Securing Graphical User Interfaces

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der

Technischen Universität Dresden

Fakultät Informatik

eingereicht von

Dipl.-Inf. Norman Feske

geboren am 29. März 1978 in Dresden

Gutachter: Prof. Dr. rer. nat. Hermann Härtig (TU Dresden)
Prof. Dr. rer. nat. Andreas Pfitzmann (TU Dresden)
Prof. Emin Gün Sirer (Cornell University, Ithaca, NY)

Tag der Verteidigung: 5. Januar 2009

Dresden im Februar 2009

Acknowledgements

On my way towards the academic degree Doktoringenieur, I had many supporters. In the following, I can only mention a few of them. With regard to finding my professional focus, I am indebted to my parents, in particular to my father. With his patience and support during my childhood, he turned my sparks of interest in engineering into the dedication that drives me today. During the 1990s, my interests in computer science had been primarily shaped by the Atari demo scene, which incited me to steadily advance my skills in low-level graphics programming. The gained experiences and the relationship with the members of the Atari community, in particular with my close friend Matthias Alles, still play an important role in my present life. In 2002, Michael Hohmuth introduced me to the TU Dresden OS group and motivated me to join the team. Since then, he maintained his attention to my work and continuously supported me by the means of guidance and advice. The working environment provided by Prof. Härtig as the head of the OS group enabled me to pursue my original research interests. I want to specifically thank the group members Christian Helmuth and Alexander Warg for many fruitful discussions that influenced my work, and Angela Spehr for her moral support. Thanks to Prof. Härtig and his relationship with Intel, I had the honor to join Intel's platform virtualization group as an intern in 2005, which turned out to be an invaluable experience. Working together with David J. Cowperthwaite, Sebastian Schönberg, and Richard A. Uhlig's research group was a huge motivation to carry on my research on securing graphics. I gained further motivation from a very inspiring email conversation with Jeremy Epstein who I highly regard as the pioneer in the domain of secure GUIs. The textual quality of the final version of this thesis was greatly improved by the feedback from early reviewers. I want to specifically thank Michael Roitzsch and Michael Hohmuth. I am deeply grateful to Prof. Dr. Andreas Pfitzmann and Prof. Emin Gün Sirer who reviewed the final version of this document.

Finally but most importantly, I thank my beloved wife Christin for not just tolerating but for strongly supporting my ambitions and for maintaining the healthy balance between my professional life and the family life with our wonderful children.

Contents

Chapter 1

Introduction

During the past three decades, the workload on desktop computers underwent fundamental changes that have driven the evolution of graphical user-interface (GUI) architectures.

For the first generation of applications with GUIs, security was no concern. In a single-tasking environment as illustrated in Figure 1.1, a GUI application included device drivers for graphics output and user input and accessed the hardware directly. Because only one application was executed at a time, the crash of the computer was synonymous with the crash of the currently executed application. Thus, a bug in the application affected only the application itself. In the worst case, the reboot of the machine compromised data of only one single application.

This did not hold true with the rise of multi-tasking desktop environments. In such environments, multiple applications have to share the physical graphics hardware. Figure 1.2 depicts two applications that connect to a central GUI server rather than accessing the hardware directly. The GUI server translates high-level GUI primitives such as windows, lines, and text to the device level. It is the only program in the system that accesses the hardware directly. This technique was introduced with the Blit terminal [56] and the Xerox Alto personal computer [68] and was later adapted by all predominant desktop OSes of the late 1980's such as Mac OS, Microsoft Windows, Digital Research's GEM, Amiga OS, and Acorn Risc OS. In a multi-tasking environment, a computer crash had more fatal consequences because it affected each executed application. Furthermore, the likelihood for bugs to happen increased with a growing number of concurrently executed applications. The typical way of resolving these issues was fixing the applications, which the user regarded as trusted.

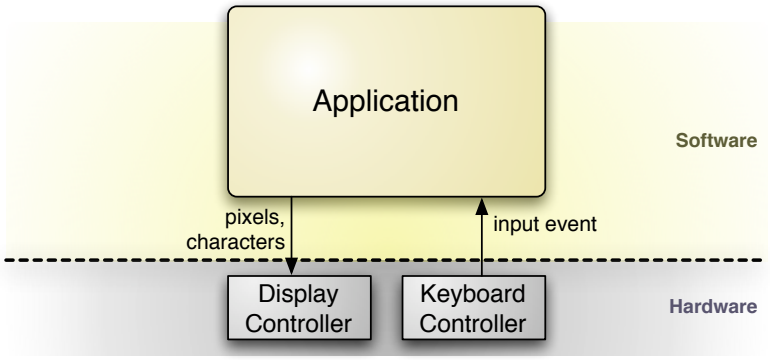


Figure 1.1: One application accesses the hardware (screen, input devices) directly.

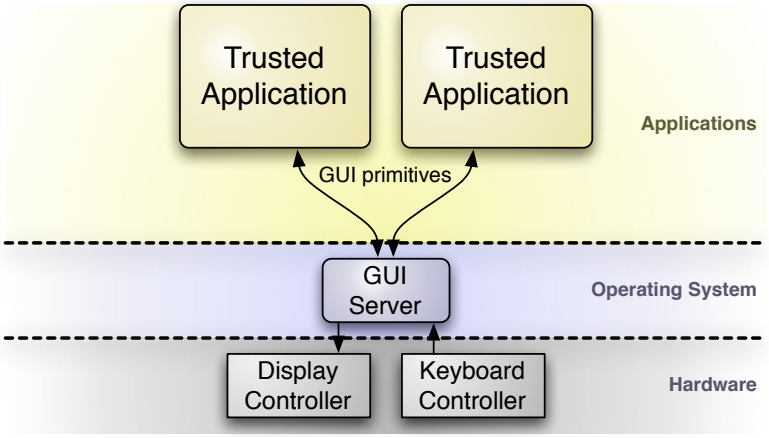


Figure 1.2: Two applications executed on a multi-tasking OS use one central GUI server, which accesses the hardware on behalf of its clients.

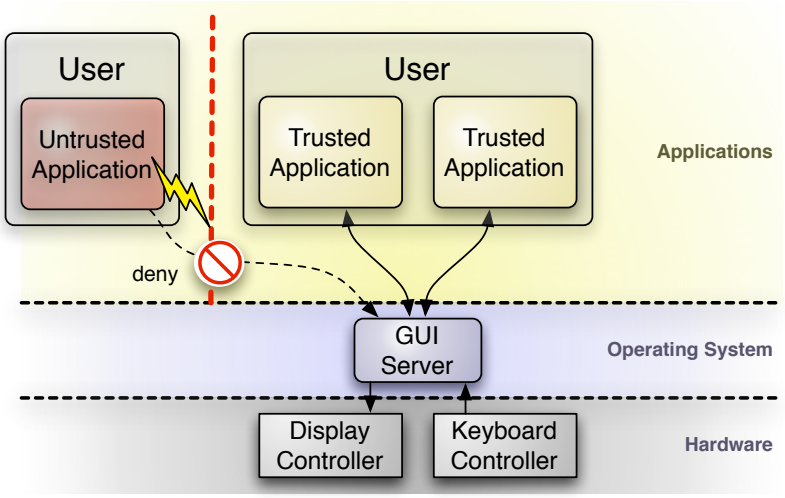


Figure 1.3: Applications executed in a multi-user environment are protected from the applications of another user. The access to GUI sessions is protected by client authentication at the granularity of users.

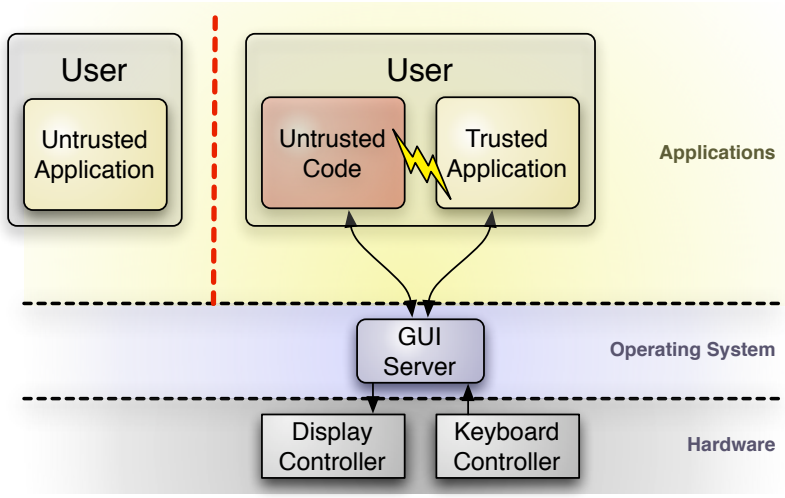


Figure 1.4: In inter-networked environments, the presumption that the user's own applications act in his interest does not hold anymore. The user executes untrusted code and thereby puts his sensitive applications at risk.

Operating systems evolved to support fault isolation between processes and users. UNIX shaped the predominant security architecture with discretionary access control on a per-user basis. With UNIX, each user is able to choose a custom set of applications he trusts and is not put at risk by the applications executed by another user on the same machine (Figure 1.3). Still, the role of each application in such a locally networked multi-user environment is that of a trusted program that works on behalf of its user. The set of installed applications is rather static and is managed by a privileged LAN administrator.

In contrast to locally networked multi-user environments of the past, most desktop computers today are connected to the Internet. The set of executed applications on one machine is no longer static but changes constantly. For example, when an operating system gets transparently updated over the Internet, the user is virtually unaware of the actual implications of the update but he has to trust an OS distribution. In addition to executing trusted code from the user's chosen OS distribution, the user executes untrusted code that is embedded in the media he consumes over the Internet. Websites cause the web browser to locally execute JavaScript code, Adobe Flash applets, or Java code. On today's commodity desktop OSes, each program that is directly or indirectly executed by the user can freely access the GUI, inspect and manipulate other applications, present a Trojan Horse to the user, access global information such as the clipboard, or render the whole GUI inaccessible.

The user is not anymore in effective control over the code that he executes because both the underlying operating system as well as the processed active content change constantly. Figure 1.4 shows the lack of protection of a trusted application from untrusted code executed by the user, who is—in a real setting—completely unaware of all the code executed in his name. The regularly changing code base combined with the overwhelming complexity of the user's trusted computing base renders a per-user-based discretionary access-control scheme ineffective. On the other hand, the mentioned complexity of a modern desktop OS cannot be avoided because of the following two reasons. First, the user expects his operating system to support existing applications. To fulfill this requirement, the OS has to carry the legacy of its whole lifetime and thus prevents legacies from the past to get disposed of. Consequently, with each new OS version that adds a new feature, the complexity increases. Second, growing functionality as expected by users imposes constant rise of code complexity. A prominent ex-

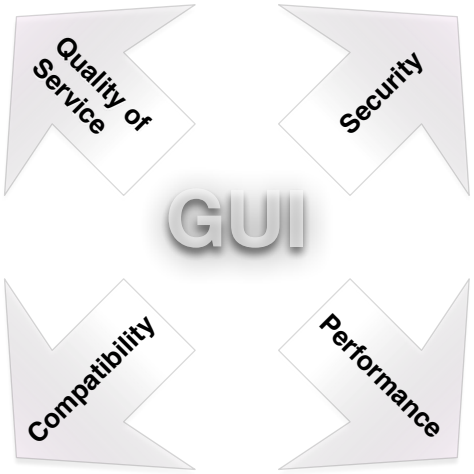


Figure 1.5: The challenges of GUI-architecture design.

ample is the necessity of modern GUIs to support high-performance hardware-accelerated 3D graphics, which is a moving target that gets more advanced with each generation of graphics devices. This requirement implicates a huge amount of code and, as of today, makes graphics drivers a major cause for stability problems on desktop OSes [11].

In addition to functional requirements, GUIs should meet ergonomic and nonfunctional expectations such as a bounded and low latency of screen updates, guaranteed responsiveness to user actions, a constant frame rate for playing media, and the prevention of overload situations. I subsume these nonfunctional requirements under the label quality of service. As a matter of fact, current commodity GUIs provide their services in a best-effort fashion but fail to effectively address quality of service.

Figure 1.5 summarizes the four competing challenges that GUI architectures have to face. Security implies the requirement of low complexity. Performance translates to efficient use of hardware-accelerated graphics devices. As illustrated, the goals are in conflict and thus, existing GUIs do only achieve subsets of these goals. The following examples highlight the crux of these conflicts by referring to the X window system as a representative of today's commodity GUIs. Similar issues can be reported about other commodity GUIs such as the Windows GUI.

Compatibility and security

When the X window system was designed in 1984 [64], the importance of security was subordinate to interoperability between applications. The design of the X clipboard protocol, the use of global window IDs, or the way of input-event handling was motivated to facilitate inter-application communication. Today the large base of existing applications relies on these protocols. Adapting the design to the present security requirements would break application compatibility. For this reason, today's commodity Linux distributions ship an X server that permits each application to observe user actions, to control other applications, to take screen shots of the whole desktop, to globally grab the mouse cursor, to make the GUI inaccessible by opening a full-screen window, and to change the keyboard layout.

Compatibility and quality of service

A similar legacy is the redraw-processing protocol of X window system, which relies on mutual collaboration of the GUI server and its clients. The GUI server has no information about

the number of graphics operations performed by a client on a window-redraw request. The redraw-processing time depends on the client and thus, the GUI server is inherently unable to provide any means of quality of service if one misbehaving client is present in the GUI session.

Performance and security

When the race between different hardware-accelerated graphics devices started, raw graphics performance became the ultimate selling point of these devices. Therefore, high performance served as the primary criterion of the graphics-driver infrastructure. Maximum performance and flexibility can be achieved by enabling 3D-graphics applications to directly access the graphics-processing unit (GPU) of the physical device. Consequently, this design was implemented into the Linux kernel and is known as DRI (Direct Rendering Infrastructure). The downside of this approach is that each 3D-graphics application acts indeed as a graphics driver that must cooperate with all other graphics applications. A faulty or malicious graphics driver, however, is able to put system reliability and security at a high risk. In effect, the security and reliability of a DRI-supporting Linux system is at the mercy of each DRI application.

The scope of my work

The aim of my work was to resolve the conflicts between the four stated goals. On the course of my work, I developed key techniques and substantiated the concepts by a number of exhaustive experiments. To each goal, I dedicate a chapter that presents the rationale behind my propositions for designing the GUI-server, refers to related work, sketches my actual engineering work, and reports on my experimental results.

In Chapter 2, I describe how to turn the GUI server into a strictly periodic process and thereby guarantee the quality of service for the GUI. Chapter 3 presents a technique for merging multiple window systems into one integrated desktop to enable the coexistence of legacy GUIs alongside modern and secure GUIs. Secure client-side window management, which I describe in Chapter 4, clears the way for reducing the GUI's complexity and, at the same time, provides essential security functions. Chapter 5 explains how my proposed design benefits from recent technical advances of graphics devices to make hardware-accelerated graphics fully exploitable without sacrificing the achieved security properties. When combined, the presented techniques enable the construction of a GUI architecture that achieves the four competing goals.

Of course, the outcome of my work is not a ready-to-use real-life solution for the pressing security problems with GUIs. With the complexity and diversity of graphics hardware alone, such a mission is beyond the power of one individual. The developed techniques and their composition, however, may hopefully serve and inspire developers of current and future commodity GUIs to address the four competing goals of GUI architectures more effectively than today.

Primary contributions

With the development of the DOpE GUI server described in Chapter 2, I initially focused my work on quality of service. DOpE is the first GUI server modelled as a periodic real-time process. By following this approach, I developed techniques to fit different GUI workloads into the periodic execution model. Thereby, I made optimization techniques such as lazy

updating of GUI client representations and redraw dropping applicable to the domain of windowed GUIs. The resulting design does not only accommodate QoS-sensitive GUI clients alongside non-real-time workload but it prevents overload situations by design.

By increasingly moving my research focus to security, I identified the combination of virtual machines with seamless window-system integration as the enabler for reconstructing a GUI server that is free from legacies but maintains compatibility to existing applications. I conducted various experiments to explore the practical application of seamless window-system integration and the involved engineering costs.

The experimental results stimulated the main contribution of this thesis, which is the fresh redesign of the GUI server for drastically improving security over the state of the art while maintaining the availability of existing GUI applications.

My focus on software-based rendering raised the question of how to combine the achieved solution with hardware-accelerated graphics. Therefore, I explored various graphics devices with regard to their applicability for secure GUIs and realized how recent feature additions to graphics devices help to handle this issue.

Auxiliary contributions

The main body of this document addresses GUI architecture but a secure GUI alone does not resolve the security issues of today's commodity desktop OSes because these OSes do not provide strong security mechanisms for isolating applications and for countering denial-of-service attacks. The even more problematic observation is the steady growth of OSes in terms of complexity, which increasingly exposes the depending applications to a growing number of bugs and attack vectors for zero-day exploits.

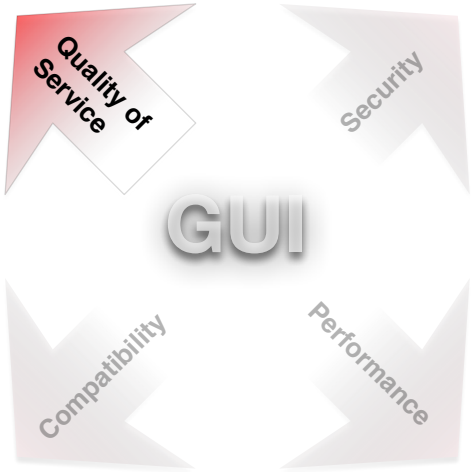
As a side project of my GUI-related work, I developed a vision of how a secure OS foundation that is able to support general-purpose desktop-OS workload may look like. In joint work with my colleague Christian Helmuth, I turned this vision into an OS design and implemented a working prototype [31]. With our implementation of the base components and protocols, we created the foundation of a secure OS at a source-code complexity of less than 20,000 lines of code.

In addition to the practical work described in this document, I conducted a number of further experiments that are loosely related to the goals of my thesis. Several of these experiments turned out to be of value beyond the scope of my personal work. My GUI implementations are employed by several industrial and academic research groups, for example at Intel, ST microelectronics, EADS, and they are valued as major components of the DROPS/Nizza research OS developed at the University of Technology Dresden. Even though originally designed for the DROPS real-time operating system, I have ported the DOpE GUI server to several other platforms such as Linux, a Coldfire-based embedded platform, and a custom hardware based on Xilinx' Microblaze soft core. Thereby, DOpE is the first full-featured windowed GUI running on a Spartan3 FPGA system-on-chip platform[1]. The outcome of my work enabled or contributed to more than a dozen student research projects and it is part of ongoing research activities.

