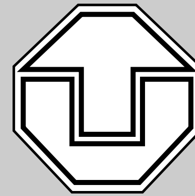
TECHNISCHE UNIVERSITÄT DRESDEN



Fakultät Informatik

Technische Berichte
Technical Reports

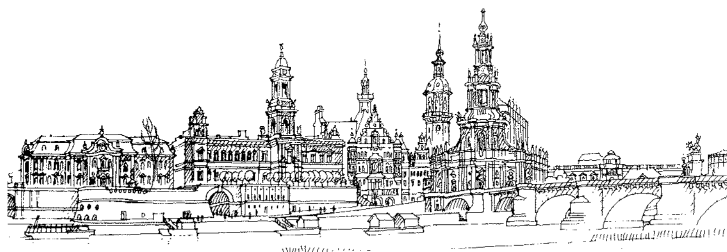
ISSN 1430-211X

TUD–FI04–02–März 2004

Norman Feske and Christian Helmuth

Institute for System Architecture, Operating Systems Group

Overlay Window Management: User interaction with multiple security domains



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

URL: http://www.inf.tu-dresden.de/

Overlay Window Management: User interaction with multiple security domains

Norman Feske and Christian Helmuth

Abstract

Graphical user interfaces for high-assurance systems must fulfill a range of security requirements such as protected and reliable presentation, prevention of unauthorized cross-domain talk, and prevention of user-input eavesdropping. Additionally, it is desirable to support legacy applications running in confined compartments. Standard isolation methods such as virtual-machine monitors provide one frame buffer per security domain, where each frame buffer is managed by one legacy window system. This raises the question of how to safely integrate multiple (legacy) window systems and protect the displayed data while preserving the usability of modern user interfaces.

Our paper describes the Overlay Window System, a general mechanism for multiplexing windows of multiple distinct window systems into the host frame buffer. Thus, each legacy window appears to the user as one corresponding host window that can be moved and resized. To achieve this, only slight modifications of the legacy window system are required whereby, the source code does not have to be available. Our implementation of an Overlay Window System successfully multiplexes Linux, GEM and native L4 applications.

1 Introduction

The ever-increasing number of security-sensitive platforms connected to untrusted public networks raises a high demand of executing trusted and untrusted applications side-by-side on one device. For example, people store and process private data on PDAs using trusted software and execute gadgets—downloaded from untrusted sources—which are potentially malicious (malware). There exist plenty of options to deal with such scenarios, which fall into two categories:

Firstly, there are operating systems that provide strong application-level isolation and protection. Prominent examples are the L4 family [18] and EROS [19]—a capability-based operating system. They provide a custom infrastructure to build applications, which in turn are especially designed for these operating systems. For example, with DOpE [11, 12] and EWS [20] there exist custom trusted window systems for these operating systems. The common drawback of these solutions is the lack of available applications.

Secondly, with virtual machine monitors and sandboxing techniques, there exist powerful solutions to partition one physical computer system into multiple protection domains where each domain can be an entirely different operating system. Härtig [15] presents an overview of such architectures and emphasizes the trend of wide application of these techniques in system design. Virtual machines lead to an unmatched flexibility and availability of user applications. As described in [14], the latests efforts with secure booting techniques—namely TCPA [6]—pushed the application area of these approaches even further. Even though the known approaches are technically different, they have one drawback in common that shrinks the application area significantly. There exists virtually no model for secure user interaction with multiple security domains.

In this paper, we present a generic solution for the addressed problem and discuss application scenarios of deploying virtualization and sandboxing techniques combined with our user-interaction model.

Currently available virtual machines perform user interaction based on consoles. Traditionally, there exist two interfaces for the interaction with the user: input devices and a frame buffer display. This paradigm appears antiquated regarding the modern way of user interaction with computer systems. Today, we expect to interact with windowed applications that can freely be arranged on the screen.

Existing virtual machines do not to deal with such applications but leave the window management to the virtualized legacy operating system. This way, the virtual frame buffer, containing all windows of the legacy operating system, is presented to the user as one host window or as full screen (Figure 1).

Once multiple instances of virtual machines come into play, as described in the numerous application scenarios in [13], the problem of multiplexing virtual frame buffers immediately arises and the full-screen approach becomes unfeasible.

