documents server to change its permissions and informs the documents dialog client of the choice. In order to achieve backwards compatibility with existing applications, the documents dialog client is implemented as a patch to common widget libraries such as GTK and Qt.

8 Evaluation

8.1 Performance

Most of the development effort for virtual machines has been focused on providing highly-efficient CPU utilization, memory access, disk I/O, and network I/O [2]. Because of this effort, the impact of virtualization on these resources is minimal. Thus, in this section, we examine those aspects of performance that are more unique to our use of virtualization, namely VM startup time, memory overhead of using a VM, and graphics performance.

	cold start		warm start	
	non-VM	VM	non-VM	VM
firefox	3	5	2	2
openoffice	5	5	2	4
gcalc	2	3	1	1
evolution	3	5	2	3
gedit	2	3	1	2
kmplayer	2	4	1	3
kate	3	5	1	3
gimp	2	3	1	2
gaim	2	3	1	2
inkscape	3	4	1	3

Figure 13: Startup times in seconds (rounded up to the nearest second). The VM times are for applications starting in a VM using pre-execution. Non-VM times are for applications started as native OS processes. Cold start times are computed after caches have been cleared, and warm start times are computed after a previously successful start has already warmed the cache. The variation in these times across multiple runs was minimal.

8.1.1 Startup Time

One of the challenges of using a VM as a fundamental primitive is that booting a VM and running code in it takes significantly longer than forking and executing the same code in a process. Starting an application in a VM booting an unoptimized kernel from scratch takes about 20 seconds, while starting an application as a process takes at most a few seconds.

Clearly, this is an unacceptable wait time for even the most patient of users. By optimizing the configuration of the kernel, stripping out unused modules and drivers, changing the start order of the remaining modules, and reducing the number of started processes, other researchers have managed to reduce this time to 5 seconds [11]. However, this boot time is still greater than we want the entire boot and application load process to take.

The solution Virtics uses is to pre-execute VM's to the point of application launch and then suspend the VM, creating an image file that is typically less than 64MB. Then, when Virtics needs to start an application, it simply loads the pre-executed image, resumes it, and loads the application code immediately. This technique was used in [24].

Our measurements show that creating an inferior VM and jumping into the custom code takes approximately 800 milliseconds. For cases where this amount of time is prohibitive, an application could create inferiors before they are needed.

We also performed measurements when our OS is acting as a privilege-separating application using our new primitive. We measured the times to start an application using a pre-executed VM versus the same applica-

tion running outside a VM as a native OS process (see Figure 13). All of the Virtics times are within a second or two of the native ones.

8.1.2 Memory Overhead of Running in a VM

Running code in a VM instead of a process means that a large amount of code and data on the system that would otherwise be shared with other code is now duplicated inside each VM. Virtics mitigates the memory overhead of this duplication by utilizing the Linux KVM beta implementation of content-based page sharing. Content-based page sharing examines the contents of individual pages (as opposed to the way sharing is typically done through mapping the same file) and coalesces identical pages both within a VM and across VMs [25]. Virtics only shares zero pages using this technique, since otherwise a malicious VM could combine a brute force search with careful timing measurements to determine if other VM's on the system have pages with the same data. Using zero page sharing, Virtics reduces the size of an empty VM with its own private kernel, its own private X server, and its own private copy of support processes, down to about 48MB.

To illustrate the minimal memory overhead, we simultaneously opened the Alexa Top 40 websites in 40 instances of WebKit that were native OS processes and 40 instances of WebKit that were inferior VM's. The WebKits running inside inferior VM's used 48 more MB of RAM per instance than the WebKits running as native OS processes.

It is worth noting that the memory overhead could be reduced further by using sub-page level sharing [9]. And it likely could be reduced much further by using a copy-on-write delta virtualization scheme as described in [24]. This scheme does not suffer from the same zero page limitation as content-based page sharing since it would only share inferior VM pages in their initial state, not after they were written with sensitive data. Copy-on-write delta virtualization has been implemented in a general manner for Xen [22]; in that work, up to 70% of memory was shared when child VM's involved web browsing and PDF viewing.

8.1.3 Graphics Performance

Virtics copies rendered window contents from VM's onto the screen, imposing a penalty on graphics performance. To evaluate the impact of copying, we tested the performance of streaming video using the Flash plug-in in Firefox and the kmplayer application. We were able to watch a full screen (1440 x 900) movie at a normal frame rate with no noticeable difference between the applications running as native OS processes and running as VM's.

We also measured sdlquake running at 640x480. The program's internal *timedemo demo1* command measured 168 frames per second when running quake as a native OS process and 128 frames per second when running it in an app VM.

8.2 Usability

Although we have not conducted a formal usability study, we have put a large amount of thought and effort into designing a usable system. A user of our system would very likely have no idea that they are using multiple virtual machines. From their perspective, the system is just a set of applications and documents, which are concepts we think a user understands.