[1] In addition to the Microblaze soft core, the Spartan3 FPGA hosts custom controllers for memory, display, mouse, and keyboard.

Chapter 2

Quality of Service



Apart from providing the functionality of multiplexing input and output devices to multiple GUI clients, a GUI server is expected to meet ergonomic and nonfunctional requirements such as a bounded and low latency of screen updates, guaranteed responsiveness to user actions, a constant frame rate for playing media, and the prevention of overload situations induced by faulty applications. Today's commodity GUIs provide their services in a best-effort fashion but fail to effectively address these expectations. Furthermore, with the workload of untrusted and potentially malicious programs being executed alongside sensitive applications on inter-networked computers, the lack of quality-of-service in the GUI-server design can become a reliability risk. This chapter presents a systematic approach to the design of a quality-of-service-aware GUI server and explains the employed key techniques.

Section 2.1 recalls past experiences with quality of service on the GUI level to substantiate the requirements to be addressed by the design presented in Section 2.2. Section 2.3 elaborates on the extensive practical experiments made with my custom GUI server implementation. Section 2.4 conclude the chapter with pointers to related work.

2.1 A brief history of quality of service on the GUI level

The following two examples are a flashback on how GUIs approached quality of service in the past. Although the examples seem rather mundane, the consequences of the taken approaches are still present in the current versions of GUI servers.

2.1.1 Example 1: Flicker-free and smooth mouse cursor

The life story of the mouse cursor over more than two decades is insightful for understanding the difficulty of achieving quality of service at the GUI level. The main challenge here is that the user expects the mouse cursor to move smoothly with a latency of no more than 20 milliseconds or less whereas the GUI server has to streamline mouse-cursor updates into the processing of potentially long-taking redraw operations. In early GUI servers, two techniques had been used to decouple the cursor-drawing logic from GUI redrawing:

On the Amiga home-computer series, the mouse cursor was realized by using a hardware sprite featured by the Amiga chip set. This hardware feature relieved the Amiga OS from resolving the concurrent screen access for handling the mouse cursor and the actual GUI.

Most of the other desktop OSes had to live without hardware sprites and the drawing of the mouse cursor had to be synchronized with redraw activity of the GUI server. Before drawing the cursor, the GUI server saved the cursor's background into a buffer. To move the cursor to a new position, the GUI server restored the saved background at the old position first, subsequently saved the background at the new position, and drew the cursor there. During the times when the GUI performed redraw operations on screen, the mouse cursor was temporarily being switched off by just restoring its background. Consequently, during long-taking drawing operations, the mouse cursor flickered and started to move in a non-smooth fashion. This method was used by the GUI of Windows version 3 and early versions of the X window system. The Atari TOS and Mac OS provided a certain degree of quality of service by sampling mouse-input events at a much higher rate than the vertical sync frequency of the screen and by calling the mouse-cursor update routine directly from the dispatcher of the vertical blank interrupt, which is an interrupt triggered by the vertical blank of the display. Thus, the mouse cursor was typically updated during screen blanks and was moving at a smooth rate of circa 70 updates per second. Still, the mouse cursor flickered during long drawing operations.

For commodity desktop OSes, the problem of flickering and jerky mouse cursors was never solved in software. In fact, this problem was regarded as critical enough to impose hardware changes by introducing the mouse cursor as a feature into graphics cards. Thereby, the hardware sprite as originally provided by the Amiga chip set was reintroduced and still resides as a standard feature in modern graphics cards.

2.1.2 Example 2: Media playback at a constant frame rate

With the rise of multi-media applications in the 1990's, smooth media playback at a constant frame rate became an important feature of desktop OSes.

The state of the art was a GUI server that provides the sequential execution of graphical primitives and served its GUI clients in a best-effort fashion. As the temporal profile of the graphical primitives varies, time-intensive graphical primitives can significantly delay subsequent operations. GUI clients compete against each other for processing graphical primitives at the GUI server. Consequently, the performance and latency of the graphical output of each GUI client depends on the graphical primitives as issued by all other GUI clients. Although this best-effort technique is feasible for non-media applications, a media application requires the update of its window at fixed intervals. For such an application, unbounded redrawing delays and unpredictable graphical throughput caused by concurrent applications are not acceptable.

Analogous to the mouse-cursor problem, a fundamental architectural change of the GUI server seemed to be less feasible than changing the hardware. Thus, hardware vendors en-

```
#include <X11/Xlib.h>

int main(int argc, char **argv)
{
  Display *dpy;
  Window   root, win;
  int      screen;
  int      x = 0, y = 0;

  /* create window */
  dpy    = XOpenDisplay(NULL);
  screen = DefaultScreen(dpy);
  root   = RootWindow(dpy, screen);
  win    = XCreateWindow(dpy, root,
                         -200, -200,       /* position */
                         1000, 1000,       /* size     */
                         0,                /* border   */
                         CopyFromParent,   /* depth    */
                         InputOutput,      /* class    */
                         CopyFromParent,   /* visual   */
                         0, 0);

  /* issue jobs to the X server by constantly changing the window position */
  while (1) {
    XWindowChanges wincfg;

    wincfg.x = x;
    wincfg.y = y;
    XConfigureWindow(dpy, win, CWX | CWY , &wincfg);
    XMapWindow(dpy, win);
    x = (x + 10) % 1000;
    y = (y + 10) %  800;
  }
}
```

Figure 2.1: Denial-of-service attack targeted at the X server. After creating a single window, the program stresses the X server by constantly adjusting the window position in a spinning loop. Once started, it renders the whole X session inaccessible and leaves the user no other choice than killing the X server. This attack works on the version 7.2 of X.org [33] that comes with major Linux distributions such as Ubuntu 7.04 released in April 2007.

hanced graphics cards by supporting color-keyed overlays, which enabled the display of media data that bypasses the window system.

Today the media-playback problem is relieved by the capacities of hardware-accelerated graphics. But even on Mac OS X featuring Quartz as one of the technically most advanced GUIs of today, uniform performance degradation occurs in the presence of several active windows in the GUI session.

2.1.3 The crux of encumbering quality of service by design

Both examples illustrate how GUI developers struggled to achieve quality of service, yet were not able to provide an adequate software solution given the existing GUI architectures at the time. It seemed easier to introduce hardware workarounds for these specific problems rather than providing quality of service through software. Another lesson to be learned from these examples is that the hardware workarounds by themselves introduced a number of new problems. For example, the number of overlays supported by the hardware imposed a new limit on the number of media-displaying windows on screen. Furthermore, each hardware vendor implemented the features differently such that GUI servers now have to support a range of different implementations. With each hardware workaround, the device drivers grew more complex and less manageable.

Although the given examples may seem to address blemishes, quality-of-service requirements extend to availability as a vital security property. For ensuring the responsiveness of the GUI server and for protecting its overall availability, the GUI server must be able to

deal gracefully with overload (denial of service) situations that may be induced by faulty or malicious GUI clients. Figure 2.1 presents an attack targeted at the availability of a current-generation X server. Unlike the previous examples, this problem is not solvable by a hardware workaround. Thus, I took a step back from the existing GUI implementations and created a GUI design that addresses quality-of-service concerns from the beginning. To recapture the actual requirements regarding quality of service, the GUI server has to serve

The user who expects immediate feedback on his input. This includes a smoothly moving mouse cursor but also mouse-focus indication when the mouse cursor is moved over a button and immediate feedback when a button is pressed or when a window is moved. Empirically, an appropriate latency for visual feedback is 20 milliseconds.

Real-time media applications that require periodic updates at predefined intervals. Therefore, the load on the GUI-server induced by such applications is predictable. Typical update intervals are in the range of 20 to 40 milliseconds.

Non-real-time applications that may cause any amount of drawing activity at any time. Their behaviour is not predictable but delaying their graphical output in overload situations is acceptable.

As best illustrated by the described examples, a best-effort strategy as applied by existing commodity GUIs cannot guarantee the temporal requirements by the user and real-time media applications. The only way to meet these requirements is to enable the GUI server to cautiously manage physical resources such as processing time and bus bandwidth according to temporal constraints. Therefore, we need to model the GUI server as a real-time process that schedules and executes redrawing jobs.

2.2 Designing the GUI server as a resource scheduler

To successfully plan ahead of time, a scheduler relies on the knowledge of scheduling parameters, in particular the execution time of each job, in advance of execution.

The classical approach for performing redrawing jobs, however, relies on a tight interplay of the GUI server with its GUI clients. To update a screen portion, the GUI server determines the set of GUI clients that are visible at the screen region and instructs each GUI client to redraw its visible portion. In turn, each GUI client responds to the request by invoking a sequence of graphical primitives composing the client's pixel representation at the GUI server. The set of graphical primitives as supported by the GUI server comprises for example the drawing of lines, the output of text strings, and the filling of polygonal shapes. Consequently, the time needed to update a screen portion depends on the number and type of graphical primitives as selected by each GUI client to produce its pixel representation. Hence, the GUI server cannot reason about the time needed to execute the involved graphical primitives ahead of execution. From the GUI server's point of view, the execution time of each redraw job is unbounded. Major commodity GUIs such as the Windows XP GUI, Mac OS (until version 9), and the X window system employ such a protocol.

2.2.1 Making worst-case execution times of redrawing jobs predictable

A method to dissolve the dependency of the GUI server from its clients during screen updates is to move each GUI client's graphical representation into the GUI server and thereby enable the GUI server to autonomously reproduce pixels out of the client representation locally stored at the server. This representation can be based on raw pixel data or on a higher-level

abstraction such as a widget set including buttons, menus, and other basic GUI elements. Performing the transformation of each client's representation to pixels locally enables the GUI server to predict all graphical primitives that are needed for any screen update and, as a consequence, to estimate overall execution times of redrawing jobs in advance of execution.

The sequence of graphical primitives is a function of both the known transformation of the GUI client's representations to pixels and the actual window layout. The latter, however, may have a significant influence on the required graphical primitives but is not known at admission time of a GUI client. This problem is best illustrated by the *painter's algorithm* as used by the Windows Vista's Desktop Window Manager (DWM), the Quartz window manager of Mac OS X, and the composite extension of the X window system.

Deficiency of the painter's algorithm As a painter, the algorithm produces the final image by painting all objects ordered by their distance from the viewer, starting with the rearmost (background) and finishing with the foremost (top window). In the final image, window portions that are covered by other windows get correctly over-painted and are no longer visible. Combined with the use of alpha channels, which is comparable with applying layers of watercolor with different translucencies to a canvas, this algorithm provides maximum flexibility with regard to the shapes of windows and their opacity. With regard to predicting redraw-execution times however, this algorithm is not well suited. Even though the sequence of graphical primitives used by the Painter's Algorithm can be determined immediately prior execution, we cannot predict a realistic redraw execution time for a specific GUI client at its admission time because the actual costs depend on the other GUI clients and on the window layout, which is not fixed during the lifetime of the GUI client. Therefore, the admission of new GUI clients is based on an overly pessimistic worst-case redraw-execution time assuming that all windows are covering each other and thus, must be painted for each redraw operation.

Decoupling redraw-execution times of different clients To dissolve the inter-dependency of windows from each other during the redraw of one particular window (target window), the painting algorithm should limit its operation to only the target window but should not paint the windows of any other windows. This can be achieved by preceding the painting operation by a geometric analysis that computes the target window's visible portion. The visible portion is determined by successively cutting out the shape of each window in front of the target window from the target window's shape. Such a technique was originally employed by most window systems but current-generation commodity GUIs discarded this approach in favour of executing the painters algorithm via hardware-accelerated graphics.

Figure 2.2 illustrates this procedure. The resulting shape is then used as clipping boundary while painting the target window to mask out all pixels that are covered by other windows. The worst-case redraw-execution time for each window corresponds to painting the window when fully exposed and it is invariant toward the presence of other windows and the window layout (ignoring the computational overhead for the geometric analysis at this point). With known temporal characteristics of the single graphical primitives, this technique enables us to predict redraw-execution times prior execution and, therefore, base the admission of GUI clients on realistic worst-case execution times.

2.2.2 Local scheduling of redrawing jobs

With the satisfied precondition of known worst-case redraw-execution times, the construction of the GUI server as a periodic process clears the way for deploying the full variety of well-

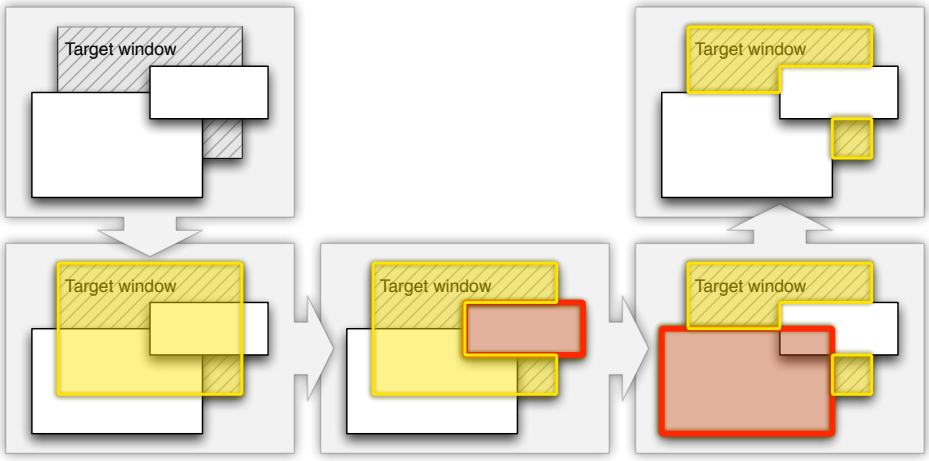


Figure 2.2: Determining the visible portion of a window by successively clipping the window's shape against each overlapping window.

understood admission and scheduling strategies for such processes. In addition, the redraw-job scheduler can take the different characteristics of planned and spontaneous redrawing jobs into account. Once admitted for a defined window size and a fixed update interval, a planned job as used by real-time media applications corresponds to a classical real-time job with its deadline being implied by the update interval. A valid schedule for a set of planned jobs can be obtained by using a standard algorithm such as Earliest Deadline First (EDF). In contrast, spontaneous jobs can be induced at arbitrary times by any GUI client posting an update of its GUI representation or by user interaction. The occurrence of spontaneous jobs is not predictable. Once triggered, such a job does not have a deadline assigned but it should be processed as soon as possible (best effort) without affecting planned jobs. A classical best-effort GUI server handles spontaneous jobs only and executes each job when it arrives at a blocking synchronous client interface shared among all GUI clients. In contrast, a real-time GUI server receives spontaneous jobs via an asynchronous client interface, enqueues the incoming redraw-job requests into a redraw queue, and processes redraw-queue elements when the planned schedule permits execution. Because all GUI-client representations are locally known to the GUI server, each redraw-queue element contains only the information about the corresponding window and the window portion to be redrawn but does not include graphical primitives.

Constraining priority inversion through artificial preemption points Executing (low-priority) spontaneous jobs during the time left in the schedule of (high-priority) planned jobs, however, raises a priory-inversion problem because once started, a long-taking spontaneous job must first be completed before the next planned job can be executed. This delay introduces jitter and may corrupt the schedule of planned jobs. Thanks to the locally known GUI-client representations as described in Section 2.2.1, we can estimate the execution time for each spontaneous job before starting its execution. If the spare time slot in the schedule is not sufficient to execute the spontaneous job completely, the job can be subdivided into smaller jobs addressing distinct screen portions of the original job in a way that each sub job's execution time fits nicely into a spare time slot in the schedule.

Managing overload situations Due to the unpredictability of spontaneous jobs that can be issued by any GUI client at any time, the GUI server can be confronted with overload situations. For example, a terminal application may generate a large number of spontaneous jobs when scrolling through large text output. If the GUI server is not able to process graphics operations fast enough, subsequent jobs will stack up at the redraw queue of the GUI server and render the GUI inaccessible until all pending redraw operations are executed. Because this delay is unbounded and depends on the behaviour of the GUI clients, a malicious GUI client would be able to impose the denial of service of the GUI server.

Redraw queueing The key for tackling an unbounded population of the redraw queue lies in the characteristics of redrawing jobs. If there exist multiple jobs in the redraw queue that refer to the same screen region, only the computational result of the most recent job is important whereas the intermediate states as produced by the other jobs get successively repainted. Consequently, such intermediate jobs can be discarded. Video players employ a similar approach for dealing with situations for which the available processing time is not sufficient for decoding all frames of a video stream. In such situations, intermediate frames get dropped to yield the remaining processing time to the most current frame. This technique provides quality of service by trading the smoothness of the video playback for the timeliness of the presented information and thereby prevents unbounded overload situations.

In contrast to a video player that performs frame dropping at the spatial granularity of the whole video frame, a window system composes the screen of a number of potentially overlapping windows, for which redraw dropping can be applied individually. For each incoming redraw job, the GUI server searches for a pending job in the redraw queue that refers to the same window. If such a pending job exists in the queue, this job gets replaced by the compound of the existing job and the incoming job. If both jobs refer to distinct regions of the window, the resulting job will refer to the bounding box of the original job and the incoming job. This way, a once enqueued job for a particular window can successively be enlarged by incoming jobs while staying in the redraw queue. The maximum extent of the enqueued job, however, is limited by the size of its corresponding window. Consequently, the redraw queue's size is bounded by the number of windows present on the screen, which prevents the queue from overrunning. Furthermore, all redraw-queue elements refer to different windows and thus to distinct screen regions. Thanks to successive clipping as described in Section 2.2.1, the actual execution time of each job correlates to the visible portion of its window. The sum of the execution times of all enqueued jobs is bounded by the number of pixels on screen. Therefore, this algorithm enables the GUI server to inherently avoid overload situations and to guarantee a bounded worst-case latency for any graphical output on screen. This worst-case latency is the time needed to perform the redraw of the whole screen.

The combination of redraw splitting with redraw dropping enables the GUI server to streamline redraw operations and input handling into one periodic process. It provides response-time guarantees for user input including visual focus feedback and processes the redraw of all GUI clients. The scheduling of redraw jobs is local to the GUI server and thereby enables the use of a wide range of scheduling strategies, for example by considering multi-threaded versus single-threaded operation.

2.2.3 Dealing with user interaction

The previous section presented how the characteristics of redraw jobs enable the scheduler to apply a specially tailored scheduling strategy leading to the prevention of overload situations by design. A similar technique can be applied to handling user input.