We want to integrate multiple legacy window systems into one user environment. Thus, we approach the following problems:

- The way of how windows are managed and how redraw operations are performed differs among legacy operating systems.
- We want to integrate proprietary legacy operating systems of which the source code is not available.
- We cannot access the internal data structures that represent the user interface on the legacy operating system.

- All properties of used virtualization and sandboxing techniques in regard to isolation and protection of security domains must persist.

In this paper, we present a solution for the problems mentioned above. We can integrate any number of legacy window systems running on different virtual machines. The legacy window systems can even use virtual frame buffers of different sizes and color depths. This can be achieved with virtually no (or marginal) modifications of the legacy operating systems. As a proof-of-concept, we integrated three entirely different window systems—namely X11, GEM and DOpE—into one user environment while running them inside fully isolated security domains. For this, we required no access to the GEM source code.

The rest of the paper is structured as follows: In Section 2 we describe our basic mechanism to multiplex window systems. It is followed by Section 3 that describes the actual implementation of our mechanism. Section 4 highlights new application fields, which can be captured by virtual machines combined with our technique. In Section 5 we give an overview about related work and put our work in the context of trusted window systems. Section 6 concludes the paper with an outlook to future work.

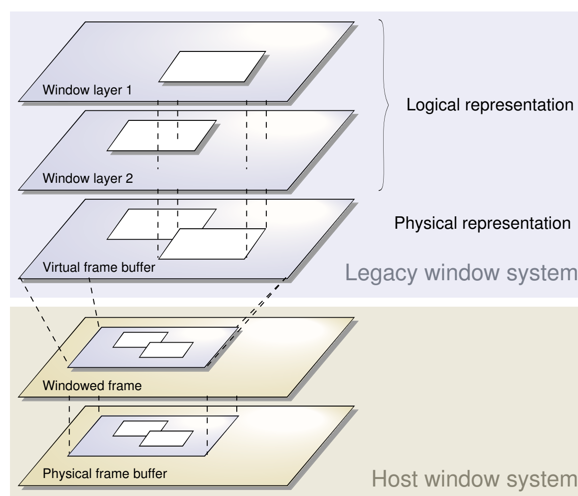


Figure 1: Virtual frame buffer displayed as one host window. Legacy window information are only exported as virtual frame buffer and the virtual machine displays the entire frame as one host window. Windows are displayed with shadows at the logical representation level to highlight the fact, that the semantics are known not only the pixel data.

2 Mechanism

2.1 Nested-window-systems approach

Current implementations of virtual machines virtualize (or emulate) standard hardware devices to enable guest operating systems to perform I/O operations. This way, legacy operating systems reuse existing device drivers to access virtual input devices or graphics cards, and thus, the host system.

The common technique for emulating graphics cards is to provide a virtual frame buffer to the guest. The legacy window system renders its private window stack into the virtual frame buffer (Figure 1). Thereby, the window system translates its logical representation (e. g., window lists and window-decoration configuration) into the physical representation of pixels in the frame buffer. The physical representation of the legacy system contains no semantics about the displayed information anymore, so that the logical representation is not known to the host window system. Therefore, established virtual machines render the virtual frame buffer into one big host window.

This approach has several usability drawbacks. Firstly, the virtual-machine window is unhandy due to its size and pollutes the screen. Secondly, all legacy windows are on the same stacking level regarding host window order. This highly restricts the flexibility of application-window placement. Interaction with such nested window systems is neither natural nor efficient.

2.2 Overlay window system—hosting legacy windows

The above-mentioned limitations are not caused by the utilization of the virtual frame buffer but by the loss of important information on the way from the legacy window system to the host representation. Since the existing interfaces—frame buffer and input devices—alone are insufficient, we looked for novel ways to obtain the semantics of legacy windows. An ideal (i. e. cooperative) legacy window system for our purposes would provide additional window information besides the virtual frame buffer. Thus, a host window system is able to manage legacy and host windows similarly, as it would know about the semantics of legacy windows. This approach is comparable to remote GUI protocols (e. g., X11 or RDP), but needs fewer high-level commands.

We have to tackle the problem of how to export legacy window contents, positions, and sizes. Furthermore, the host window system needs to know the legacy window stacking order to present a consistent global window state to the user. Figure 2 illustrates this idea.