File sharing is a great example of how we approached the problem of security and usability. Obviously, if a user couldn't access files they created with one application from another application, the fact that they were using a VM would be apparent and would be a barrier to usability. Instead, we leverage file open/save dialogs to provide security and usability. They provide security because by default, applications don't have access to the user's documents and must gain access through dialogs. They provide usability because file open/save dialogs are a common paradigm in GUI applications, so the user will be familiar with them, and the user doesn't have to do any extra work of answering an annoying security popup.

Our windowing system reuses the familiar concepts of a window bar and window focus. We introduce a new GUI element known as the application bar, with one entry for each running application, but each application is represented by its icon; the icon should be familiar to the user because it is the same icon they used when they installed the application.

Although our armored application implementations are not production quality commercial applications, there is strong evidence that armored applications can be made just as usable as their unarmored counterparts. This comes both from our own design experiences, which could be implemented, and similar applications implemented by other people. For example, the Chrome [3] team has done a great job of creating a browser that uses privilege separation but that does not provide a subpar user experience.

8.3 Security Analysis

The security approach of Virtics is to isolate all applications and documents *a priori* in case they are malicious and to maintain this isolation throughout their lifetime, only granting small amounts of permission when the system observes the user taking certain actions. The classes of attacks prevented and not prevented by our approach are listed below, followed by a discussion of the virtual machine interface as a security interface.

8.3.1 Attacks Prevented

In general, the class of attacks we prevent results from our ability to separate the privileges of the user from the privileges of a single application, and then to further separate the privileges of an application from the privileges of a document it is working on. A taxonomy of these attacks follows:

- *Document exploit resulting in application compromise* Example: Malformed HTML resulting in exploit of HTML renderer, resulting in compromise of entire browser.
- *Application compromise resulting in total user compromise* Example: PDF exploit resulting in exploiting PDF application, resulting in access to all files/programs of the user.
- *Spyware/Trojan Horse compromising privacy of user* Example: Game that gets installed, pretending to be a harmless application, then proceeds to hook all keyboard events.
- *Spyware/Trojan Horse resulting in total user compromise* Example: Music player that gets installed, pretending to be a harmless application, then proceeds to access all files/programs of the user.

8.3.2 Attacks Not Prevented

The class of attacks not prevented includes attacks on the mechanisms that enforce privilege separation. It also includes attacks that result from allowing code in VM's to carry out arbitrary computation and network communication. A taxonomy of these attacks follows:

- *Attacks against the hardware* Example: Using a cache race condition to write an arbitrary value to a cache line.
- *Attacks against the hypervisor* Example: Tricking the hypervisor into mapping a page into an application that the application does not own.
- *Attacks against the application part of an application/inferior separated application*. Example: Malicious website running in an inferior that asks the application part to open a malicious URL for which it mishandles the parsing.
- *Intra-inferior security* Example: Enforcement of the same origin policy among the frames in a single tab of a web browser.
- *Spyware/Trojan Horse tricking user* Example: Picture application that gets installed, pretending to be a harmless application, and then tricks the user into opening sensitive files with it or typing sensitive information into it.
- *Joining a Botnet* Example: Weather application that gets installed and then acts as part of a botnet, accepting commands from a remote attacker and carrying out attacks on other machines.
- *Denial of Service* Example: Chat application that gets installed and then proceeds to monopolize the CPU.

We discussed the security of individual armored applications in Sections 4.1.1, 4.2.8, 4.3.2, 4.4.1, and 4.5.1. In general, these security analyses boil down to attacks on the application/inferior protocol, which we aim to keep as simple as possible, and attacks on the virtual machine environment. The argument is not that the virtual machine environment is perfect. Indeed, attacks against virtual machines have been carried out in the past [15]. CPU errata are also a concern since we allow inferior VM's to run arbitrary code. The argument is that VM's provide the simplest interface for easy backwards-compatibility with legacy code and that if we can reduce the security of large amounts of code down to the security of virtual machines, then a large amount of effort can go into scrutinizing and verifying them. Some of the foremost experts in attacking systems at the virtual machine layer [31, 29, 30] have recently stated that systems using virtual machines for isolation are our best option for a secure desktop, and in recent work [18], they have gone a long way toward making the interface between the system and untrusted code running in VM's very narrow. Other recent work [28] has looked at assuring the control-flow integrity of a running hypervisor; they have developed techniques that work on commodity hypervisors, such as Xen [2].

9 Limitations

9.1 Applications in a VM

Some classes of applications are not well-suited to running in an app VM. Applications that want to easily scan the entire hard disk (e.g. desktop search) are not well-suited. Applications that rely on reading the content of other application's windows (e.g. a screenshot application) are not well-suited. Applications that rely on hardware-accelerated 3-D graphics are not well-suited since an app VM has no access to the 3-D hardware. However, if 3D developers return to software-based rendering on massively multicore desktops, this particular limitation would go away.

9.2 Reliance on the User

When running the OS as a privilege separating application, we allow the user to install any application and give any application access to any file. If a user installs a malicious application, we attempt to protect the system integrity and all of the user's files by default, but no attempt

is made to prevent the user from typing sensitive information directly into the malicious application or opening sensitive files with it. Similarly, with a benevolent application, we do not prevent the user from telling it to open a malicious document, but we do provide the ability for an application to protect itself using inferiors.