Pointer devices such as mouses or tablets sample user input at high rates (e. g., 16K bits per second for PS/2) and generate a flood of motion events during mouse or stylus movements. Each motion event is a spontaneous job that requires event handling in the GUI server. This includes translating device-specific coordinates to screen coordinates, moving the mouse cursor, determining the GUI element under the mouse cursor by traversing meta data, visually changing the GUI element on changed mouse-focus, and the routing of the event to the referred GUI client. Due to the cost of these operations, a steady supply of user input events at such a high rate can induce a high load to the GUI server and its clients. The user, however, is only able to perceive the resulting visual changes at a rate lower than 100 Hz. Consequently, for each perceived GUI state, the GUI server may have undergone intermediate states that are ignored by the user[1] but produce system load.

By turning event handling into a periodic mode of operation, the overhead for handling high-rate user input can be significantly reduced. Analogous to the redraw handling, the first step is the decoupling of job submission (an input device interrupt occurs) and execution (the GUI interprets the event) by introducing a first-in-first-out queue. Each time, an input event is generated by the input device, the interrupt handler enqueues the event into a device-event queue. Therefore, the insertion of device events happens aperiodic but at a known maximum rate, which dictates the required queue size.

At a low rate of 100 Hz, the periodic event-processing thread of the GUI server interprets the batch of device events currently stored in the queue. Due to the characteristics of motion events, the batch contains large sequences of motion-only events that can be merged to only one event by accumulating the motion vectors of successive motion events. Consequently, the resulting number of input events to be executed by the GUI server is bounded by the rate on which the user can supply non-motion events such as button press or release events. Typically, this rate is not higher than 100 Hz such that for each period, the GUI server must handle only a few (empirically ca. 0 to 3) input events that imply only negligible computational costs.

2.3 The DOpE real-time window server

The rationale as described in Section 2.2 is the result of extensive practical experiments using the DOpE real-time window server [28, 29] and the TU Dresden's custom OS called DROPS [37] as a testbed. This section describes the most interesting properties of DOpE and reports on the practical experiences made.

2.3.1 Widgets as server-side client representation

Section 2.2.1 highlighted the need for server-side client representations to enable the GUI server to process redrawing jobs independent from its clients and thereby make the job execution times predictable. The design space for a server-side client representation ranges from pixel-based representations to high-level descriptions of the GUI elements (widgets).

By using a pixel-based representation shared between the GUI client and GUI server, the redraw functions in the GUI server are simple pixel copy operations whereas the GUI client can freely express its visual appearance. This approach is used for example by the Mac OS X Quartz engine and the EROS Window System [67]. The great flexibility for GUI clients and the simplicity of the GUI server, however, comes at the cost of a high memory usage. Each

[1] The high temporal resolution of input events as supplied by pointer devices is required by only a few applications such as paint programs to accurately digitize brush strokes. For such applications, the GUI server should provide the raw stream of input-device events via a dedicated interface.

window requires an equally sized pixel buffer to store the representation, even when the window is fully covered by other windows. Furthermore, this approach requires a tight interplay between the GUI server and its clients for providing visual feedback to user interactions. For example, to highlight the GUI element under the mouse cursor, the GUI server has to provide mouse motion events to the GUI client, which, in turn, determines the GUI element at the mouse position, updates the corresponding part of the pixel buffer, and then notifies the GUI server to refresh the changed pixels on screen.

In contrast, when the GUI server implements the widget set, functionality such as mouse-over focus and window resizing can be handled locally in the GUI server without involving the GUI client. With regard to memory-resource usage, a server-side widget set is significantly more efficient because a typical semantic description of a widget consumes only a few bytes regardless of the actual size on screen. For example, for representing a button widget, the GUI server needs to store only its position, size, state, and the button text, which consumes significantly less memory than the corresponding pixel-based representation. The GUI server produces the pixel-based representation from the internal semantic representation only if the widget is visible on screen and therefore provides a large potential for performance optimizations based on window layout. For example, if a client updates the text of a button, it pushes the new button property to the GUI server, which stores it locally. The transformation to pixels, however, is only performed if the button is not completely covered by other windows. If partly covered, the transformation costs are proportionally related to the visible portion. Further arguments in favour of a server-side widget implementation are fostered consistency and interoperability between GUI clients because the GUI server facilitates one common look and feel for all GUI clients. However, as proven by the GNOME and KDE projects, such properties can be provided by client-side libraries as well.

With DOpE, I explored the design range by providing a fully functional server-side widget set that also facilitates the use of pixel-based client representations by the means of a special widget type. The widget set consists of layout widgets, which organize a number of child widgets according to geometric rules, and leaf widgets, which represent the actual state of the GUI client. Therefore, the representation of each GUI client is a tree of widgets. DOpE provides the following layout widget types:

Window A window consists of standard window controls such as a title bar and resize elements and manages exactly one child widget as its content.

Grid A grid arranges its child widgets in rows and columns. It can determine the size of the rows and columns based on the geometric constraints of its child widgets but also allows for client-defined weighted or fixed sizes.

Container A container enables the GUI client to freely position child widgets via pixel coordinates. It is normally not used by GUI clients but by DOpE internally for arranging window-control elements.

Frame A frame holds one arbitrarily-sized content widget, which can be larger than the frame's dimensions. In this case, the frame provides scrollbars to let the user freely choose the view port on the content widget.

For expressing actual client state, DOpE provides labels, buttons, text entry fields, load displays, numeric scales, and scrollbars as leaf widgets.

DOpE's widget set is designed to enable GUI clients to realize more complex GUI elements by composing these basic widget types. For example, a tree widget can be realized by combining nested grids with leaf widgets. In addition to the already mentioned leaf widgets, DOpE

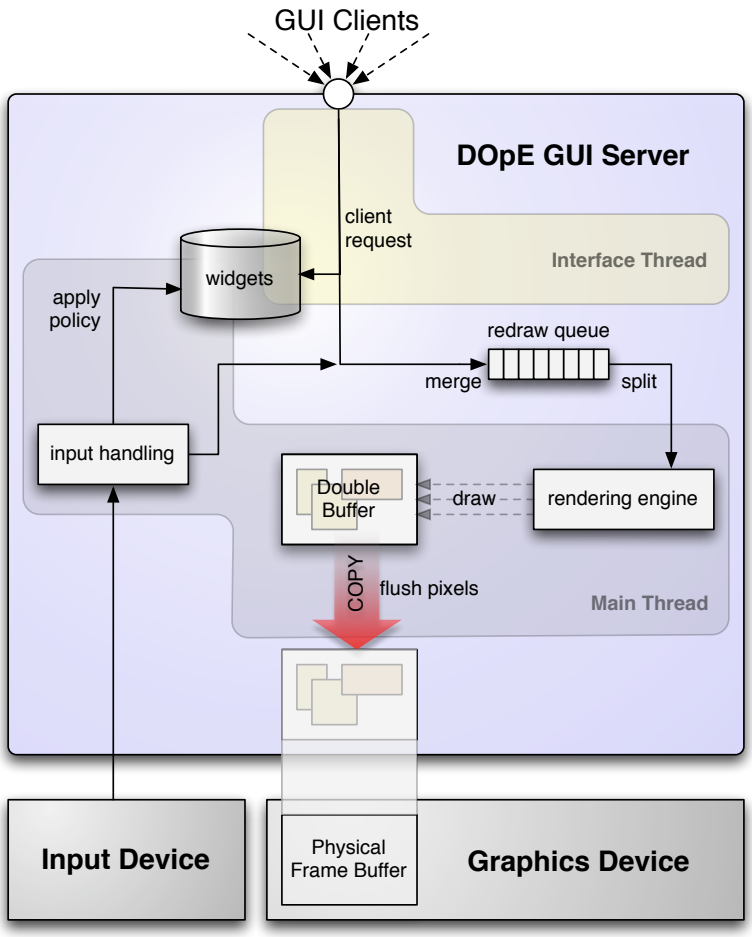


Figure 2.3: Structure of the DOpE GUI server.

provides a widget type called *vscreen* that enables GUI clients to use pixel-based representations shared with DOpE. Each vscreen widget has an associated pixel buffer. The normal mode of operation is that a GUI client writes pixels to the vscreen buffer and then notifies DOpE to update the changed part of the buffer on screen. Additionally, vscreens can be used for continually updating the vscreen buffer at fixed intervals. In this strictly periodic mode of operation, DOpE statically reserves the bus bandwidth required for performing the screen updates and thus guarantees a fixed update rate. The GUI client can synchronize itself to the periodic vscreen updates by collaborating with a synchronization thread in the GUI server created independently for each periodic vscreen widget.

2.3.2 Resource scheduling

Figure 2.3 illustrates the structure of the DOpE GUI server. For its basic operation, DOpE uses two threads of control flow. The *interface thread* serves the client interface for all DOpE clients to enable the creation and management of widgets. The functions of DOpE's client interface perform simple operations on the widget representation and enqueue redraw requests accordingly. Because all functions return immediately and never block, multiple GUI clients can use one and the same communication interface with little interference of each other.

In contrast to the interface thread, the main thread is executed strictly periodic. In each period, this thread handles user-input and processes redraw operations. Because DOpE employs the user input handling as described in Section 2.2.3, user-input handling causes only negligible resource usage. The redraw processing, however, is the primary consumer of both processing time and bus bandwidth and therefore, is subject to local scheduling within DOpE.

Each redraw request undergoes three stages that correspond to different abstraction levels. First, a redraw request is triggered by a client request (e. g., a client places a button into a window) or by user input (e. g., the user moves a window). This request refers to the targeted window and gets enqueued into the redraw queue by applying the redraw-merging technique described in Section 2.2.2. If multiple redraw requests targeting the same window are issued at a high rate, these requests get merged and reside as one request in the redraw queue. Note that redraw requests can get issued asynchronously from the interface thread at any time. At the second stage, the periodic main thread transforms the window referenced by the redraw request to pixels. Enabled by the knowledge of the local representation of the widget tree for any window on screen, DOpE is able to generate the sequence of graphical primitives required to create the pixel-based representation. These graphical primitives are essentially the drawing of scaled images, the drawing of vertical or horizontal lines, and the output of text. DOpE provides these graphical primitives via software rendering that operates on a pixel back buffer in main memory. Its widget rendering engine is designed such that most pixels get touched only once during one transformation. However, there are cases for which one pixel gets subsequently written multiple times. For example, when drawing text on a button, the background of the button is drawn first and then partially overwritten by the textual label. Furthermore, translucent graphical primitives (with alpha blending) require read operations from the pixel buffer to combine the painting color with the background color. During the third stage, the result of the transformation gets transferred from the pixel buffer over the I/O bus to the frame buffer of the graphics card and thereby becomes visible on screen.

This three-stage technique is motivated by the following reasoning. Bus transfers are an order of magnitude slower than memory accesses. By transferring only the final result of the transformation over the bus, each pixel is transferred only once, minimizing the bus load. On commodity graphics cards, read operations from the frame buffer are slow. By performing the second stage in host memory, such costly read operations are completely avoided. Thanks to the clipping and optimization techniques of DOpE's widget-rendering engine, the bus transfer of the pixels clearly dominates the overall rendering performance. Consequently, a good approximation for the temporal costs of each redraw request can be derived from the amount of pixel data that must be transferred over the I/O bus. Because this amount correlates with the size of the redraw request, a feasible temporal model for any redraw operation is:

$$processing\ time = \frac{redraw\ request\ size}{bus\ bandwidth} \tag{2.1}$$

DOpE uses this simple model as the basis for redraw scheduling, which essentially corresponds to I/O bus scheduling. DOpE determines the bus bandwidth via run-time monitoring of its graphics performance and calibrates its temporal model dynamically using a sliding-means algorithm. This way, DOpE is able to dynamically adapt its redraw scheduling to changing bus loads. A further consequence of the approach is the inherent double buffering of graphical output that completely avoids displaying inconsistent GUI states that occur during the transformation of the widget representation to pixels. Because DOpE performs the handling of the mouse cursor in the second stage, the visible mouse cursor moves always smoothly at the fixed rate of the main thread and is free from flickering artifacts. No hardware mouse cursor is needed.

Beside those overly positive properties, employing pure software rendering discards the opportunity to use hardware-accelerated graphical primitives for transforming widget representations to pixels. The alternative of moving the second stage of the redraw processing from main memory to local memory of the graphics card and letting the graphics card's GPU perform the transformation was elaborated in [72]. In the course of this work, DOpE's graphics back end had been implemented for the ATI Radeon 7500 and Matrox G450 graphics cards. As the drawing of scaled images is the most performance-critical operation, the attempt was made to find a temporal model for this operation, which takes the arguments of the operation and the hardware-clipping conditions into account.

Although the implementation of DOpE's graphics back end for the specific graphics devices improved the graphics performance of DOpE dramatically, this approach is a dead end with regard to enabling GUI clients to exploit hardware-accelerated graphics in parallel with DOpE. Chapter 5 addresses this conflict.

2.3.3 Advanced features

In addition to the previously described redraw scheduling, DOpE's local widget representation enables the effective implementation of advanced features such as partially translucent windows, drop shadows, and arbitrarily-sized windows while maintaining bounded worst-case redraw processing time.

A straight implementation of such features would employ the painter's algorithm by drawing windows from back to front and properly incorporating each window's translucency values for painting pixels (alpha blending). Therefore, the processing time for such a redraw operation would correlate with the number of overlapping windows and is unbounded. DOpE's redraw engine functions differently by prepending the actual redraw operation with a geometric visibility analysis. For each pixel on screen, DOpE can determine the front-most window that contributes to its color value. Based on this information, it subdivides each redraw request into a set of fully exposed window areas and propagates a redraw request to each of these windows. The window, in turn, decides if the background of the window contributes to the window's pixel (alpha value is smaller than 1.0). If so, the window first issues a redraw operation for its used screen area to the windows that are visible through it as part of the window's background and then paints its foreground. Consequently, the redraw engine always paints from front to back and lets the actual widget for each layer decide to process another background layer (if the widget is at least partially translucent) or not (if the widget is opaque) before applying its foreground colors. As a consequence of this strategy, the costs of processing a redraw request comprising a number of translucent layers depends on the policy of each incorporated widget but it can also be bounded by limiting the recursion-depth of the background redraw processing. Imposing such a limit results in depth-limited translucency and bounded redrawing costs. Figure 2.4 displays the result of the depth-limited translucency algorithm for a limit of two translucent layers.

2.3.4 Evaluation

This section evaluates the design of the DOpE window server with regard to the chosen server-side client representation and to its resource-scheduling approach. It condenses the lessons learned and summarizes further observations made through my practical experiments.

Effectiveness of DOpE's resource scheduling DOpE obtains the parameters of its temporal model for the prediction of redraw-job execution times from runtime monitoring of its

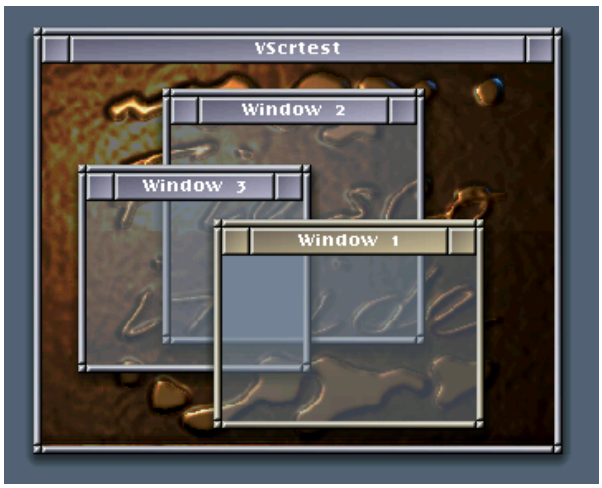


Figure 2.4: Screenshot of DOpE with enabled depth-limited translucency.

drawing performance. As presented in Section 2.3.2, the use of software rendering with the data path to the graphics device as the most significant performance constraint suggests a proportional relationship between the number of pixels to redraw and the redraw-execution time. In [57], this claim was thoroughly analyzed as a real-life use case for the Ferret runtime monitoring framework.

The key finding of this analysis is the nonuniform pixel throughput to the graphics device on varying widths and heights of redraw operations as displayed in Figure 2.5. First, the CPU caches improve the performance of redraw operations that span small address ranges, in particular, redraw request with a small height. Second, the computational per-line overhead of the pixel transfer loop becomes a significant contributor to the transfer costs for redraw requests with a small width. Although these anomalies hint at possible refinements of the temporal model, in practice, the simple model turned out to be effective for taking scheduling decisions. With a typical workload, the corner cases of thin redraw operations occur mostly combined with bigger redraw jobs and do not dominate the overall performance. This behaviour is fostered by DOpE's redraw merging technique. Furthermore, DOpE steadily adapts its temporal model to the runtime-measured pixel throughput by using a sliding-means algorithm. When a corner case dominates the redraw processing for a longer time, the temporal model gets adjusted accordingly. As researched in [72], for hardware rendering, the temporal models for drawing operations become nontrivial and show a large variance among different graphics devices. Even on a single device, the rendering performance shows a high variance depending on a large number of parameters such as pixel alignment, the hardware clipping range, and texturing attributes. Without the deep insight into the inner functioning of the GPUs of modern graphics cards, the construction of realistic temporal models becomes almost impossible.

The overall responsiveness, performance, and output latency of the user interface as perceived by the user are dictated by the parameters of DOpE's strictly periodic main thread. To guarantee the aliasing-free output of streaming pixel data at typical frame rates such as 25 Hz, 30 Hz, and 50 Hz, the Nyquist-Shannon sampling theorem sets forth to sample the client representation at a frequency greater than twice the signal bandwidth (the maximum rate at which the client updates its GUI representation). Consequently, a period length of 10 milliseconds for DOpE's main thread turned out to satisfy this requirement.

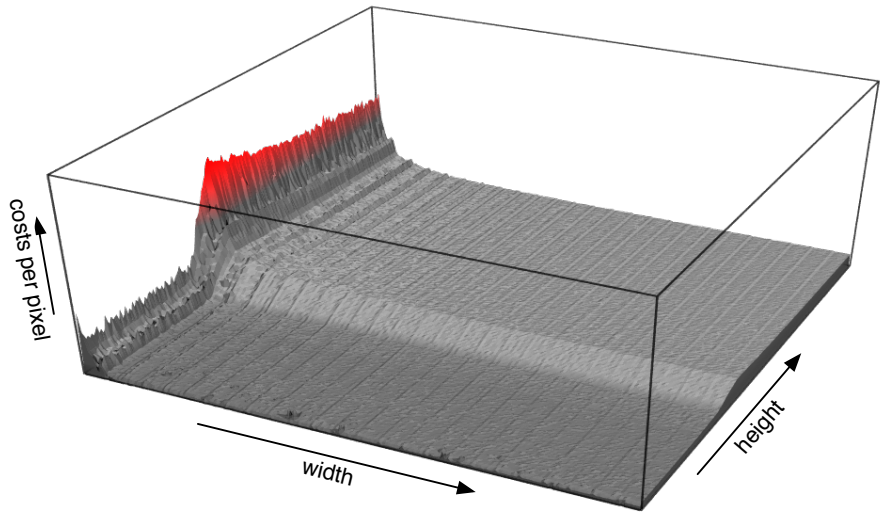


Figure 2.5: DOpE's pixel throughput is a nonconstant function of the width and height of redraw operations. The data set is taken from [57] by courtesy of Martin Pohlack. It displays the costs for the bus transfer per pixel for redraw requests ranging from 1x1 to 400x400 pixels. The measurement was performed on an Intel Pentium-Pro machine clocked at 200MHz with 8 KByte first-level and 256 KByte second-level cache (32 bytes per cache line). More technical details and the measurement results for more recent machines are published in [57].