The generic export of window state information from arbitrary legacy window systems is tricky, as the used data structures differ significantly. Keeping the two window stacks—legacy and host—consistent on the basis of totally different structures appears to be infeasible. On the other hand, window-state *changes* (e. g., movement or resizing) are very easy to export. The host window system can use them to reconstruct the legacy window configuration on the host side.

We call such a window system *Overlay Window System* as it overlays distinct window stacks and thus, legacy and host windows. The Overlay Window System tracks *top* (come to front), *move*, *resize*, *open*, and *close* window-state-change events, it receives from the legacy system. This way, the Overlay Window System is at all times

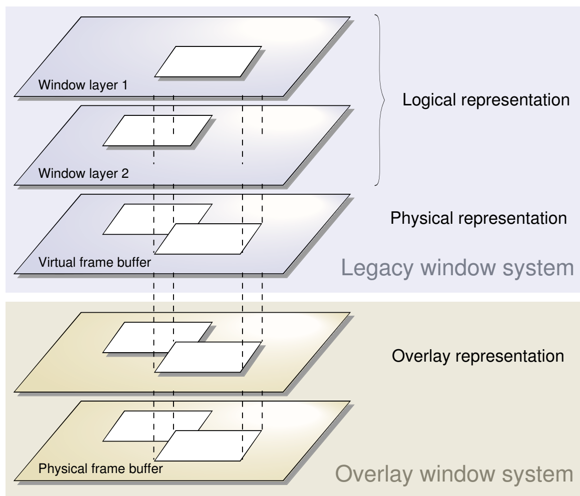


Figure 2: Overlay window system hosting legacy windows. The legacy window system provides window information to the Overlay Window System. The host presents legacy windows as separate host windows (marked by shadows in the overlay representation) to the user.

in a consistent state with the legacy window system and may cover all possible legacy window configurations. We implemented a simple state machine/tracker to map each legacy window to a corresponding overlay window.

In contrast to remote protocols with high-level functions, the Overlay Window System utilizes the physical representation—virtual frame buffer—to access the actual window contents and is thus more general than these protocols. The frame-buffer representation is sufficient and it is not required to know the complete contents of each single window.

Virtual machines provide two low-level interfaces—input devices and frame buffer—that enable the legacy system to communicate with the host. The Overlay Window System adds the high-level abstraction of a *view*. A view is a rectangular region of the frame buffer plus a stream of window events corresponding to one legacy window. For this purpose, *hooks* inside the legacy windowing software have to be identified and exploited via *hooked functions*. Hooks are instructions that provide interfaces for future expansion. The hooked function exports a state change to the Overlay Window System via the view, when the configuration of a legacy window changes. The view may be a virtual device or may utilize another mechanism of the virtualization platform (e. g., microkernel IPC).

The Overlay Window System receives all input events from the user. It propagates only input events referring to the inside of windows to the legacy system. In this case, the input events are injected into the virtual input device of the virtual machine.

User inputs referring the window controls cause state changes of the windows (e. g., resize) and are a higher-level problem. The Overlay Window System informs the hooked function installed at the legacy window system about these state changes via the view abstraction. In turn, the hooked function initiates reconfiguration of the appropriate window. Thus, the state of the view remains consistent.

We assume the legacy window system keeps its window stack at all time locally consistent—local windows may overlap invisible regions—and the virtual frame buffer contains the appropriate physical representation. Other host windows displayed may overlap existing legacy windows, but cannot reveal locally-overlapped (i. e., invisible) legacy regions. Thereby, the Overlay Window System can display a consistent window configuration at all time utilizing the virtual frame buffer and the view abstraction.

2.3 Hosting multiple legacy systems

Since the consistency property holds for multiple legacy systems too, the approach described above is sufficient to multiplex several legacy window stacks in an elegant way (Figure 3). We exploit this fact to host two or more legacy systems (e. g., operating systems for multiple security levels) sharing one overlay screen. The distinct legacy window systems can even use different screen resolutions and color depths. In this case, the Overlay Window System scales the physical representation.

Legacy systems never interfere regarding the overlay screen, because the Overlay Window System strongly separates different clients/systems. The implementation of the display policy inside the Overlay Window System permits configurations tailored to user or security demands, for example, that no window of legacy system *A* overlaps the top window of *B*. Discussion of security policies is out of the scope of this paper.