9.3 Semi-Local Adversary

Right now, we assume the adversary is remote across the Internet, but we'd also like to deal with a semi-local adversary; that is, an adversary on the same physical network as us. With a remote adversary across the network, they are limited to IP and higher network stack attacks. An adversary on the local network could carry out physical layer attacks, trying to compromise the system's network driver. Future Directed I/O technology could allow us to assign the network card and its driver to a separate VM, potentially mitigating this type of attack.

10 Conclusion

Now is the time to deploy desktop operating environments that can protect applications and users from malicious applications, documents, and network flows. In this work, we introduce Virtics as such a system. We demonstrate that replacing traditional process-based application isolation with high-performance virtualization-based isolation, where every user application and document executes in a separate virtual machine can be done in an efficient manner.

Virtics has been in daily use for over a year and it supports both unmodified applications and applications that provide internal isolation of documents. Performance using applications running in Virtics is comparable to that of native OS process-based applications, and the startup time and memory overheads are not unreasonable given the gain in security.

One direction for future work would be applying our design to other operating systems, such as Microsoft Windows. There is nothing fundamental that limits the concepts we have developed to one particular operating system or another.

References

[1] BALLE, J. 1.91% of all PCs are fully patched!, Dec. 2008. http://secunia.com/blog/37/.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 164–177.

[3] BARTH, A., JACKSON, C., REIS, C., AND TEAM, T. G. C. The security architecture of the Chromium browser, Dec. 2008. http:crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[4] BRUMLEY, D., AND SONG, D. Privtrans: automatically partitioning programs for privilege separation. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 5–5.

[5] COX, R. S., HANSEN, J. G., GRIBBLE, S. D., AND LEVY, H. M. A safety-oriented platform for web applications. In *In IEEE Symposium on Security and Privacy* (2006).

[6] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the Symposium on Operating Systems Design and Implementation* (2008).

[7] FESKE, N., AND HELMUTH, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 85–94.

[8] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)* (October 2003).

[9] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of OSDI* (2008).

[10] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference* (2001).

[11] MARTI, D. LPC: Booting Linux in five seconds, Sept. 2008. http://lwn.net/Articles/299483/.

[12] MCMILLAN, R. Security industry faces attacks it cannot stop, Mar. 2010. http://www.itworld.com/security/100320/security-industry-faces-attacks-it-cannot-stop.

[13] NATIONAL VULNERABILITY DATABASE. CVE-2008-1693. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1693.

[14] NATIONAL VULNERABILITY DATABASE. CVE-2010-1349. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-1349.

[15] ORMANDY, T. An empirical study into the security exposure to hosts of hostile virtualized environments.

[16] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. *12th USENIX Security Symposium* (August 2003), 11.

[17] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems* (New York, NY, USA, 2009), ACM, pp. 219–232.

[18] RUTKOWSKA, J., AND WOJTCZUK, R. Qubes OS architecture, Jan. 2010. http://qubes-os.org/files/doc/arch-spec-0.3.pdf.

[19] SEABORN, M. The powerbox: a GUI for granting authority, Mar. 2009. plash.beasts.org/powerbox.html.

[20] SHAPIRO, J. S., VANDERBURGH, J., NORTHUP, E., AND CHIZMADIA, D. Design of the EROS trusted window system. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 12–12.

[21] STIEGLER, M., KARP, A. H., PING YEE, K., AND MILLER, M. Polaris: Virus safe computing for Windows XP. Tech. rep., HP, 2004.

[22] SUN, Y., LUO, Y., WANG, X., WANG, Z., ZHANG, B., CHEN, H., AND LI, X. Fast live cloning of virtual machine based on xen. In *11th IEEE International Conference on High Performance Computing and Communications* (2009).

[23] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: making trust between applications and operating systems configurable. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 20–20.

[24] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Oper. Syst. Rev. 39*, 5 (2005), 148–162.

[25] WALDSPURGER, C. Memory resource management in VMware ESX server. In *Fifth Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[26] WALSH, D. Introducing SELinux sandbox. In *Linux Plumbers Conference* (2009).

[27] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium* (Montreal, Canada, August 2009).

[28] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[29] WOJTCZUK, R. Adventures with a certain Xen vulnerability, Oct. 2008. `http://invisiblethingslab.com/resources/misc08/xenfb-adventures-10.pdf`.

[30] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking SMM memory via Intel CPU cache poisoning, Mar. 2009. `http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf`.

[31] WOJTCZUK, R., RUTKOWSKA, J., AND TERESHKIN, A. Xen 0wning trilogy. In *Black Hat USA* (2008).

[32] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 243–257.

[33] YEE, B., SEHR, D., DARDYK, G., CHEN, B., MUTH, R., AND ORMANDY, T. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy* (2009).

[34] YEE, K.-P. User interaction design for secure systems. In *In Proceedings of the 4th International Conference on Infor mation and Communications Security* (2002), Springer-Verlag, pp. 278–290.