The original implementation of DOpE provided a special treatment of streaming pixel data as hard real time jobs by pre-allocating time slots for specialized real-time widgets that provided synchronization messages to the real-time client. In practical experiments with the video-streaming application called Verner [61] however, this feature was discarded. Most streaming videos use their audio track as primary time source. The coordination of the audio timing and the GUI synchronization messages as provided by DOpE's hard real-time widgets turned out to be overly complicated. Second, performing GUI client updates on demand via DOpE's asynchronous client interface rather than using a negotiated fixed frame rate of a hard real-time widget turned out to provide a lower output latency for the common case whereas the worst-case latency remains still lower than 15 milliseconds[2] on a two years old machine with a Celeron D (Prescott) CPU at 2933 MHz, 256KB L2 cache, 512MB of memory, and Intel 915G graphics[3]. With enabled drop shadows and two levels of translucent windows, the worst-case latency (each pixel is a composition of three colors) increases to 30 milliseconds. Concluding from this experience, the special support of real-time clients operating at fixed frame rates did not pay off. Instead, DOpE's asynchronous client interface provides feasible output latencies for each GUI client at the same time without requiring knowledge of the client's temporal profiles and without executing a client admission protocol.

Feasibility of server-side widgets The decision of using widgets as server-side client representation was taken to broaden my experimental playground as much as possible and thus, to enable the exploration of a number of GUI-server-related problems beyond quality of ser-

[2] The worst-case latency is the sum of the period length of DOpE's main thread and the time needed to perform the redraw of the complete screen.

[3] The frame buffer of 1024x768 pixels at 16bit color depth is mapped as cacheable memory with enabled write combining.

vice. One particular field of interest was the application of a domain-specific language as client API for a GUI server. In my previous work on GUIs, I observed that the use of high-level script languages such as Tcl/Tk [6] can drastically reduce the GUI-related code complexity compared to the use of binary interfaces. Using such a high-level abstraction as client interface of a GUI server raised interesting questions regarding the server-side performance overhead on parsing textual commands, the costs of communicating textual strings instead of binary data between client and server, the complexity of the server-side support code, and the utility value and convenience of a language as API.

DOpE clients communicate with the DOpE server by using textual commands as illustrated in the following example:

```
grid = new Grid()
button_1 = new Button(-text "OK")
button_2 = new Button(-text "Cancel")
g.place(button_1, -row 1 -column 1)
g.place(button_2, -row 1 -column 2)
win = new Window(-content grid)
win.open()
```

This example code creates a window presenting two buttons arranged horizontally within a grid layout. Note that the GUI-describing code is principally generic and does not contain resolution-dependent physical pixel values, font parameters, or style attributes. It is up to the DOpE GUI server to translate this raw semantic description of widgets and their topology to physical pixels in a way that fits the target device and the needs of the user best, for example by adhering the look and feel as configured by the user. Each command is handled by DOpE as a nonblocking atomic operation that returns immediately with either an error code or a success indication. In contrast to the approach taken by Tcl/Tk, which uses a turing-complete and powerful script language, the DOpE command language does not feature primitives for handling control flow or conditional execution. This design facilitates DOpE's simple program logic with regard to its client API and keeps the complexity of the server-side command interpreter and the widget-support code for the textual commands at less than 1,500 lines of source code (SLOC) [5]. Unlike the X protocol, which transports a potentially high number of graphical primitives and pixel data from the client to the server on each redraw operation, DOpE's client API requires only few messages to communicate the client representation to the GUI server and effectively decouples the client and the server for the most of the time. In the extensive use of DOpE within our group during the past five years, the communication and parsing costs turned out to be negligible compared with costs of the implicated drawing operations. Furthermore, a detailed analysis of the parsing overhead of DOpE's textual command interface in [60] points out that the string parsing costs are in the same range as the IDL stub costs. The alternative use of a binary interface instead of the textual interface would gain only marginal performance but would sacrifice the expressiveness and easy extensibility of the command language. On the other hand, the convenience of using a dedicated command language for the interaction with the GUI server remains unclear and depends on the preference of each programmer. Whereas several client developers were fond of the rapid prototyping and debugging capabilities of this approach and valued the flexibility of tag-value arguments for configuring widgets, others criticised the use of two nested languages with different syntaxes as counter intuitive, for example C code with inlined DOpE commands such as

```
dope_cmd(app_id, "w = new Window()");
dope_cmd(app_id, "w.open()");
```

I observed users of the DOpE client API to construct C++ wrappers for the command language and take this as an indicator for a lack of convenience.

Beside the exploration of the textual client API for communicating the client-side GUI representation to the GUI server, the presence of server-side widgets opened realms of optimizations for advanced features such as translucency. In contrast to plain pixel-based client representations that are only able to track translucency per window, DOpE is aware of the translucency of each widget. Therefore, DOpE can perform effective optimizations. For example, DOpE identifies redraw requests that refer to window regions that are completely covered by an opaque widget of another window even if other parts of the covering window are transparent. In this case, DOpE is able to discard the redraw request. Furthermore, server-side widgets minimize the coupling between GUI client and GUI server. After creating a window as done in the example above, the further handling of its rearrangement on screen (move, top, resize) and the management of mouse highlighting and keyboard focus are locally handled by DOpE and do not require any interaction with the client. The client gets involved only when an event occurs for which the client signalled interest beforehand. For example, to respond to the activation of a button by the user.

With ca. 6,500 SLOC, the implementation of DOpE's widget set makes up half of the overall source-code complexity of ca. 13,000 SLOC. The widget set provides only fairly basic widget types that are supposed to be combined with each other to form more powerful higher-level widget types implemented in the client. One example is a tree widget that is a composition of several nested grid layout widgets for realizing the tree structure and vscreen widgets for displaying the handles of the tree nodes.

2.4 Related Work on GUI-level Quality of Service

There exists surprisingly sparse work by the real-time community taking the special characteristics of graphics operations on GUIs into account. I consider Artifact [63] as the most significant contribution in this domain. Artifact is a real-time window system built in 1995 on RT Mach. It differentiates between real-time clients and non-real-time clients. Real-time clients can only use graphical operations with known execution times and must provide a client model for the use of these primitives. In the GUI server, real-time and non-real-time requests are processed by independent threads, which concurrently access the frame buffer and are executed at different priorities. Artifact has no server-side knowledge about client representations but immediately reacts upon graphical commands supplied through client-interface invocations. Consequently, the Artifact GUI server cannot perform redraw-dropping techniques. In contrast to Artifact, DOpE does not require temporal models of client behaviour because it performs the transformation from client representations to pixels locally.

As mentioned in Section 2.1, the traditional approach to achieve fluent media playback is the use of hardware overlays that effectively remove the GUI server from the latency-critical data path between the application and the hardware. A generalization of this technique that moves the composition of the screen from multiple independent pixel buffers into the hardware is described in [21]. The proposed hardware modification introduces a programmable per-pixel indirection for pixel-read operations performed by the output unit of the graphics device. For each pixel to be displayed in screen, a *Frame-Selection-Vector* table, exclusively accessible by the GUI server, contains an offset value to be added to the current pixel address when outputting the corresponding pixel. With such a hardware in place, the scheduling policy of the underlying operating system would be directly applicable to graphics. To my knowledge however, this technique was never implemented.

In 1995, another approach for creating custom display hardware with QoS support was conducted in the context of the Nemesis project. Nemesis [46] is an OS architecture specialized for distributed multimedia applications. Based on the observation that resource accounting and QoS scheduling become extremely hard when shared servers consume resources on behalf of their clients, Nemesis facilitates the replication of typical server-side functionality in each client and to leave only the functionality of low-level resource multiplexing to the server. Ideally, if each client performs all the costly operations by itself and the resource multiplexing in a shared server is cheap, resources such as processing time or bus bandwidth can be accounted per client and QoS scheduling becomes manageable.

The Desk Area Network (DAN) [16] provided an adequate environment for applying this principle of Nemesis. In a DAN, each system device is interfaced to an Asynchronous Transfer Mode (ATM) interconnect. In contrast to bus-based interconnects with variable-sized packets, ATM is based on fixed-size packets called cells. Thereby, bus scheduling can be performed at a fine granularity and at low jitter. DAN exploits these properties for enabling devices to communicate with QoS guarantees. The DAN Framestore [58] is a frame-buffer device that is capable of arbitrating the access to the physical frame buffer for up to 256 stream connections with individual QoS properties. To spatially isolate the different clients on screen, the DAN Framestore performs key-based clipping protection. For each pixel, the device maintains an additional tag value. In contrast to the frame buffer, the tag buffer is only writable by a privileged software component. Each client stream has a unique stream ID. When a client stream issues a write operation to a particular pixel, the DAN Framestore performs the pixel-write operation only if the client ID equals the tag value of the targeted pixel. The mode of operation of the Nemesis Window System [14] is that each client applies arbitrary graphics operations such as drawing of lines, polygons, and text to a local pixmap. Thereby, long-taking graphics operations are executed in the context of the client, which is subject to the scheduler. During these operations, the client uses no shared resources and thus, can be preempted at any time. After finishing its graphics operations, the client flushes the modified pixels from the local pixmap to the frame buffer. Even though, ATM-based interconnects between processor nodes and peripheral devices such as the DAN framestore are able to provide a guaranteed communication bandwidth and thereby facilitate QoS, commodity desktop computers rely on hardly predictable bus architectures. Lacking the notion of stream connections, a frame-buffer device connected to a bus cannot deploy an elegant client-isolation scheme as the DAN Framestore's key-based clipping.

In 2008, N. Manica presented a QoS enhanced version of the X window system [52]. The work was motivated by the observation that overload situations imply a uniform performance degradation of all X clients, including applications such as movie players that are expected to provide quality of service. The X server performs costly operations on behalf of its clients. When a client issues such an operation to the X server, the information about the client's priority is discarded. Priority inversion occurs when a high-priority client issues a request and the X server is busy with processing a long-taking operation on behalf of a low-priority client. The high-priority request gets further deferred by the default policy of the X server: To minimize the visibility of intermediate drawing states on screen, the X server processes all incoming requests from one client before processing its other clients. This results in an unbounded priority inversion. N. Manica's approach is replacing the default policy by a custom scheduler that takes client-specific QoS parameters into account. The scheduling is based on the concept of the constant bandwidth server (CBS) [12] for which each client is represented by its reservation period, scheduling deadline, the remaining budget of the current period, and the maximum budget. When performing an operation for a client, the X server decreases the client's budgets accordingly. Operations are only issued for clients that have a budget left for the current period. The order of serving requests of those clients is determined

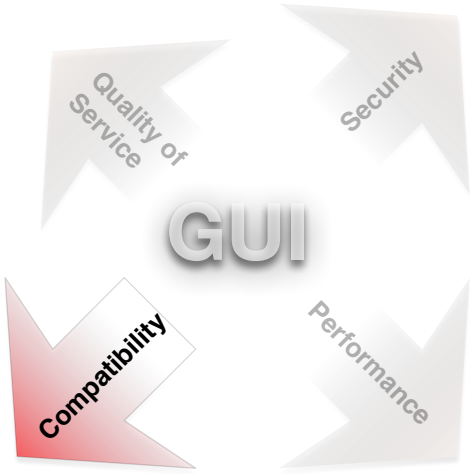
by their scheduling deadlines. At the beginning of a new period, the client's budget gets replenished. For the admission of new clients, the X server performs a simple check against the maximum utilization of 100%. Because once started X operations are not preemptible and their execution times are not known in advance, operations may exceed the client's budget and thereby delay other client's operations. The proposed CBS scheduler accounts the consumed time by incorporating the negative budget into the client's budget replenishment of the next period.

The paper does not consider the duration of operations and states that thanks to the new generation of graphics devices, operations are typically fast enough to enable a sufficiently fine-grained scheduling. The experimental results show a great improvement for executing multiple periodic X clients in parallel, for which the scheduling parameters are well defined. Even when overloading the X server by the execution of `xperf`, the QoS clients preserve their update rates. Effectively, the QoS problem of the X server gets translated to the policy-definition problem of finding the right scheduling parameters for QoS clients. As noted in the paper, determining correct scheduling parameters is not straight forward and gets further complicated by the interaction of the X scheduler with I/O schedulers and the CPU scheduler. In contrast, DOpE's periodic mode of operation combined with its redraw splitting and redraw dropping techniques does not require scheduling parameters to be defined for each QoS client. The quality of service for the entire GUI is a global system parameter that can be adjusted at runtime. The work presented by N. Manica does not consider non-malicious X clients and does not take operations with side effects into account. For example, moving a window as exploited by the code example in Section 2.1.3 implies that other client windows must be updated.

N. Manica argues that implementing a solution for the particular QoS problem from scratch—as I did with DOpE—discards compatibility to existing applications and thereby disqualifies such solutions from real-world usage. Therefore, an evolutionary improvement of an existing and mature GUI server was preferred over a new design. With the following chapter, I recognise the compatibility problem but my favored solution to that problem is different from an evolutionary approach.

Chapter 3

Compatibility



The beginning of the previous chapter presented the difficulty of incorporating changing requirements into a complex existing software design under the constraint of maintaining backward compatibility, which significantly limits the problem-solving space. On the other hand, ignoring the backward-compatibility constraint clears the way for a fresh design that takes the changed design premises fully into account. In the case of delivering quality of service at the GUI level, DOpE's design from scratch turned out to deliver a low-complex and elegant solution for problems, which were considered as virtually impossible to solve purely in software for existing commodity window systems and even facilitated hardware modifications. Regardless of technical advantages however, by sacrificing compatibility, a wide adoption of the fresh design seems to be infeasible because users rely on existing applications.

As the conflict between carrying design legacies for the sake of compatibility and the clarity of software design does not only apply to GUIs but is a general problem on the OS level, several approaches for integrating legacy software into new execution environments have been developed. The approaches differ in the degree of achieved compatibility and in the required engineering costs.

Standard protocols and *binary-level interfaces (ABI)* as relied on by legacy applications can be built into the new execution environment. For example, a VT100-conforming terminal service enables command-line-based applications to be displayed in DOpE windows. In a similar fashion, legacy applications that rely on the pSLIM [41] protocol are supported by a service that translates the pSLIM protocol to a DOpE window. For achieving compatibility with X applications however, this approach is not practical. In contrast to simple protocols such as pSLIM or VT100, the X protocol as one of the most essential GUI-related protocols

is complex and, therefore, costly to reimplement. A characteristic example of applying this approach is conducted by the Wine [8] project, which is an ongoing endeavor to reimplement the Windows OS ABI and libraries to enable the execution of unmodified Windows applications on UNIX. After the tremendous work of 12 years of development and with the support of more than 50 developers, Wine reached the state of a beta version (0.9), which supports a wide range, yet not all, of Window's applications.

With the rise of the free-software ecosystem, providing compatibility on the *source-code level* in the form of compatible application programming interfaces (API) has become a feasible approach for satisfying the user's demands for applications on a new OS platform with the steadily growing pool of open-source applications. On the protocol level, X.org [33] as the most prominent open-source X server implementation, can be ported and adapted to a new OS environment. For example, the adaption of X.org to (the fully POSIX compatible) Mac OS X comprises ca. 12,000 SLOC and, thereby, makes the X protocol natively available to the Mac OS X window system. Even though the engineering cost appears moderate, the actual execution of X clients require the further porting work of complex client libraries. On the application level, several APIs, most prominently Qt, Gtk, WxWidgets, and libSDL, evolved during the past decade as de facto standards and each supports a wide range of applications. The implementations of the above mentioned APIs feature a clean separation of platform-dependent and generic code and, thereby, are easily portable. As described in [34], the port of libSDL to DOpE running on DROPS [37], which features only rudimentary POSIX compatibility, made applications such as Quake natively available on DOpE. In another experiment, the embedded version of Qt was ported to the DROPS environment [71]. Consequently, applications that solely depend on the Qt API can be ported to DROPS by a simple recompile. In practice however, such applications are rare. Most applications rely on further infrastructure such as convenience libraries or the UNIX file-system layout. Whereas the attainable coverage of supported legacy applications is largely constrained by such dependencies, the approach requires a continuing development work for keeping the ported API implementation up-to-date with the upstream version as required by the applications.

By natively supporting protocol stacks, ABIs, or APIs, significant development and maintenance costs are required whereas the achievable compatibility and application coverage remain suboptimal. The complementary approach is to support unmodified legacy software through machine-level compatibility. This approach is widely deployed by the means of remote-desktop protocols (RDP), which make the GUI of a remote physical machine accessible to a workstation over a network connection. Both the remote machine and the workstation may run entirely different operating systems. The simplicity of a RDP such as SLIM [65] enables a low-complex implementation on the workstation side. On the other hand, application compatibility is fully maintained because the original legacy OS is used on the remote machine. These advantages are still preserved when consolidating the remote machine and the workstation into one machine by using a virtual-machine monitor and thereby avoiding the costs of operating multiple physical machines.

Today there exists a wide variety of virtualization products such as VMware [7], Virtual PC, Parallels, and Virtual Box [47] that enable the reuse of unmodified guest operating systems. Furthermore, projects such as L^4Linux [38] and Xen [15] demonstrated near-native performance of slightly modified guest OSes.

Among the presented approaches to providing compatibility to legacy applications, virtual machines (VM) achieve the best level of compatibility at acceptable performance and without the need for extra hardware. In contrast to API/ABI-based approaches, beside the initial engineering costs for providing the virtual-machine implementation, further advances of legacy GUI-based applications and the guest OS imply no further maintenance work be-

cause the hardware interface as relied on by guest OSes and provided by a virtual machine remains rather static.

Section 3.1 introduces the different approaches of user-interaction with virtual machines and describes my proposed technique to seamlessly integrate guest GUIs into the host GUI. Section 3.2 describes two experiments that I conducted to validate the feasibility of the taken approach and to estimate the involved engineering costs. Section 3.3 presents an opportunity to shorten the data path between a guest GUI and the physical frame buffer by using a hardware overlay as provided by commodity graphics devices. Section 3.4 closes the chapter by presenting a timeline of projects employing related techniques.