3 Implementation

We implemented our concept to proof its application in real scenarios. The Overlay-Window-System mechanism in not tied to any particular operating system and thus, it could easily be implemented on top of the Linux operating system with its large infrastructure. However, our argumentation refers to trusted systems. Therefore, the minimal complexity of the trusted computing base is our major design criteria. We consider the monolithic Linux kernel including its device drivers as far too complex to be part of our trusted computing base.

A revision of our actual requirements led us to fairly basic architectural demands:

- The host kernel must enforce isolation between multiple protection domains but needs to enable monitored communication between these domains.
- On top of the host kernel, there must exist a trusted window system with native drivers for at least input devices and the physical frame buffer.
- A sandbox for the safe execution of a legacy operating system is needed. This could be achieved with a virtual machine monitor, an emulator, or a ported OS personality.

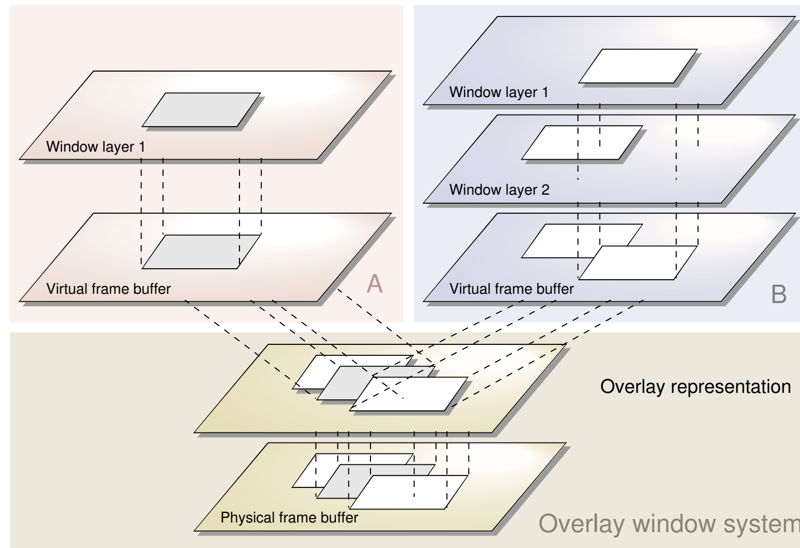


Figure 3: Overlay Window System hosting legacy windows of multiple legacy operating systems. The overlaid representation is at all times consistent with all legacy systems.

3.1 Architectural playground

With the L4/Fiasco [18, 17] microkernel, there exists a kernel that provides exactly the needed functionality of enforcing protection domains and performing safe inter-protection-domain communication. Furthermore, it provides some additional features such as real-time capability and shared memory, which will pay off in our concrete application scenarios described in Section 4. L4/Fiasco is implemented with only 15,000 lines of code and runs on x86 PCs.

With L^4Linux [16], we identified a grateful victim to act as a sandbox. L^4Linux is an user-level port of the Linux kernel on top of L4. The L4 kernel revokes all privileges to access host devices from L^4Linux. Multiple instances of L^4Linux can be started inside fully isolated protection domains running unmodified Linux programs, for example the X window system.

Right on top of the L4/Fiasco microkernel, we run a stripped-down version of DOpE as the trusted window system. Containing only 7,000 lines of C code, it features powerful mechanisms to scale, display and synchronize pixel buffers with client applications. It enforces separation of client applications, which can only receive user input events refering to a window of the actual application. As an additional candy, it is able to provide the real-time capabilities of L4/Fiasco at the user-interface level [11].

3.2 Integrating XFree86 with DOpE

As described in Section 2, the Overlay Window System relies on a virtual frame buffer, input devices, and views to integrate a legacy window system. Fortunately, the XFree86 [9] X window system provides clearly documented hooks for these interfaces:

- XFree86 provides a custom driver infrastructure to access graphics cards and plain frame buffers. Thus, we were able to export the output of X via a custom virtual display driver, whose implementation was a straightforward task—thanks to the shadowfb module of XFree86.

- Input events can be passed to the X server via the input-driver interface of XFree86.

- In X, the arrangement of windows is handled by a window manager. We used a slightly modified version of AEWM [1] to propagate window events from X to the Overlay Window System and vice versa.

The hooks for these interfaces belong to one and the same instance of L^4Linux but are implemented in different processes (X server and Window Manager). Therefore, they cannot directly speak to DOpE because DOpE would consider them as distinct client applications and thus, would isolate them from each other.