3.1 User interaction with multiple virtual machines

A VM exposes virtualized standard hardware devices to the guest OS to enable the reuse of guest-OS device drivers for performing input and output. With regard to the GUI, virtual input devices and a virtual frame-buffer device interface the guest OS to the host GUI. The guest window system translates the semantic GUI representation of windows, widgets, and other GUI features of the guest OS to pixels and writes them to the virtual frame buffer. The host OS, in turn, displays the virtual frame buffer of the VM on the host GUI and thus, makes the guest GUI visible to the user. At the host GUI level, the guest GUI is regarded as nothing more than a 2D image, which can be displayed as a virtual console or in a host window. A host GUI that provides virtual consoles time-multiplexes the physical frame buffer to make one of potentially many virtual frame buffers visible at a time. Typically, the user chooses the virtual frame buffer to be displayed by using keyboard shortcuts that are interpreted by the host GUI. This simple mode of user interaction is suitable for users that switch between multiple domains at a low rate, for instance for running a version of the X window system on Linux as host OS while executing one instance of Windows OS within a VM for the sporadic use of a Windows application. For switching domains at a high rate or for tight cross-domain interaction such as transferring data between domains via drag and drop, this approach becomes unsuitable. In addition to the temporal multiplexing of the physical frame buffer as provided by virtual consoles, a host window system is able to spatially multiplex the physical frame buffer to present a (guest) window system in a (host) window system and thus, facilitates better integration of multiple VMs into one GUI. For example, the concurrent output of multiple VMs can be made visible at the same time and data can be transferred between domains via drag and drop. From the usability perspective, such nested window systems are still inconvenient to use because all windows of the guest GUI are displayed at the same window-stacking position of the host GUI. Interacting with only a small guest window pollutes the host-screen real estate with the complete guest window stack and the guest desktop. Furthermore, with a growing number of virtual machines used in parallel, user interaction via both alternatives—virtual consoles and nested window systems—becomes unnatural and inefficient.

3.1.1 Displaying guest windows in host windows

Presenting each guest window in a dedicated host window would overcome these usability problems. The virtual frame buffer as exported by the VM, however, contains no higher-level semantics such as the window layout about the contained pixel data. A reconstruction of these information from outside the guest OS is impossible. An ideal guest OS would export its GUI via a higher-level protocol than a virtual frame buffer, for example by exporting the content of each window in a separate pixel buffer to the host OS. The host OS could then

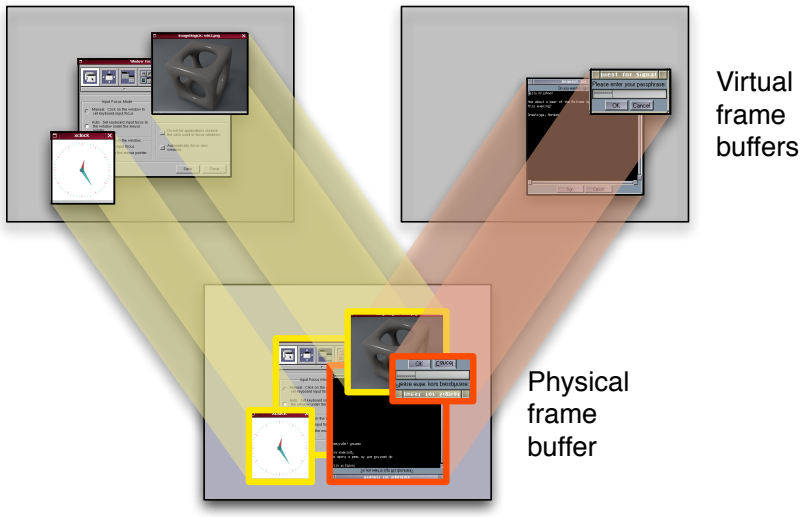


Figure 3.1: Seamless integration of two guest GUIs into one host GUI. Each guest GUI manages its local window stack and performs graphics operations onto a virtual frame buffer. The host GUI uses the window state information of all guests to derive a consistent global host window stack. Each host window displays fragments of the corresponding guest's virtual frame buffer.

properly incorporate these pixel buffers into host windows to provide a seamless integration of the guest GUI into the host GUI. But as legacy OSes are not prepared for such a technique, abandoning the virtual frame buffer in favor of a higher-level protocol for their GUI output is impractical. The way of how windows are managed and how redraw operations are performed widely differs among legacy OSes. Each legacy OS would require a manual and potentially drastic adaption, which effectively locks out proprietary legacy OSes for which the source code is not publicly available. Instead of discarding the virtual frame buffer, the seamless integration of guest windows into the host desktop can alternatively be achieved by supplementing the virtual frame buffer by a simple protocol for exporting the information about the guest OS's window layout.

At the first sight, the generic export of window-state information from arbitrary legacy window systems is complicated because the guest-OS-internal data structures for representing windows may differ significantly and may be inaccessible for proprietary legacy OSes. Keeping the host window stack and the legacy window stack consistent by polling the guest OSes data structures is not feasible. On the other hand, window-state *transitions* such as window movements and stacking-order changes are easy to export by installing a simple tracking program in the legacy OS. By exporting each state transition of the guest window system to the outside of the VM, the host OS is able to construct a model of the guest window configuration that remains consistent with the guest GUI through applying each guest-window state change to the model. The host window system can then use this model to create a host window for each guest window and to display the corresponding parts of the virtual frame buffer within these host windows. Figure 3.1 displays a scenario with two guest GUIs merged into the frame buffer of the host GUI. The virtual frame-buffer representation with the incomplete knowledge of the content of partially covered guest windows is sufficient for transporting the pixel data from the guest GUI to the host GUI because the covered content of a partially covered guest window remains covered also in the host GUI.

Complementary to the propagation of guest-GUI state transitions to the host GUI, host-sided GUI transitions must be applied to the guest GUI consistently. Intuitively, the policy of

the host GUI's window management should be stronger than the guest GUI's policy to protect the user from denial-of-service problems driven by a corrupt guest GUI. When moving a host window, we expect the corresponding guest window to follow. Consequently, bidirectional communication between the host GUI and the guest GUI for propagating window-state transitions is required to preserve the consistency between the guest GUI and the corresponding host windows.

3.1.2 Input handling

For propagating user input from the keyboard and the pointer device into the guest GUI, VMs provide device models for standard hardware interfaces such as PS2 and convert host input events to synthetic interrupts and device-register values. Therefore, unmodified guest-OS drivers for mouse and keyboard can interpret these hardware events and enable the guest OS to manage its local mouse pointer and keyboard focus transparently. When seamlessly integrating the guest GUI into the host GUI, we expect the host mouse pointer and the guest mouse pointer to correspond. Feeding relative mouse events to the guest OS, however, discards the absolute position of the host mouse pointer and the guest OS may nonlinearly transform relative mouse events to the mouse-cursor position, for example by applying mouse-cursor acceleration. Hence, instead of using a mouse protocol supporting only relative motion events, an absolute input protocol such as the Wacom Artpad protocol should be employed for feeding absolute pointer coordinates to the guest OS.

3.2 Feasibility analysis through experiments

Apart from the general considerations expressed by the previous sections, the following technical questions decide upon the feasibility of the approach: How complex is the protocol for communicating window-state transitions and what types of state transitions must be included to maintain consistency between guest GUI and host GUI? Which communication mechanism can be used to adequately transport window-state protocol messages? What are the proper hooks in existing legacy OSes to use for obtaining window-state transitions and for imposing window-state changes into the legacy GUI? How complex are the changes of the legacy GUI? Is the approach practically applicable to proprietary legacy GUIs? To answer these questions, I conducted a series of experiments.

The practical evaluation of the approach requires an experimentation platform that features a host GUI and a facility to execute a legacy OS inside a VM. A Linux OS executing the X window system as host GUI and Qemu as container of a guest OS would suffice. As I steer my research towards highly secure and low-complexity OSes, I selected DROPS [37] with L^{4}Linux [40] as legacy OS and the DOpE host GUI as experimentation platform with the aim of determining the lowest possible OS requirements for applying the described techniques. DROPS is an L4-microkernel-based [49, 43] OS that facilitates the execution of low-complexity native applications alongside L^{4}Linux as a user-level variant of the Linux kernel and enables message-based communication between L^{4}Linux user processes and native DROPS programs via the L4 inter-process-communication (IPC) mechanisms.

3.2.1 The X window system

The GUI integration technique as described in Section 3.1 relies on a virtual frame buffer, virtual input devices, and a hook in the guest-GUI to propagate window-state transitions between the host GUI and the guest GUI. L^{4}Linux provides a virtual frame buffer and virtual input devices such that the X.org [33] server can be executed unmodified.

In the X window system, the arrangement of windows is handled by a window manager, which is a special X client. The X window manager gets notified for each new window to be created within the X session and it implements the policy of displaying and handling window elements such as the title and the handles for resizing and maximizing the window. By replacing the X window manager with a custom implementation, we get full control over the window policy of the X session. For my experiment, I slightly enhanced a version of the AEWM [32] window manager to propagate the following window-state transitions from the X window system to DOpE:

Create window The host GUI gets notified for each guest window. In return, the X window manager receives a handle from the host OS to further reference the created window.

Destroy window The X window manager informs the host GUI about a disappearing guest window by specifying the corresponding window handle.

Place window The X window changes its position on screen. This event must be communicated in both directions because any X client as well as the host GUI may move or resize windows.

Stack window A window can change its stacking position within its local window stack, for example a window comes to front when its content receives a mouse click. The host GUI can also impose a change of the stacking order of windows, for example when the user clicks on a host-window title. Therefore, window-stacking transitions must be communicated in both directions. When reported, the stack-window transition references the new neighbor window in the window stack and specifies whether the new position is in front or behind the referenced window. Furthermore, the foremost and backmost window positions can be specified.

The X window manager implements this simple RPC protocol directly via L4 IPC mechanisms and thereby communicates with the DOpE window server without an indirection over virtual devices. It is implemented in less than 1,500 SLOC. For other virtual machines that do not feature IPC mechanisms, communication over a virtual network device would be a viable alternative. In this case, the host window system could provide its interface via a network proxy connected to a virtual network device of the VM. The required communication bandwidth is negligible because messages contain only geometric coordinates and window handles but no pixel data and the number of messages is bounded by the interactivity of the user. Furthermore, the latency requirement of the message-passing mechanism is low because the transmitted messages only delay the restoration of the consistency between host GUI and guest GUI on rare window rearrangements but not on the frequent user interaction with the window content, which is provided by the virtual frame buffer of the VM. Therefore, message latencies of several milliseconds still yield acceptable accessibility.

To further enhance the integration between the DOpE host GUI and X.org and to maximize graphics performance, I implemented custom X.org drivers for screen output and user input that directly use DOpE's client interface via L4 IPC mechanisms rather than relying on the indirection through the virtual devices as provided by L^{4}Linux. The complexity of these custom drivers is less than 1,000 SLOC.

In the process of integrating the X window system into the DOpE host GUI, neither the X server nor the L^{4}Linux kernel had to be modified. The official X.org device-driver API and the X window-manager concept provided appropriate hooks to achieve the seamless integration at the low engineering costs of developing less than 2,500 SLOC.

3.2.2 The Atari GEM GUI

The integration of the X window system as described in the previous section represents an overly optimistic case because the X window system provides well-established interfaces for obtaining and imposing window-state transitions. To explore the opposite case and challenge the approach with an overly pessimistic case, I experimented with the Digital Research GEM GUI of the Atari TOS operating system, which I consider a legacy GUI in the strictest sense. The GUI is integrated into the OS and the source code of the OS is unavailable. Although the GEM GUI features a window system, the OS does not provide a mechanism analogously to an X window manager for intercepting window operations.

For running GEM inside a VM, I ported the Atari ST emulator Hatari [4] to run natively on the DROPS platform. Hatari uses libSDL [34] as its hardware-abstraction layer and, in turn, emulates the complete hardware of an Atari ST computer including display, mouse, and keyboard.

For passing window-state transitions in and out of the Hatari VM, I enhanced Hatari by a custom virtual hardware device that implements the window-state transition protocol as device transactions on virtual hardware registers. The device model, in turn, uses the L4 IPC mechanism to communicate with the DOpE host GUI. To make GEM use these new virtual hardware facilities, I had to install a small hook of less than 200 lines of assembly code at the GEM system-call interface. As a single-tasking OS at the time of 1986, the OS does not take measures against user programs hooking into the exception vectors of the machine. The custom trap handler monitors the system-call interface for window operations, writes the system call arguments of window operations directly to the custom hardware registers and resumes with executing the original OS code. Furthermore, a timed loop that gets cooperatively scheduled by the OS periodically polls the custom virtual hardware-register interface for window-transitions imposed by the host GUI and applies them to their corresponding GEM window.

This experiment demonstrates that even if appropriate interfaces for installing window-management hooks are not in place, the technique for seamlessly integrating legacy GUIs is applicable at moderate engineering costs.

Figure 3.2 displays the result of the conducted experiments. L^{4}Linux executes Xeyes (left) and Xterm (center) via the X window system while the Hatari VM executes native GEM-based applications (upper left and top). Furthermore, native L4 applications (right and lower left) show off the real-time properties of DOpE by using its native client interface.

3.3 Data path from the guest GUI to the physical frame buffer

Because the rendering in the described experiments is performed by the guest GUI via software routines that operate on a virtual frame buffer, the performance of the guest GUI's graphics routines and the data path from the virtual frame buffer through the host GUI to the physical frame buffer of the graphics device determines the effective output performance.

Figure 3.3 illustrates the data path for using the X window system as guest GUI and DOpE as the host GUI. Thanks to the shared-memory facility provided by the DROPS experimentation platform, both the X server and DOpE can access the same pixel buffer via shared memory such that no pixels must be copied to keep the virtual frame buffer and DOpE's server-side representation consistent. DOpE provides an effective mechanism to display arbitrary portions of one and the same pixel buffer in multiple windows that can be freely positioned on screen. Therefore, no copy operation is required to transform the virtual frame buffer into the snippets displayed by different host windows. Finally, DOpE's redraw engine copies the

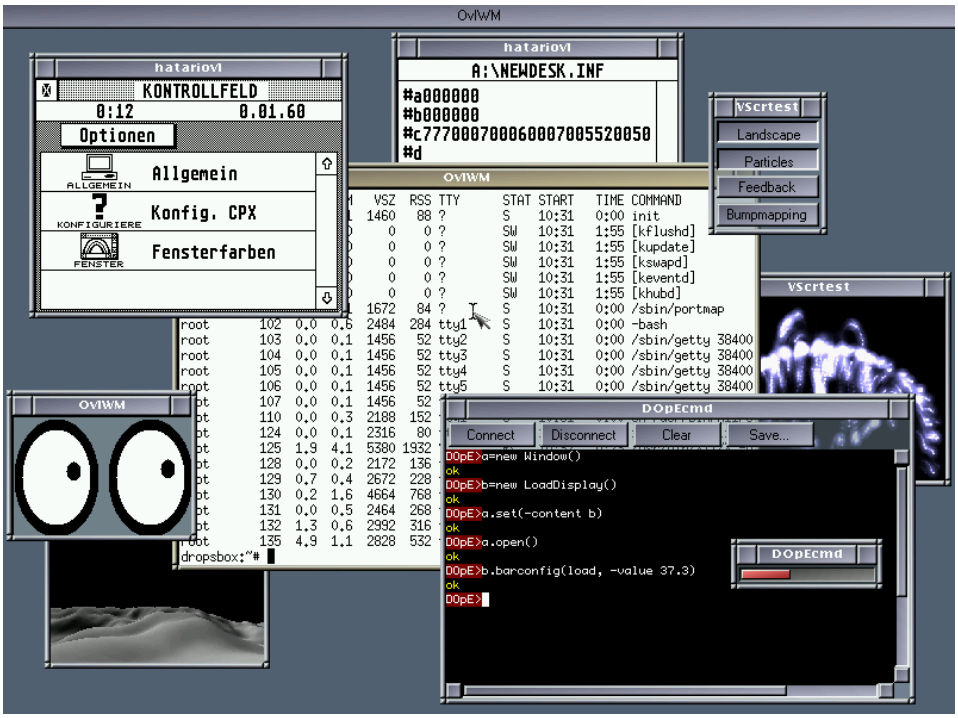


Figure 3.2: GEM and the X window system integrated into DOpE.

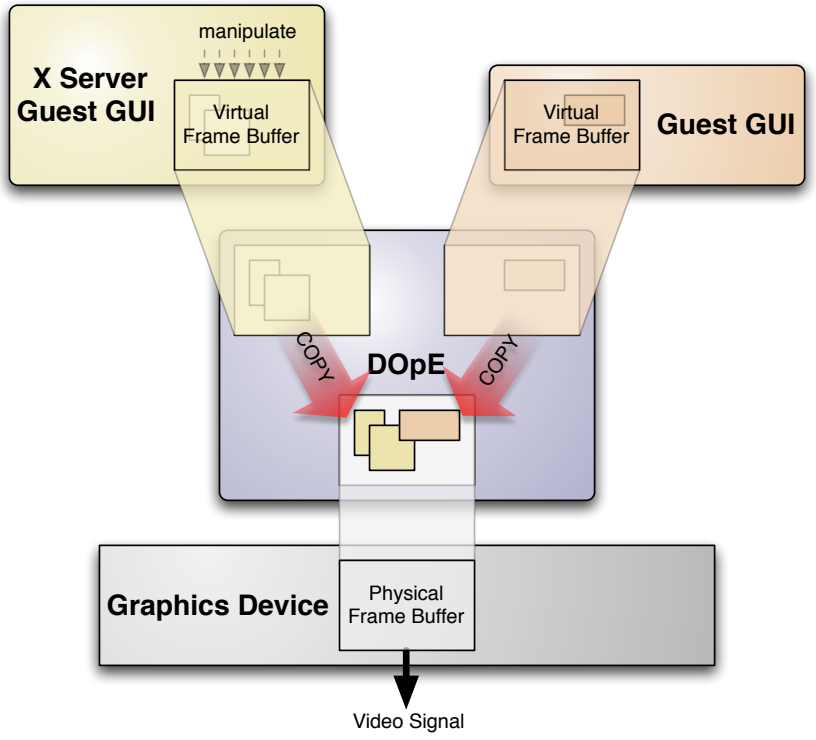


Figure 3.3: Data path from the guest GUI to the physical frame buffer.

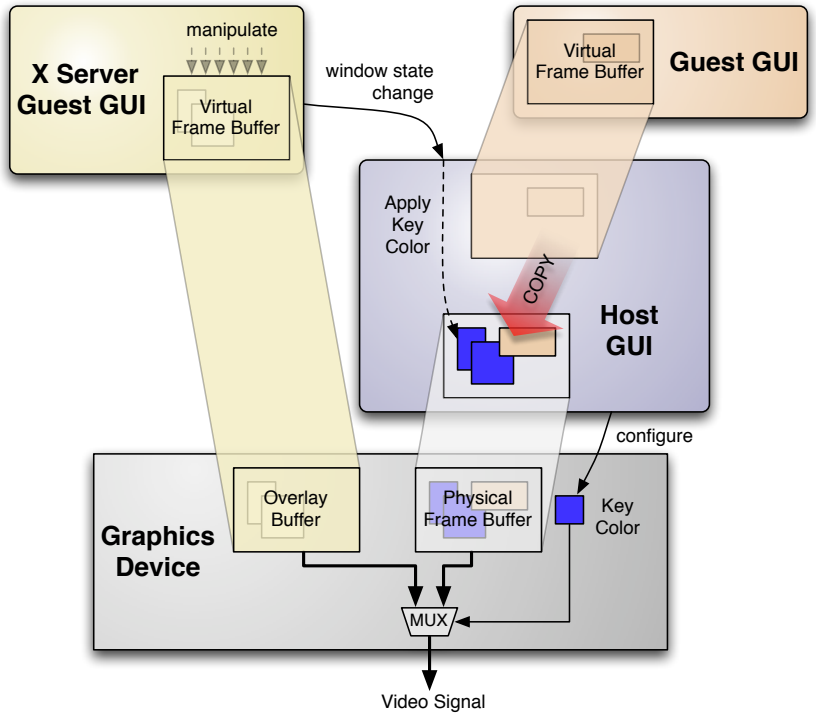


Figure 3.4: Data path if using a hardware overlay for the seamless integration of one guest GUI alongside another virtual-frame-buffer-based guest GUI.

buffered pixels to the physical frame buffer by applying the redraw technique described in Section 2.3.2. As the data path is the same as when following the desktop-in-window approach, the usability benefit of the seamless GUI integration comes at no additional costs.

For the special case of integrating only one guest GUI into a host GUI, the hardware overlay of commodity graphics devices provides the opportunity to eliminate the pixel-copy operation from the buffer to the physical frame buffer. As mentioned in Section 2.1.2, hardware overlays were originally introduced to enable media players to bypass the window system. For each pixel of the physical frame buffer, the graphics device compares the color value against a defined key color and, if equal, displays a pixel from the overlay memory portion instead of the frame-buffer color. If placing the hardware overlay in correspondence to the physical frame buffer, writing the key color to the physical frame buffer makes the corresponding pixel from the overlay buffer visible on screen. The actual per-pixel compositing work is then performed by the hardware.

As illustrated in Figure 3.4, the host GUI exclusively controls the video registers of the graphics device and the physical frame buffer and can therefore control the visibility of the overlay buffer per pixel by writing the key color to each pixel of the physical frame buffer that corresponds to a guest window's content. The host GUI can make the overlay buffer directly accessible to the guest OS as virtual frame buffer. Because the virtual frame buffer resides on the local device memory, the guest GUI could even use the GPU of the graphics device to perform hardware-accelerated graphics operations targeted at the virtual frame buffer. Even though this solution does not scale if the number of guest OSes exceeds the number of hardware overlays, it covers the common case of interacting with one performance-demanding main application while having several other programs with weaker performance-requirements present on screen. Furthermore, this technique supports operating the guest GUI at a different resolution and color depth as the host GUI by exploiting the overlay's

hardware-scaling facility and pixel-format configuration as supported by today's commodity graphics devices.

3.4 Related work on seamless window-system integration

The general approach of seamlessly integrating multiple de-privileged window systems into one screen was invented in 1990 by J. Epstein et al. for the Trusted X (TX) [24, 22] window system. For accommodating existing programs using the X protocol but still providing strong isolation between multiple protection domains, the X server was split into a low-complex trusted back-end part (TX TCB) and an untrusted single-level server such that multiple instances of the single-level server could be executed in parallel on top of the TX TCB. Whereas the TX TCB included the fundamental services for multiplexing the display, routing user input, and enforcing security policy, the single-level server contained the major parts of the high-complex X-protocol implementation. The TX Display Manager provides windows as first-level objects, to which each single-level server supplies the pixel data that corresponds to its local virtual windows. Due to missing shared-memory primitives of the underlying Trusted Mach platform, output operations by the single-level servers onto their virtual frame buffers had to be copied to the TX display manager. The resulting graphics performance and inability to exploit hardware-accelerated graphics operations were identified as weak spots of the TX implementation. In his further work, J. Epstein recommended to move the composition of the visible screen from multiple virtual frame buffers to the graphics hardware [21]. The proposed hardware modification introduces a programmable per-pixel indirection for pixel-read operations performed by the output unit of the graphics device. For each pixel to be displayed on screen, a *Frame-Selection-Vector* table, exclusively accessible by the host GUI, contains an offset value to be added to the current pixel address when outputting the corresponding pixel. In effect, this proposal is a generalization of hardware overlays, for which pixels containing the color key loosely correspond to Frame-Selection Vectors. Despite of the appealing simplicity of J. Epstein's proposal, such a generic facility to spatially multiplex the physical frame buffer has not found its way into the presently available commodity graphics devices.

As presented in [22], the implementation of the TX TCB comprises circa 12,700 LOC. Although the publication does not quantify the engineering costs for the adaption of the MIT-developed X window system to enable its execution as single-level server, it describes these changes to be not invasive. Despite its universality, the TX TCB was never evaluated against GUIs other than the X window system because the primary focus of the project was compliance to the X protocol. However, the results of my experiments suggest that the generalized approach embraces the usage of other legacy GUIs at moderate costs. In [30], I further elaborated on additional application scenarios of the technique, for example, for integrating the Windows OS GUI into the X window system as host GUI by using VMware on Linux.

Since 2001, the seamless integration of multiple windowing systems into one desktop were featured in several products, for which details about the taken approach and the implementation costs are not public. With the release of Apple's Mac OS X, an execution container for classic Mac OS 9 applications was provided. This *Classic Environment* featured the seamless integration of Mac OS 9 windows with their original window style into the Apple Quartz window system of Max OS X. Furthermore, the release of Mac OS X included a modified version of the X.org X window system that is able to seamlessly integrate X windows into the Apple Quartz desktop. The adaptation of X.org to Mac OS X consists of more than 11,000 SLOC, which hints at rather invasive changes of the X server. In 2006, the seamless integration of the Windows GUI into Mac OS X was announced as a new feature of the Parallels VM

software. In 2007, the two competing VM products VMware Fusion [7] and Parallels provide this feature (called Unity and Coherence respectively) and advertise the tight integration of the Windows GUI and the Mac OS X GUI as a breakthrough.

3.5 Lessons learned

With my elaboration on the cost and feasibility of seamlessly integrating legacy GUIs into one host GUI, I found that the combination of virtual-machines with the presented frame-buffer-composition technique provides a practical solution for providing compatibility to legacy GUI-based applications. This approach removes the burden of legacy considerations from the design of the host GUI. The functional requirements posed on the host GUI for accommodating any number of guest GUIs at the same time are surprisingly little. This realization was the driving motivation for the work described in the following chapter. As learned from the conducted experiments, the required modifications of legacy GUIs are marginal, which supports the practical feasibility of the technique with regard to engineering costs. Of these modifications, the hook for enabling the guest GUI to react upon window-state changes of the host GUI turned out to be the only tricky part because this hook requires the guest OS to listen for incoming control commands and a facility for controlling the legacy window system. Both are unnatural additions to the legacy GUI, which—fortunately—can be avoided as explained in the following chapter.

Chapter 4

Kernelizing the Host GUI

4.1 Approaching security

The security of an application is subject to two classes of problems:

1. Protocols that were not designed with a specific attacker model in mind but that are nevertheless used for security-sensitive tasks while potentially being exposed to such attacks

2. Bugs in the application or in system components on which the application relies

Typical for the first class of problems is UNIX that enforces discretionary access control to protect different users of the same LAN from each other while assuming that each user's applications act in the interest of the user. By following this assumption, the attacker model is represented by a malicious user who tries to gain unauthorized access to another user's account. The X window system has built-in protection against this threat by the means of the X authentication protocol that checks the authorization of the user when a user's client connects to an X session. Once permitted access to an X session, each X client can arbitrarily interact with all other X clients of the same session, for example by sharing data over the X clipboard protocol, by painting into another X client's window, by killing another X client (`xkill`), by locking the X session by a full-screen window (`xlock`), or by retrieving the user's input to any X client (`xeyes`). By executing untrusted and potentially malicious applications

side by side, the user breaks the premise of the security model. Today, when using UNIX and the X window system for browsing the internet, running scripts fetched from untrusted websites, and executing downloaded code in a browser plug-in, the fundamental assumption that each application acts in the interest of its user has become unrealistic. The extensive undertaking of extending the Linux kernel with mandatory access control as security feature as done by SELinux [50] considers the presence of malicious applications. On the GUI level, this work is supplemented by the X Access-Control Extension (XACE) [18, 69, 70]. Despite of the tremendous efforts that went into incorporating mandatory access control into Linux, SELinux and XACE are not widely deployed by commodity Linux distributions. The applied methodology of improving the security of an existing software design by fixing critical holes in interfaces and protocols and adding security features to the system remains to be a fight against symptoms. The cause of the security problems lies in the use of inadequate interfaces and protocols.

4.1.1 Security by design

The complementary approach is the redesign of the system software from the ground up by taking a realistic attacker model into account. In contrast to fighting symptoms, this route has the potential to eliminate the root of weak security. Capability-based operating systems such as EROS [66] facilitate the enforcement of the principle of least privilege throughout the whole OS and can thereby provide strong isolation between applications. In 2004, J. Shapiro introduced the EROS Trusted Window System (EWS) [67], which exploited the inherent strengths of capability-based system design with regard to security to isolate GUI clients. The fresh start and the opportunity to redesign the software accommodates the goal of strong security well but, at the same time, sacrifices compatibility. EROS with EWS support only a dedicated set of applications crafted for this particular platform. The broad range of commodity software remains unavailable.

4.1.2 Application-specific trusted-computing base

In contrast to the problems caused by insufficient protocols that can be tackled by carefully analyzing and refining the weak spots, programming bugs as the second class of problems impose a rather diffuse threat on the applications' security. Regardless of the strength of a software design with regard to security, programming errors in security-critical code do happen and—as the growing market for trading zero-day-exploits suggests—attackers do not hesitate to exploit these errors. The amount of security-critical code differs for each application. In addition to the application's code, the code of each system component that has direct or indirect control over the execution of the application (affecting availability and integrity) or that can access the processed information (affecting confidentiality and integrity) is security-critical. The sum of these components is called trusted computing base (TCB) of this application [1]. For example, the TCB of the Enigmail email-signing application executed on a commodity Linux distribution comprises the following components:

- The Linux kernel, which has control over all physical resources in the system
- All loaded kernel modules such as device drivers, which share the unlimited privileges of the Linux kernel

[1] In the following, TCB refers to software components only. If applied more precisely, the term TCB would also comprise hardware and firmware.

- All processes that are executed as *root* and can therefore load arbitrary code into the kernel, access all files in the system, and control all other running processes. For a typical Linux setup, this applies to all system startup scripts and long running user-level services (daemons) such as `init`, `syslogd`, `acpid`, `inetd`, `powernowd`, `hald`, `cron`, `login`, and `dhclient`.

- The X window system including the X server, the X extensions, and graphics drivers

- The desktop environment such as Kde or Gnome, which includes the window manager, control panels, file manager, and applets

- GUI client libraries as relied on by the majority of GUI applications, for example the Mozilla Thunderbird email program relies on the following shared libraries (as returned by the `ldd` command), which accumulate to more than 10 MB of stripped binary size:

```
/usr/lib/libplds4.so.0d                  /usr/lib/libpango-1.0.so.0
/usr/lib/libplc4.so.0d                   /usr/lib/libcairo.so.2
/usr/lib/libnspr4.so.0d                  /usr/lib/libX11.so.6
/lib/libpthread.so.0                     /usr/lib/libgobject-2.0.so.0
/lib/libdl.so.2                          /usr/lib/libgmodule-2.0.so.0
/usr/lib/libgtk-x11-2.0.so.0             /usr/lib/libglib-2.0.so.0
/usr/lib/libgdk-x11-2.0.so.0             /lib/libm.so.6
/usr/lib/libatk-1.0.so.0                 /usr/lib/libstdc++.so.6
/usr/lib/libgdk_pixbuf-2.0.so.0          /lib/libgcc_s.so.1
/usr/lib/libpangocairo-1.0.so.0          /lib/libc.so.6
/usr/lib/libfontconfig.so.1              /lib/ld-linux.so.2
/usr/lib/libXext.so.6                    /usr/lib/libpangoft2-1.0.so.0
/usr/lib/libXrender.so.1                 /usr/lib/libfreetype.so.6
/usr/lib/libXinerama.so.1                /usr/lib/libz.so.1
/usr/lib/libXi.so.6                      /usr/lib/libexpat.so.1
/usr/lib/libXrandr.so.2                  /usr/lib/libXau.so.6
/usr/lib/libXcursor.so.1                 /usr/lib/libpng12.so.0
/usr/lib/libXfixes.so.3                  /usr/lib/libXdmcp.so.6
```

- All processes executed by the user, which can access and control all the user's files and processes, for example via `strace`, the GNU debugger, or the `kill` command

- All X clients that share the same X session

- The Thunderbird email program

- The Enigmail application and the GNU Privacy Guard

The source-code complexity of Enigmail's TCB stacks up to over ten million lines of code. Software written in leading-edge software-development organizations can be expected to have a defect density of 2 defects per 1,000 SLOC [51]. Following the optimistic assumption that Enigmail's TCB is subjected to equally rigid quality-assurance measures, the confidentiality of data processed by Enigmail is endangered by thousands of potential attack vectors due to programming bugs in its high-complexity TCB. Although there exist multiple approaches for increasing the confidence in the correctness of software such as exhaustive tests, code auditing, static code analysis, and formal verification, those measures are either limited in coverage or they scale badly with increasing complexity. For today's commodity OSes, system complexity is the peril of security that even overshadows the design problems of their protocols and interfaces. The drastic reduction of the TCB complexity is the only effective measure to counter the total number of attack vectors that put application security at risk.

As presented in [39, 45, 42], the functional requirements from an OS for hosting a guest OS on top can be implemented in less than 100,000 SLOC. Such an implementation allows for the concurrent execution of a sand-boxed legacy OS alongside a low-complexity software stack for security-sensitive native applications. In addition, recent hardware enhancements by CPU vendors such as Intel [9], AMD (Pacifica), and SUN [48] facilitate further simplification of the VM monitor implementations. Virtualization combined with the GUI integration techniques as presented in the previous chapter relieve us from the burden of carrying legacies and allow for a fresh start for designing a both VM-capable and secure OS and the host GUI by following the principles of minimalism and least privilege. Thereby, strong security and the potential for large deployment enabled by maintained compatibility are no longer a contradiction.

4.2 Premises for designing the host GUI server

For minimizing the host GUI server, the same principles apply as for constructing microkernels:

Orthogonal and minimal interface Only the functionality that is required for maintaining security or that is fundamental for enabling workload should be implemented in the GUI server and provided to its clients via a simple protocol. Functionality that can be implemented outside the server is excluded from the server.

As a consequence, all further functionalities that are not provided by the server yet required by all clients must be replicated in each client. This does not imply that each client has to bring along a substantial amount of overhead. Shared libraries can provide functionality that is common among multiple clients and must be loaded only once.

Separation of policy and mechanism In existing GUI servers, policy management is a major contributor to complexity. Examples for such policies are the placement strategy for new windows, the management of fonts, the support for different keyboard layouts, and the visual styles of window decorations. Instead of providing such policies, the GUI server should provide a small set of flexible mechanisms that enable the implementation of these policies by de-privileged clients.

To substantiate the design space of the host GUI server, the following sections define the preconditions upon which the design rests, outline the workloads to be supported, and present the scope of the attacks to be encountered.

4.2.1 Preconditions

If deployed on top of an insecure platform, the security properties of the GUI server remain ineffective. To enable the security measures of the GUI server to take effect, the target platform must meet the following preconditions:

The *host OS* must be able to host multiple untrusted guest OSes capsuled in separate protection domains. It must maintain isolation between different protection domains and enable communication between protection domains only when authorized. All physical resources such as memory and processing time must be accountable and controllable for each protection domain. In a client-server scenario where one protection domain serves another, temporary delegation of physical resources from the client to the server must be supported to prevent clients from exhausting server-side resources.

The *boot process* of the host OS must be trustworthy, for example by the means of secure booting [13]. Once started, the base platform must provide the GUI server with reliable labeling information for each client such as the complete secure-boot chain of the client.

The *data path* from input devices to the GUI server must be protected against tampering. An attacker who succeeds in installing a tampered input device equipped with a wiretap renders any attempt for preventing key loggers at the software-level as done by the GUI server ineffective. In an environment where such man-in-the-middle attacks between input devices and the OS are considered critical, the communication between the input devices and the OS must be secured via end-to-end encryption. Vice versa, the data path from the GUI server to the display must be protected in accordance to the confidentiality demands of the user. In practice, such protection is hard to achieve because even if encrypting pixel data end-to-end between the OS and the display, an attacker may sample the electromagnetic signals of the display as they are receivable by an antenna.

In the context designing the host GUI server, these properties are considered as preconditions for achieving its security properties and, in the following, are not further regarded.

4.2.2 Workloads

The functional requirements of the host GUI server are dictated by two classes of workloads:

Seamlessly integrated legacy GUIs The client interface of the GUI server must support the primitives required by the seamless GUI integration technique described in Chapter 3.

Low-complexity native GUI applications In addition to accommodating guest OSes and their legacy GUIs, we strive for executing native applications that rely only on a minimally-complex TCB provided by the secure host OS and thereby provide crucial security functions. Following the principle of minimal complexity, the client interface of the GUI server naturally tends to be spartan. The simplistic client interface, however, must support native GUI applications in a way that prevents a complexity explosion in the GUI client caused by need to bridge the functional gap between high-level and convenient GUI elements and the raw interface of the GUI server.

4.2.3 Attacker model to defy

The GUI server is shared by multiple protection domains. While one protection domain performs a security-critical function, other protection domains may execute malicious code such as the following examples:

Trojan Horses By imitating trusted applications familiar to the user, Trojan Horses attempt to wrest sensitive information such as banking-account credentials from the user. Although Trojan Horses cannot be prevented from attempting such an attack, they can be uncovered. The GUI server has to provide a mechanism to assist the user to defeat such attacks by providing a reliable way to authenticate the GUI client that is presented on screen.