With the Overlay Mediator, we introduce a new component that acts as one DOpE client application while providing three distinct interfaces—namely Screen, Input and View—to one sandbox. For each sandbox window, it creates a DOpE window that displays its corresponding part of the virtual frame buffer and keeps its position, size and stacking order consistent with the associated window of the sandboxed window system. The Overlay Mediator forwards all input events applied to one of its DOpE windows to the sandbox via the Input interface. Figure 4 illustrates the relationship between DOpE, the Overlay Mediator, and XFree86.

This way, we successfully integrated XFree86 windows into DOpE by only using existing interfaces of XFree86. We did not need to change L^4Linux, our legacy operating system, at all.

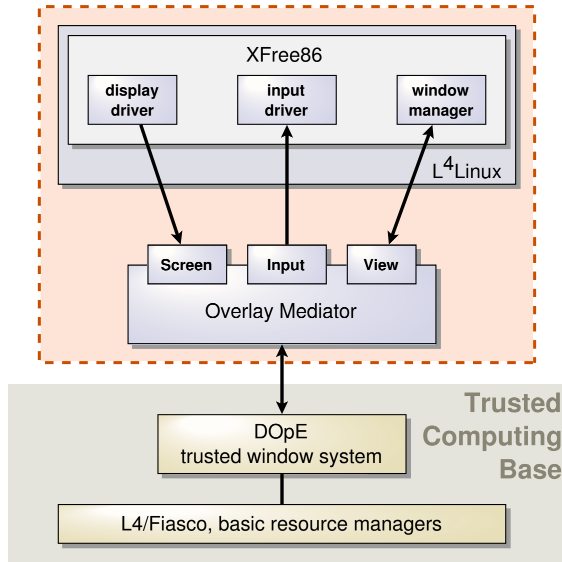


Figure 4: Relationship between DOpE, the Overlay Mediator and XFree86

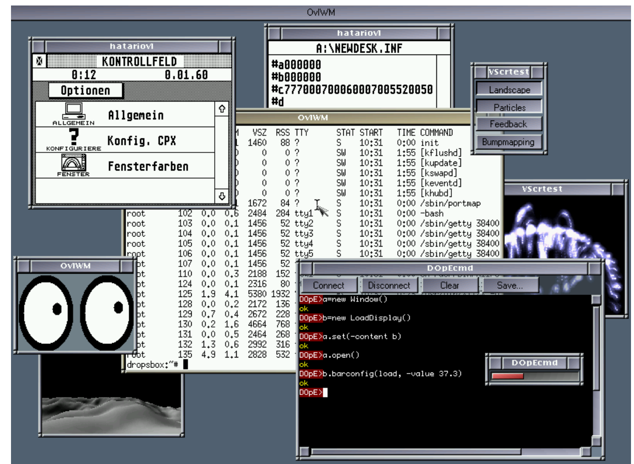


Figure 5: GEM and X11 window systems integrated into DOpE. The sessions running are Linux with Xeyes (left) and Xterm (center), GEM (upper left and top), and native L4 real-time applications (right and lower left).

As soon as a legacy window system provides interfaces for screen drivers, input drivers, and window management, it can be integrated into an Overlay Window System with moderate effort. This is the case for almost all modern window systems, for example, the graphical user interface of the Microsoft Windows operating system.

3.3 A harder nut to crack: Atari GEM

With a modicum of effort, even legacy window systems that lack clean interfaces for input devices, display devices and window management can be convinced to cooperate with an Overlay Window System.

We picked out one of the earliest window-based graphical user interfaces—namely GEM—to demonstrate the universality of our approach. For running GEM inside a sandbox, we ported the Atari ST emulator Hatari [3] to the L4/Fiasco platform. Hatari emulates all hardware components of an Atari ST including display, mouse, and keyboard; Hatari itself uses libSDL [4] as its hardware-abstraction layer. We provided a custom version of libSDL that uses the Screen and Input interfaces of a dedicated Overlay Mediator as its backend. For passing window events in and out of the Hatari sandbox, we enhanced Hatari by adding a new virtual hardware that provides window views. Hatari now passes all window events coming from the Overlay Mediator into the sandbox via a memory-mapped device. Until this point we did not modify GEM itself. The same way, the sandboxed operating system can propagate its window events to Hatari that—in turn—forwards them to the Overlay Mediator.

To make GEM actually use these new virtual hardware facilities, we had to install a small hook of less than 200 lines of assembly code at the GEM system-call interface. Thus, we succeeded to integrate basic GEM windows into our Overlay Window System with only a very small effort—even without having any source code of GEM.

3.4 Results

Our successful experiments with integrating the user interfaces of the sandboxed legacy operating systems L^4Linux (X11) and GEM into the trusted DOpE environment demonstrate the universality of the Overlay Window System approach (Figure 5).

Thanks to the used L4/Fiasco kernel, we kept the code complexity of the trusted computing base in our scenario as low as only 30,000 lines of code. In Figure 4, the components of the trusted computing base are highlighted. Note that the Overlay Mediator does not belong to the trusted computing base. It acts as a translation tool that is exlusively used by one sandbox. Therefore, it belongs to the protection domain of this actual sandbox.

The raw output performance of the sandboxed window system is equal to the traditional desktop-in-a-window approach because we do not introduce new pixel-copy operations. With the help of L4/Fiasco, DOpE can share pixel buffers with its client applications. Thus, a virtual machine can render its graphical output into the same buffer that is used by DOpE to draw the window on screen.

4 Application scenarios

4.1 Multiple instances of L^4Linux

In Section 3, we presented how we integrated XFree86 with an Overlay Window System. This technique can now be deployed to realize the initial scenario of running trusted and untrusted applications on one device as constituted in Section 1.

4.1.1 Multi-level security

This scenario is a special case of a multi-level security architecture. For each security level, we start one instance of L^{4}Linux with a corrsponding Overlay Mediator. For example, when using two instances of L^{4}Linux, one instance can be used to perform security-sensitive tasks such as editing and storing confidential data, while another instance is dedicated to download files from the Internet and execute untrusted code. Both instances run inside isolated protection domains unable to communicate with each other on their own. From the user's point of view, both instances are integrated into one desktop environment. Thus, he can interact with both domains in a natural way. The Overlay Window System labels all windows with their corresponding domain to provide the integrity of displayed information to the user. Additionally, there is a menu bar on top of the screen that cannot be covered by any window. It displays the identity of the currently focused window. Thus, we can prevent trojan horses running at the untrusted domain to gather security-sensitive information from the attentive user.

4.1.2 Monitored flow of information

In [20], there is an extensive discussion about the flow of information between applications—authorized by the user via explicit drag-and-drop and copy-and-paste operations.

Although all information flow from the trusted domain to the untrusted domain must be blocked, we need to provide a mechanism for transfering data from the untrusted domain to the trusted domain. For this, we want to use the well established X clipboard mechanism. A dedicated oneway-communication channel from the untrusted domain to the trusted domain must be established and monitored by the L4/Fiasco kernel. This communication channel can now be used to implement a custom protocol into two dedicated X client applications—each running inside one domain—to tunnel clipboard information over the unidirectional communication channel.

4.2 Linux and Windows applications on one desktop

VMware [7] permits to run a sandboxed Microsoft Windows operating system on top of Linux. Thus, the great functionality of Windows can be combined with security policies implemented in Linux. With the current implementation, VMware either uses a fullscreen mode or displays the Windows desktop inside one X11 window.

In Figure 6, we illustrated a proposed solution for integrating Windows and Linux/X11 applications into one desktop environment. The foundation of this scenario is a commodity Linux running an X11 session with normal X11 desktop applications and an Overlay Mediator implemented as a plain X11 application. Just beside the visible X session, a second X server (Remedy X11) is executed in the background and hosts the fullscreen window of VMware running the Windows operating system. Remedy X11 uses custom drivers for input and a virtual frame buffer, which use the Overlay Mediator as backend.