Spyware In contrast to Trojan Horses that interact with the user, spyware such as key loggers remain passive but leak key strokes and mouse input supplied by the user to the attacker. Spyware corresponds to wiretapping performed through software.

Information burglars and prying eyes By observing the GUI of a sensitive application, an attacker may seize critical information from the user. For example, a small program

installed by fraud may sample the user's actions by taking screen shots in a periodic manner and transmitting the images to the attacker.

Traitors Untrusted GUI clients that are supposed to exist isolated from each other, must not be able to establish a hidden communication channel through the GUI server and thereby circumvent OS policy. The host GUI server must ensure that any exchange of information at the GUI level is properly authorized by the user and compliant to the information-flow policy dictated by the host OS.

Denial of service Defective or malicious GUI clients may attempt to infinitely grab the mouse pointer, open a full-screen window that captures all input events to render the user interface inaccessible, or impose an overload situation onto the GUI server. Furthermore, if allocating resources on client requests, the GUI server may put itself at the mercy of its clients to not exhaust the available resources. Consequently, when the user relies on high availability of GUI-based applications, malicious or defective applications become an unbearable risk that must be encountered by the GUI server through defensive measures.

Therefore, the GUI server must be designed such that the following security properties are in place. By default, GUI clients must be isolated from each other to prevent unwanted information flow between them by exploiting the GUI server's client API[2]. The GUI server must enable to user to clearly identify each GUI client he is interacting with such that Trojan Horses can be uncovered. Furthermore, the design must guarantee fairness between the GUI clients with regard to the use of shared physical resources.

4.3 Design

This section describes my proposal of the mechanisms composing a minimal-complexity host GUI server.

4.3.1 Client-side window handling

High complexity of today's GUI-based applications is required to manage *widgets*, which are the basic building blocks of a GUI. Widget toolkits such as Gtk and Qt offer a large variety of widgets (e. g., cascaded menus, trees, multi-column lists) and powerful mechanisms for widget layout. This comes at the cost of high complexity, for example the Qt toolkit consists of more than 300,000 lines of C++ code[3]. The wxWidgets toolkit including the back end for the X window system consists of 150,000 lines of C++ code.

There are window systems that implement widget handling on the server side, for example DOpE, which exploits the global knowledge of the widget structure of all its client to perform global optimizations of the rendering process. In contrast, the authors of EWS [67] favor a client-side implementation of the widget toolkit to keep the complexity of the GUI server low. As widget toolkits are not supposed to enforce security policies but are solely used for improving the convenience of developing GUI applications, they are not mandatory as an enabler for workload. Consequently, EWS only provides windows but no buttons, menus and other widgets. In DOpE however, a window is implemented as a widget, which raises

[2]With isolation, I refer to API-based information flow but not to covert channels such as timing channels exploiting memory caches. Covert-channel analysis is beyond the scope of my work.

[3]In addition to being a widget toolkit, Qt is a complete OS abstraction layer with support for networking, scripting, and database access

the question of why not to implement window handling on the client side as well. Should a window enforce a security policy and provide means to protect availability?

J. Shapiro [67] answers the latter question with yes. Clients should not decide by themselves where they are placed on screen and therefore, are not able to arbitrarily cover other clients. On the other hand, a user may expect a client to behave exactly like this and to place its windows in a special way. It does not seem feasible to lock out those clients. The window system has no information about what behavior a user expects from a particular client. Only the user, not the window system, can classify misbehaving applications. To protect availability against malicious clients, the user needs a mechanism to freeze and lock out a client at any time. The policies of window placement, window stacking, and window decoration are no security mechanisms and therefore should not be attributed to the server. Client-side window handling is a key point for achieving exceptionally low complexity of the GUI server.

Note that the X window system provides the concept of a *window manager*, which is one central client that manages the decorations and policies of all windows of an X session. From the security perspective, the window manager belongs to the X server because it has unlimited control over all clients. In contrast, my usage of the term "client-side window handling" refers to managing GUI client windows by each GUI client itself.

As summarized from my practical experiments with the seamless integration of guest GUIs in Section 3.5, the hooks in the guest GUIs to react upon window-state changes of the host GUI had been the only problem that required tricky and unnatural changes in the guest GUI. By deciding not to impose window policy on the client, the host GUI server does never produce such window-state changes and—as a convenient consequence for the seamless window integration—the need for unnatural adaptations of guest GUI vanishes.

4.3.2 Buffers and views

As stated in Section 4.2.2, the support for seamlessly integrating guest GUIs into the host GUI is considered crucial. The minimal mechanism for such support is based on two kinds of objects in the host GUI server: *buffers* and *views*.

A buffer is a memory region that holds two-dimensional pixel data. The memory region is provided by the client and imported into the GUI server via shared memory. The pixel format of each buffer is equal to the pixel format of the current screen mode. Color-space conversions are not performed by the host GUI server because converting color spaces is no security-relevant functionality. Consequently, each client must be aware of the pixel format provided by host GUI server.

The host GUI server has no notion of windows. A window is expected to have window decorations and policies, for example a window can be moved by dragging the window title with the mouse. In contrast, the host GUI server requires only a much simpler object type called *view*. A view is a rectangular area on screen presenting a region of a buffer. Each view has an arbitrary size and position on screen, defined by the client. If the view's size on screen is smaller than its assigned buffer, the client can define the view port on the buffer by specifying a vertical and horizontal offset. There may exist multiple views on one and the same buffer whereas each view can have an individual size and position on screen and can present a different region of the buffer. Each time a client changes the content of a buffer, it notifies the GUI server, which then updates all views that display the specified buffer region. Analogous to conventional windows, views may overlap on screen. A client can define the stacking position of a view by specifying an immediate neighbor in the view stack. Each view can optionally be titled by the client by specifying a text string.

Each client owns private name spaces of the buffers and views it created. No client can access the objects of another client. While each client manages the local stacking order of its

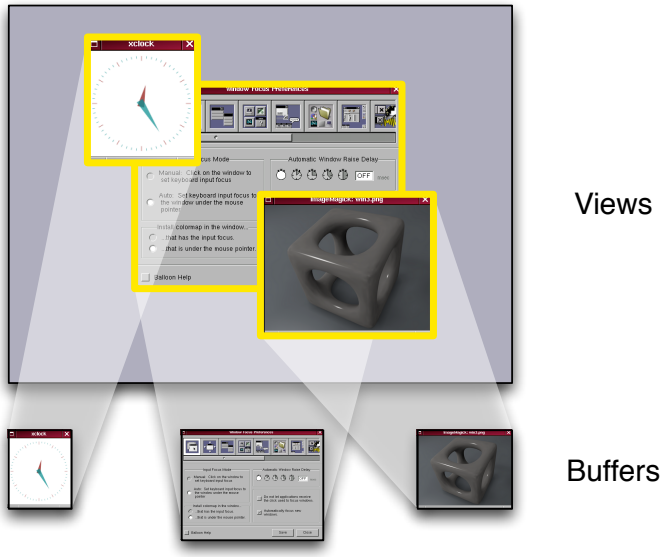


Figure 4.1: One buffer per view. The physical frame buffer shows a composition of views. For each view, there exists a distinct buffer that is rendered by the client.

views, the global stacking order of all views is only known to the GUI server. This fulfills our initial security goal that one client can neither use the GUI-client API to obtain information about other clients nor manipulate other clients.

Based on the mechanisms provided by buffers and views, there are two ways of implementing a window system on top of them.

The straight-forward approach for implementing a window system using buffers and views is to render each window into a dedicated buffer and create one view for displaying the buffer on screen. Figure 4.1 illustrates this approach, which basically corresponds to the mode of operation of EWS and Apple Quartz. The obvious advantage is simplicity. The performance of moving windows and changing the stack layout is great because no re-rendering of windows is needed in such situations. The performance only depends on the blitting operation of the GUI server. Resizing windows, however, implies the need for a buffer reallocation and a new rendering process. Each window requires a buffer of the window's size regardless of whether the window is visible or covered by other windows. The authors of EWS argue that modern graphics cards provide an abundance of memory. On the other hand, graphics memory should be left available to applications instead of being occupied by the window system. Additionally, for mobile platforms and embedded devices, graphics memory is still considered a precious resource.

Another way to deploy the buffers-and-views mechanisms is to apply the seamless GUI integration technique as introduced in Chapter 3. As illustrated in Figure 4.2, this approach uses only one buffer and renders a complete windowed desktop into this buffer. The client is a window system by itself. In the following, we use the term *client window* to entitle a window on a desktop managed and rendered entirely by the client. Instead of using one view to make the whole buffer visible on screen, we create one view for each client window. Each view is positioned exactly to the geometry of its corresponding client window. Consequently, the set of views reveal the part of the buffer that is occupied by the client windows. Furthermore, we keep the stacking order of views consistent with the stacking order of the client windows by applying all state changes of the client windows to their corresponding views, too. For example, when the client raises a client window, it also raises the corresponding view at the

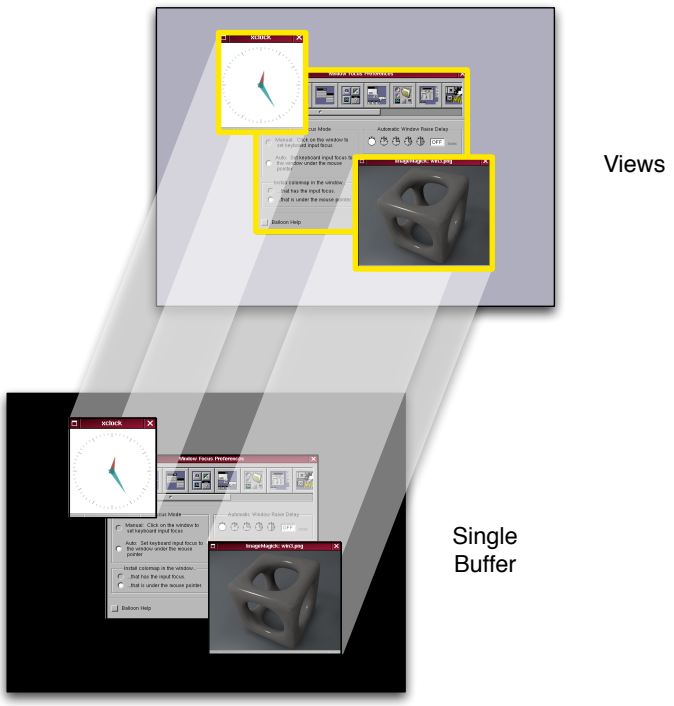


Figure 4.2: Multiple views display one buffer. The client renders a complete windowed desktop into one buffer. For each client window, there exists a view that makes the corresponding area of the buffer visible on screen.

same time. If all *state changes* of client windows are consistently applied to their corresponding views, the stacking layout of the views is equal to the stacking layout of the client window system.

If multiple client window systems are present in a session, each client window system manages its local desktop and its local stack of views. Isolation between clients is preserved because the GUI server alone knows the global stacking order consisting of the interlocked view stacks of all clients. Consequently, each protection domain in the system can implement a custom window system with the desired functionality. With regard to memory consumption, this technique scales well with the number of windows on screen because all windows of one client are using one and the same buffer. On the other hand, moving windows and changing the stacking layout require the client to refresh the affected areas on its local desktop. This makes the client more complex and involves costly rendering operations.

By providing buffers and views as mechanisms, the host GUI server enables the usage of both techniques by different clients at the same time. One GUI client can implement the window handling policy for single windows by itself while another client can be a full-fledged window system that manages a number of sub-clients and thereby provides convenience to application programmers at the cost of increased complexity.

4.3.3 Input handling

The buffers and views mechanism presents clients on screen and lets them communicate to the user. For enabling the secure communication in the other direction—from the user to the client—the GUI server needs to route mouse and keyboard events to the addressed client while hiding the user input from other clients that may execute spyware.

Each client receives input events only if they refer to one of its views. Among all views, there is one *focused view* that represents the keyboard input focus. Only the user selects the focused view by mouse click. No client can define the focused view. The GUI server routes key strokes only to the *focused client*—the client that owns the focused view. The focused view does not need to be the topmost view. It may be completely covered but it still defines the routing of input events.

Input events contain only device-level information. Key strokes are reported as consecutive press and release events supplied with the corresponding hardware scancode. There is no support for high-level information such as the Unicode of a character, the keyboard layout, and the state of modifier keys because such functionality is not required to enforce security. Analogous to the pixel format of buffers, clients must be aware of the meaning of hardware scancodes.

With the exception that a mouse-press event selects a new focused view, mouse buttons are handled like other keys with a defined scan code. Mouse motion and scroll events are reported to the view under the mouse cursor, but only if this view belongs to the focused client. This policy prevents other clients from observing mouse gestures by the user.

If the user moves the mouse while a mouse button is pressed, the GUI server reports all mouse motion events and the finishing mouse release event to the view that received the initial mouse-press event. This clears the way for implementing a rich variety of client-side window-handling policies. For example, if the user enlarges a window by dragging a window resize border, the mouse cursor constantly leaves the view area of this window. Attributing the complete sequence of events including the final release event to the referred window enables the window to catch all events that belong to the resize operation.

There are two magic keys that are exclusively in use by the GUI server and never can be used by clients. Clients do not receive events about these keys. The *Kill* key is used to freeze the current state of the view layout and to let the user pick a client to lock out from the GUI session. It is the emergency brake for a misbehaving client. The other key that I call *X-ray* will be explained in the following Section.

4.3.4 Trusted path

Buffers and views alone are not sufficient to uncover Trojan Horses. The user needs a way to clearly identify the client with which he is interacting. In the following, I address the two problems of what textual information should be used to describe a client and how to present labeling information on screen while keeping the user interface flexible for a broad use.

Commodity window systems such as the X window system let clients choose the text to label a window. This enables nice-behaving clients to be as expressive as possible. For Trojan Horses however, this policy is an ideal opportunity to attack. In multi-level secure systems as addressed by Trusted X [23], labeling information is required to identify the valid classification level in an unforgeable way. On a system with support for secure booting, a trusted loader could provide the labeling information for authenticated clients. We want to support both expressive textual information provided by the client (untrusted label) and unforgeable labeling that represents underlying policies (trusted label). Consequently, a complete label as handled by the GUI server is a concatenation of the trusted label and the untrusted label. Therefore, the first part of the label contains the most sensitive information and is required to be always visible.

Traditionally, labeling information is displayed in window titles. EWS also relies on this way while mentioning that there may be windows without a title at all or a window title may be covered by other windows. In [20], J. Epstein introduced techniques to maximize the visibility of labeling information. One option is to add an additional border that contains

labeling information on all four sides of the window. While this technique is feasible for targeted multi-level secure systems, it consumes precious screen space and limits applications. Windows without the labeling border are not possible by definition.

All the presented label-placement strategies do have one problem in common: A Trojan Horse can mimic a complete desktop by creating a window that is bigger than the whole screen and placing the window in a way that all window controls are outside of the screen area. Such a full-screen window could present a picture of a trusted client, including the faked labeling information. This example illustrates the need to preserve a dedicated screen space for presenting labeling information only. The DOpE window server uses a region at the top of the screen for displaying information about the currently focused window. This area cannot be covered by windows and the information is always visible. However, the top of the screen is not in the focus of the user when he interacts with windows and he may miss to pay attention to the labeling information. A more noticeable presentation of labeling information is desired.

Another idea to preserve a unique capability for presenting labeling information is to cut the color space into two parts. The currently focused client and all labeling information is presented in full color while the brightness of all other clients is dimmed. This guides the user's attention to one bright spot on the screen that displays one clearly visible communication partner at a time. Dimming is implemented in the Exposé function of Mac OS X [1] and in EWS.

I propose a combination of the reserved area and dimming techniques with a novel label placement mechanism that I call *floating labels*. The GUI server dims all views that do not belong to the focused client. All views are surrounded by a thin bright border. The focused view is additionally highlighted by a border of a different color. In contrast to existing label placement strategies, the GUI server analyzes the arrangement of visible views and places all labels in a way that they are visible.

The GUI server chooses the topmost position within the view where the complete label is visible. If the label cannot be completely displayed, it is placed in a way that the first—most important—part of the label remains visible. Labels float over their corresponding view and thereby cover a part of the view's content. By placing the label at a top-most position, the label typically overlaps with the window title of the guest window and does not interfere with the interaction of the user with the work area of the window. All labels are drawn with the color of their corresponding view border and feature a black outline so that they are clearly readable on any background color. Because of the maximum brightness of the label text, a dimmed view can never mimic or manipulate a label because it is doomed to paint gray instead of white. When looking at the screen, the most noticeable information are the view borders, the labels and the focused view. Similar to DOpE, a bar at the top of the screen displays the information about the focused view.

In multi-level-security systems, the GUI server could tint unfocused views of different classification levels with different colors instead of just dimming them. For application areas where more cautious security policies are needed, the dimming may completely blend out the content of unfocused clients.

There are other application areas where high productivity is needed. For example, a user wants to watch a full-color movie while programming. In this scenario, dimming would reduce inspiration and consequently, lower his efficiency. A mechanism to enable the user to toggle two operating modes via a magic key solves this usability issue. In *Flat mode*, no labels, no borders, and no dimming is displayed. The only visible part of the GUI server is a gray shaded bar at the top of the screen that displays the labeling information of the focused view. The gray color of the bar signals that Flat mode is currently active. In *X-ray mode*, dimming, floating labels, and the view borders are active. The bar at the top of the screen is shaded

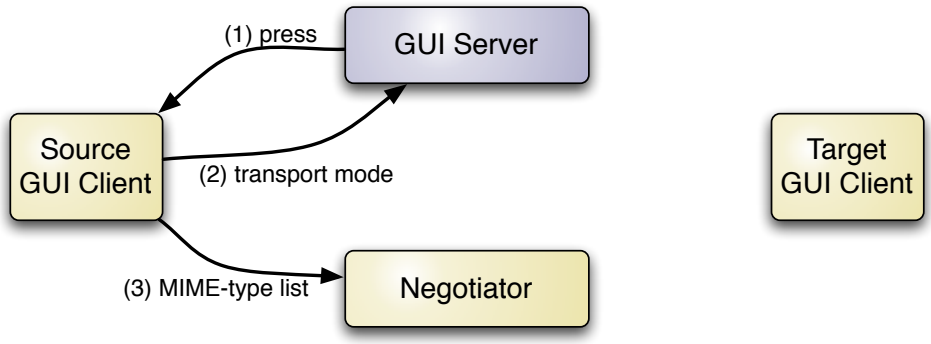


Figure 4.3: Picking an item.

blue, signalling that X-ray mode is active. The toggling between both modes can only be performed by the user. However, clients can request the currently active mode. If a security-sensitive client detects Flat mode, it should ask the user to switch to X-ray mode before it starts processing sensitive data. Passwords should never be entered in Flat mode. For daily use at home or in productive environments, Flat mode may be default and X-ray mode will be used occasionally to perform sensitive tasks, for example bank transactions. In contrast, in highly sensitive environments, switching to Flat mode could be completely disabled.