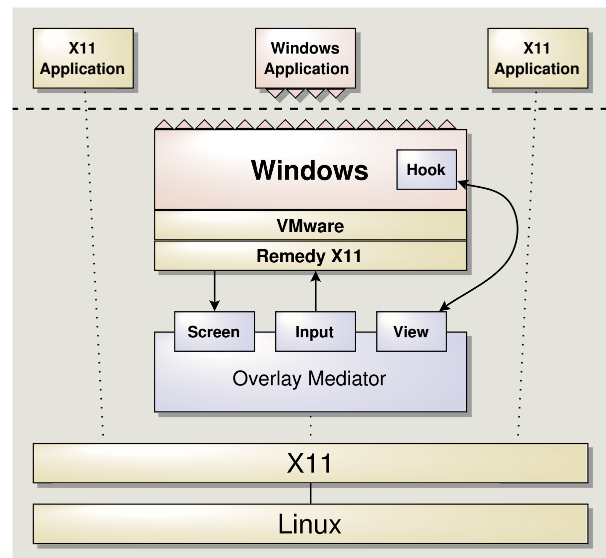


Figure 6: Running Microsoft Windows applications and Linux/X11 applications side by side. VMware uses a remedy X11 with virtual frame buffer and custom input device driver while the Overlay Mediator displays guest windows as host (X11) windows. Microsoft Windows is extended by a hooked function that uses the view via network sockets.

For synchronizing the views between the Overlay Mediator and Windows, a hook in the Windows Operating system is needed. One way to implement such a hook is providing a custom Explorer replacement. The communication of the hook with the Overlay Mediator can be performed via sockets through the host-only networking facility of VMware.

4.3 Enhancing X11

As identified in [20], the X window system relies on cooperating client applications. It does not protect client applications against each other. Once a client application has access to the X server, it can compromise the security of the X session by sniffing keys, grabbing the mouse, taking screenshots and blocking the whole screen via a fullscreen window.

The L^{4}Linux approach combined with an Overlay Window System as presented in Section 3 allows the execution of high-availability and trusted applications aside a running X server as native L4 processes. Examples for such trusted applications are password checkers, signing applications, login managers, and video players with support for digital rights management. These applications can neither be affected nor observed by L^{4}Linux and thus, can implement indispensable services. Furthermore, L4/Fiasco and the DOpE window system provide real-time support for native L4 applications, which can implement real-time services with graphical output at guaranteed frame rates.

From the user's point of view, such trusted or real-time applications are tightly integrated into one environment together with X client applications.

4.4 Digging out buried operating systems

Ancient operating systems were designed without any security considerations assuming a nice behaviour of all applications. This assumption does not hold true with todays broad use of the Internet. Still, such operating systems host a wide base of available applications. Instead of abandoning these operating systems including all applications, their functionality can still be used via appropriate sandboxes. The Overlay Window System goes a step further and enables a tight integration of ancient applications into a productive user environment of today.

Figure 5 depicts an example of this application scenario. Window-based GEM applications running inside an Atari ST emulator are displayed in distinct DOpE windows as described in Section 3. There is also an X session running on an L^{4}Linux instance. Additionally, there are native windows of the DOpE window system displaying the output of real-time applications. Although the applications belong to completely isolated protection domains, the windows of these applications are integrated into one user environment.

5 Related work

5.1 Integration of legacy software

In this section, we go into prominent examples for solutions integrating existing software into foreign platforms. The virtual-machine approach adopted by VMware [7] and Mac-on-Linux [5] is the first solution that comes to mind. These virtual machines do a good job in reusing any operating system and application completely unmodified. The dark side of the virtual-machine approach is that existing implementations obtain almost no user-friendly interfaces. The user ends up with console-style graphical user interfaces abandoned years ago.

Beside virtual machines, MacOS X [2] perfectly integrates modern and classic MacOS applications with applications based on UNIX and X11 because usability is the major design criteria for MacOS. In MacOS X (Quartz), each window is buffered individually and the buffers possibly consume a better part of the memory. The Overlay Window System approach needs no extra space beside one virtual frame buffer for each legacy OS. In general, the MacOS solution needs—depending on the affected host operating system—extensive modifications of the legacy system respectively (as with MacOS X) strong consideration in the design process. On the other hand, the Overlay Window System approach only needs small helpers or patches of the legacy system.

Another, radically different approach is the basis for Wine [8]. It implements Windows functionality based on the host-system infrastructure and provides the Application Binary Interface (ABI) of the Windows operating system. This approach is inherently platform specific and tight-knit with the host operating system to achieve reasonable performance. A grave shortcoming of this approach is that Wine has to provide the complete ABI; a goal not achieved until now because new features are added to the ABI continuously and used by new applications. This problem does not exist in our approach combined with a virtual machine. A further advantage of an Overlay Window System compared to Wine is its simplicity and modest demands to the host platform.

5.2 Trusted window systems

Trusted X [10] was an approach to make the X Window System usable for multi-level security systems by running dedicated untrusted X sessions (single-level server) for different compartments and factoring out the commonly used, security-critical functionality into a separate trusted component (TX master). TX master composes the output of the single-level servers, distributes input events and implements the policy for the sharing of information among the single-level servers. It implements similar functionality as an Overlay Window Server but is tied to one particular protocol (X protocol). Trusted X is an example of a hardened legacy window system that is able to execute the broad range of commodity X client software. The concrete implementation of the security-sensitive parts of Trusted X— as presented in [10]—consists of 30,000 lines of code. As discussed in [20], such enhardened legacy window systems can still be regarded as complex when compared to other approaches of trusted window systems and fundamental design flaws remain.

A much lower complexity can be achived by considering adequate security models during the design of a trusted window system. For example, the EROS window system consists of only 5,000 lines of code while it implements security policies at the granularity of applications. The drawback of such solutions is the absence of any available commodity applications.

An Overlay Window System as presented in this paper combines the advantages of both the broad range of existing applications on legacy systems and the ultimately low complexity of specially-designed trusted window systems.

6 Conclusion

In this paper we presented a mechanism and its implementation to integrate multiple legacy window systems of distinct security domains into one trusted user environment. While deploying unmodified legacy window systems executed at the trust levels of their corresponding domains, the trusted computing base contains only a simple multiplexer (Overlay Window System). With our implementation, we kept the overall trusted computing base lower than 30,000 lines of code—including the L4/Fiasco kernel, basic resource managers and DOpE as Overlay Window System.

In our paper we focused on virtual frame buffers for passing raw pixel data from legacy systems to the Overlay Window System and thus left the hardware-acceleration facilities of modern graphics cards unheeded. However, these facilities are essential for a lot of today's applications, for example, games and 3D software. Furthermore in the current design, legacy systems cannot export features like desktop enhancements, icons, or menubars outside of windows. In future work, we will address these problems.

We described in Section 3.3 how we extended an emulator by adding a dedicated virtual device for the propagation of window events between the sandboxed operating system and the host system—the view interface. In reverse: If such a view interface pays off in an emulator/virtual machine, could this be also a reasonable extension for real hardware? Modern graphics cards already implement a very similar feature—called overlay. It is mostly used to display video streams while bypassing the host's windowing system and thus, avoiding overhead when displaying streaming data. Consequently, a graphics card could be the right place to implement (at least parts of) a minimal-complexity Overlay Window System.

References

[1] Aewm website. URL: `http://www.red-bean.com/~decklin/aewm/`.

[2] Apple MacOS X Website, howpublished = URL: `http://www.apple.com/macosx/`.

[3] Hatari website. URL: `http://hatari.sourceforge.net`.

[4] libSDL website. URL: `http://www.libsdl.org`.

[5] Mac-on-Linux website. URL: `http://www.maconlinux.org/`.

[6] TCPA website. URL: `http://www.trustedcomputing.org`.

[7] VMware website. URL: `http://www.vmware.com`.

[8] Wine website. URL: `http://www.winehq.org/`.

[9] XFree86 website. URL: `http://www.xfree86.org`.

[10] Jeremy Epstein, John McHugh, Hilarie Orman, Rita Pascale, Ann Marmor-Squires, and Bonnie Danner et al. A high assurance window system prototype.

[11] Norman Feske and Hermann Härtig. Demonstration of DOpE — a Window Server for Real-Time and Embedded Systems. In *24th IEEE Real-Time Systems Symposium (RTSS)*, pages 74–77, Cancun, Mexico, December 2003.

[12] Norman Feske and Hermann Härtig. DOpE — a Window Server for Real-Time and Embedded Systems. Technical Report TUD-FI03-10-September-2003, TU Dresden, 2003.

[13] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual-machine based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, New York*, October 2003.

[14] Tal Garfinkel, Mendel Rosenblum, and Dan Boneh. Flexible os support and applications for trusted computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2003.

[15] Hermann Härtig. Security Architectures Revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[16] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998.

[17] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[18] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[19] J. S. Shapiro. *EROS: A Capability System.* PhD thesis, University of Pennsylvania, April 1999. Available from URL: `http://srl.cs.jhu.edu/~shap/EROS/thesis.ps`.

[20] Jonathan Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. The EROS trusted window system. Technical Report SRL2003-05, Johns Hopkins University, 2003.