4.3.5 Drag-and-drop

Drag-and-drop is a widely used paradigm to transfer data from one application to another by dragging an item with the mouse. The GUI server does not need to provide support for drag-and-drop between views of one client. Proprietary drag-and-drop protocols can be used, thanks to the input routing policy described in Section 4.3.3. More challenging is the use of drag-and-drop for establishing communication between different GUI clients and thereby crossing protection-domain boundaries. The drag-end-drop mechanism should establish information flow only when consented by the user and only between protection domains that are permitted to communicate according to system policy. The mere presence of the drag-and-drop mechanism must not enable information flow between arbitrary GUI clients.

In [67], J. Shapiro proposed a drag-and-drop protocol and multi-level-security (MLS) format negotiation for EWS. The proposed solution relies on the capability concept of EROS. It has slight shortcomings such as the lack of user feedback from the target client during the dragging phase. This section presents a refined version of EWS' drag-and-drop protocol.

Communication via drag-and-drop is restricted by the action of the user *and* global policy, for example the permitted information flow in an MLS system. The user expresses his intention by supplying input events to the GUI server. I introduce a dedicated component—the *negotiator*—for representing the global policy.

The drag-and-drop protocol consists of three phases: Picking an item at the source client, dragging the item over the views of potential target clients, and releasing the item at the target client.

Picking an item (Figure 4.3): When the user clicks on a view, only the client knows the meaning of the clicked object. If the selected object is drag-able, the client tells the GUI server about the special meaning of this mouse transaction and the mouse cursor is set to *transport mode*. The client deposits a list of *MIME* types at the negotiator who may filter the list.

Dragging the item (Figure 4.4): While the mouse is moved in transport mode, the user expects feedback from the potential target client. Each time the mouse cursor crosses a view

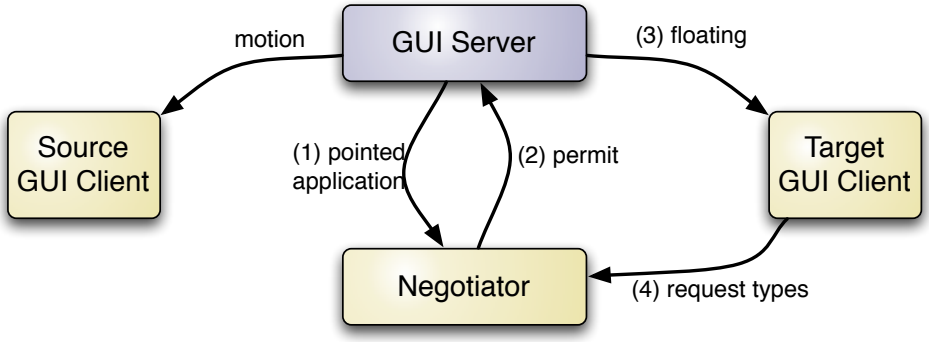


Figure 4.4: Dragging the item.

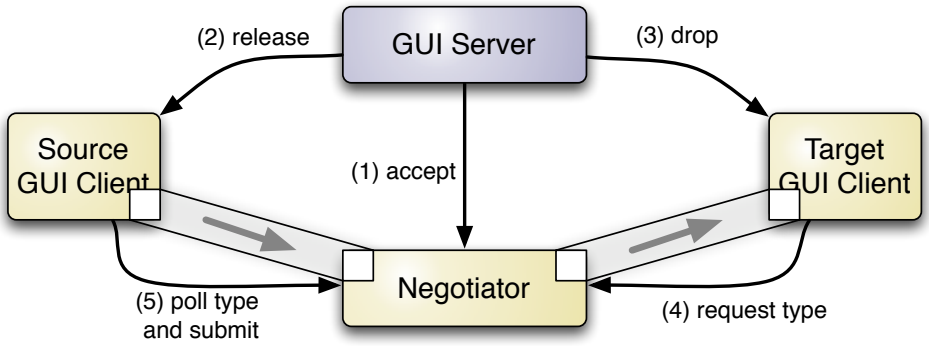


Figure 4.5: Releasing an item.

border, the GUI server tells the *negotiator* about the new *pointed client* (1). In turn, the GUI server receives the policy decision about the information flow from the source to the target client (2). If permitted and the user moves the mouse, the GUI server sends motion events to the source client and *floating* events to the potential target client (3). When a potential target client receives floating events, it can request the offered list of MIME types at the negotiator (4). The negotiator denies the request if the client is not equal to the currently pointed client as told by the GUI server. If the potential target client receives the list of MIME types and a type is supported, it gives feedback to the user.

Releasing the item (Figure 4.5): When the mouse button is released, the GUI server tells the negotiator that the user accepts the transaction (1). Subsequently, the GUI server sends a *release* event to the source client (2) and a *drop* event to the target client (3). The target client can now request one MIME type at the negotiator and supplies a target memory buffer via shared memory (4). When the source client receives the release event, it polls the requested type information at the negotiator and, in turn, transfers a source memory buffer with the payload to the negotiator (5). Now, the negotiator can copy the payload from the source to the target memory buffer and confirm the transaction.

The GUI server has neither to deal with type negotiation, nor does it implement the policy of information flow, and it is not involved in payload transfer. The whole job of the GUI server during a drag-and-drop transaction is to supply input events to both clients and the negotiator. The implementation of the negotiator is highly platform-specific whereas the GUI server's mechanisms are applicable to a wide range of potential target platforms.

There is one low-bit-rate communication channel from the target client to the source client. The target client could encode data in the actual decision of what type from the MIME type list it selects. However, the proposed protocol keeps the involved clients anonymous and the channel is bounded by user action.

Besides drag-and-drop, the most popular mechanism to transfer information among applications is cut-and-paste. In contrast to drag-and-drop, which requires support by the GUI as described previously, cut-and-paste can be implemented aside the GUI server. Clients can directly communicate with a clipboard component that enforces the policy of information flow and performs format negotiation. Therefore, cut-and-paste is not further addressed in detail but the following consideration is important to mention: By employing prevalent clipboard semantics of broadcasting copied information to authorized clients, the flow of information is subjected to information-flow policy between clients in general but not for each copy-paste item separately. Once permitted, information may flow from one client to another at arbitrary times even when the user does not explicitly signals his consent by issuing the corresponding keyboard shortcuts. If an information-flow policy at the granularity of a single copy-paste item is desired, the GUI server has to support the policy decision of the clipboard component with information about copy-paste-related user events. For example, when the user issues the copy shortcut, the GUI server notifies the clipboard that one new clipboard item is expected from the focused client. Analogously, if the user issues the paste shortcut, the GUI server notifies the clipboard component about the user's approval for a paste request by the focused client.

4.3.6 Resource management

A server that allocates resources on request of a client from a fixed pool of resources is vulnerable to denial-of-service attacks. One malicious client can exhaust server-side resources and thereby reduce the quality of service for other clients or even make the service unavailable.

In the case of the GUI server, critical server-side resources are the heap that holds client-specific session information and the processing time that is consumed to serve the redraw of a client.

Dynamic resource limits enabled by heap partitioning To fully maintain the independence of its clients with regard to memory usage, the GUI server has to account for all server-side memory allocations performed on request of each client separately. Furthermore, the GUI server must enforce resource-consumption limits based on the gathered accounting information. For dynamic workloads such as GUI-based applications, statically configured resource limits are unfeasible to administer. If, however, the host OS accounts for the assignment of all physical resources to individual components as stated as one precondition in Section 4.2.1, it is able to support temporary donation of memory from the client to the server. Each time a GUI client requests a service at the GUI server, the client attaches a resource donation to the request. Thus, the resource limits in the GUI server get dynamically adapted to the donations contributed by the GUI clients. Conversely, when a GUI client finishes using the GUI server, the GUI server is expected to release the donated memory resources. To comply with this requirement, the GUI server must store the state of each client session on a backing-store partition that can be released independently from other client sessions. Instead of using one heap to hold anonymous memory allocations, the server creates a *heap partition* for each client and performs client-specific allocations exclusively on the corresponding heap partition. Once a GUI client quits using the GUI server, the GUI server can destroy all local objects belonging to the client's session and release the complete backing store of the client's heap partition.

Redraw scheduling The pivotal realization of Chapter 2 is that—given a fixed quantum of processing time—temporal quality-of-service properties such as redraw performance, latency, and throughput can uphold guarantees for any amount of workload and, thus, are resistant against overload situations imposed by GUI clients. As a consequence, the amount of processing time to be used by the GUI server can be a global system parameter that, effectively, correlates with the temporal quality of service provided by the GUI server to all GUI clients.

4.4 Practical estimation of the achievable minimalism

Without constraining the general applicability, I implemented the presented design to prove its concept, observe its performance, and evaluate the source-code complexity of an actual implementation.

For my intermediate experiment that I called Nitpicker [54], I relied on DROPS as basis platform. Even though DROPS does not fully meet the preconditions as stated in 4.2.1, the availability of L^4Linux as VM container and the infrastructure as provided by the L4 environment provided a suitable experimentation platform.

For handling mouse and keyboard input, I used a port of the input subsystem of the GNU/Linux kernel version 2.6 to L4/Fiasco. The graphical output is realized by using the VESA frame buffer that is provided by the majority of modern graphics cards. The used light-weight software graphics routines consist of functions for drawing rectangles, blitting pixels, and rendering text using a compiled-in font. The rectangle drawing function is used for painting the view borders in X-ray mode. The blitting function supports *solid*, *dimmed*, and *masked* pixel transfer. The masked mode is used for the mouse cursor that is implemented as a special view that stays always on top. Nitpicker performs all graphics operations directly on the physical frame buffer without double buffering.

In addition to Hatari and the X window system (X.org), I also enabled DOpE to be seamlessly integrated into a Nitpicker session. This required supplementing DOpE by merely 160 lines of support code, including the replacement of DOpE's screen and input drivers and the propagation of window placement information to corresponding views. Figure 4.6 displays Nitpicker in X-ray mode with the X.org and DOpE as clients. Note that the translucency effect, which is implemented by DOpE, does not display X windows. DOpE has no access to data of the X session and therefore cannot incorporate X windows into the computation of the translucency effect.

As stated in Section 4.1, I considered minimizing TCB complexity as first-grade design goal. The implementation of Nitpicker consists of merely 1,500 human-written lines of C code (LOC). This amount of code is only a fraction of EWS's size (5,400 LOC) and an order of magnitude smaller than Trusted X (between 12,700 and 30,000 SLOC, depending on which TX components are counted) and X.org ($>$ 80,000 LOC without drivers and extensions).

The prime reason for the small complexity in relation to EWS as the most comparable GUI server is the client-side window handling. Thanks to this design decision, Nitpicker does not need to implement the policy for rearranging windows. This simplifies the internal logic, leads to further simplification of the drawing primitives, and enhances the flexibility of clients, which can implement GUI paradigms such as cascaded menus without special support from Nitpicker. For example, the scroll-able menus of WindowMaker and virtual desktops work with X.org on Nitpicker exactly in the same way as on native X.org.

An interesting side aspect regarding source-code complexity is the considerable amount of generated code that Nitpicker as well as EWS rely on when using IDL for describing the client interface. Whereas the client-interface description of Nitpicker consists of merely 50

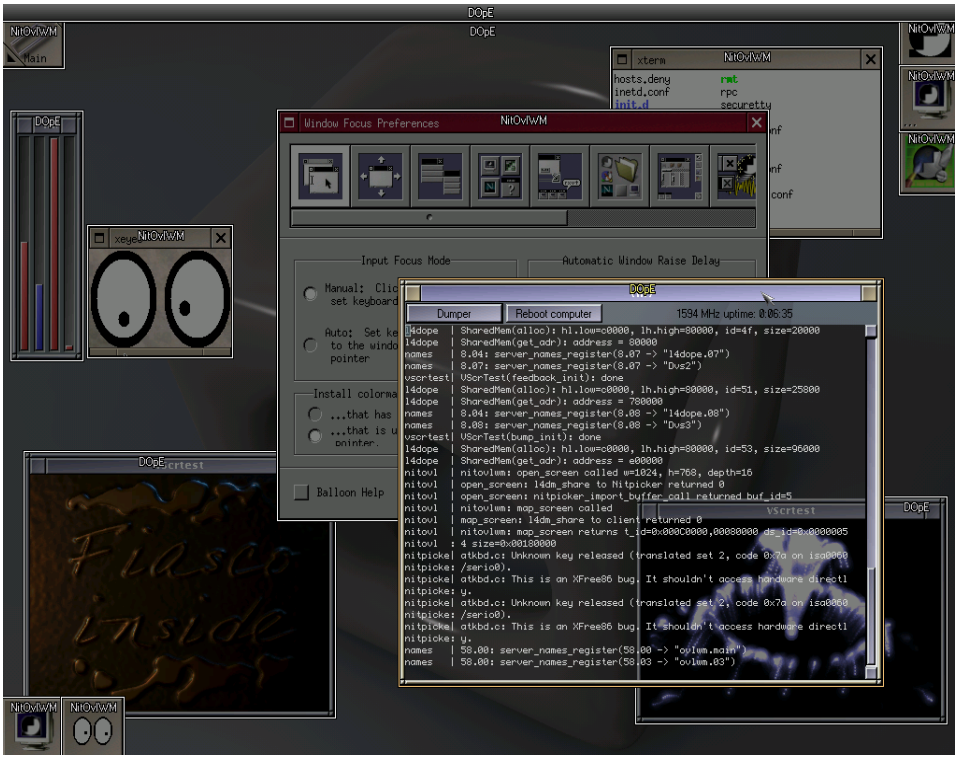


Figure 4.6: Screenshot of Nitpicker.

lines of IDL code, the generated stub code comprises about 1,000 lines of C code. Comparing this to the complexity of the human-written code highlights the critical role of compilers and tools for secure systems. In [27], I addressed this problem by introducing an alternative RPC mechanism that facilitates the elimination of generated stub codes from the TCB.

Intuitively, Nitpicker's low source-code complexity and its spartan and feature-lacking client interface suggests that the achieved low server-side complexity comes at the cost of increased complexity of each native GUI client, which has to provide GUI widgets, graphics functions, font handling, and user-input handling by itself. To pursue this assumption, I created a native Nitpicker client application called Scout (Figure 4.7). Scout is an interactive tutorial browser that displays a multi-page hyper-linked document. The document can contain accentuations, images, nested items, enumerations, verbatim text, and special execute-links for starting external programs. The implementation features advanced details such as anti-aliased fonts, a real-time-generated procedural texture that changes while scrolling, smooth acceleration and deceleration of scrolling with automatic deceleration at the boundaries of a page, images with alpha-channel, translucent icons that distort their background with correct refraction as computed by POV-Ray, drop-shadows, and fading icons and hyper-links on mouse-over. The complete source code of Scout including the graphics functions, the widget set, and the window handling comprises less than 4,000 SLOC. Scout demonstrates that a useful and graphically appealing native Nitpicker client can be realized at low source-code complexity. Thereby, this experiment revises the intuitive assumption that a minimal-complexity server inherently implies high client-side complexity.

I estimated the performance impact on account of the indirection introduced by Nitpicker by comparing the CPU demand of DOpE running as Nitpicker client against native DOpE. In both scenarios, I stressed DOpE by displaying four animations of the size of 320x240 pixels at a rate of 25 frames per second while permanently generating artificial redraw requests

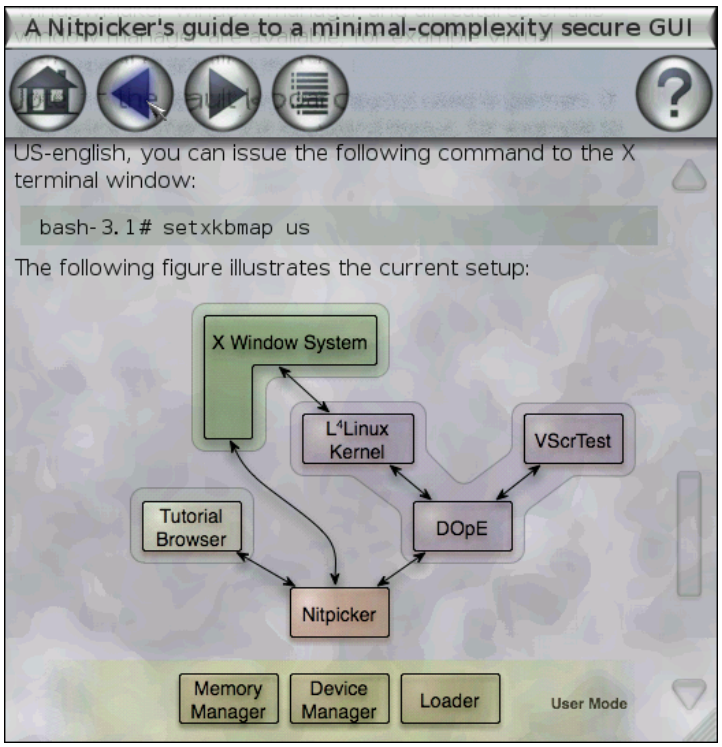


Figure 4.7: The Scout tutorial browser implemented as native Nitpicker application.

for another DOpE window. For the tests, I used an Intel Celeron PC clocked at 900 MHz. Nitpicker does not require additional copying of pixels. I expected DOpE on Nitpicker to perform slightly worse than native DOpE because of two additional context switches for each redraw operation and a computational overhead for traversing Nitpicker's view stack. In X-ray mode, the additional load raises up to 25 percent caused by the *dimmed* blitting function that I have not optimized for performance. When switching to Flat mode, the additional load drops to less than one percent. The observed low overhead matches my previous estimations and supports the feasibility of Nitpicker's design with regard to output performance.

4.5 Intermediate result

When employed on a host OS that provides isolated protection domains, Nitpicker maintains the isolation of its clients to prevent applications from spying on each other by exploiting GUI server functionality. In contrast to today's commodity GUI servers, which expose user input to any application, Nitpicker protects the user from spyware by routing user input to exactly one focused client at a time. Provided an OS that supports secure booting and client authentication, Nitpicker enables the user to clearly identify each client application via a combination of dimming and labeling techniques while preserving the high flexibility of client GUIs. This enables the user to identify and disarm Trojan Horses. Thanks to the low complexity and the deployed resource management, Nitpicker is robust against denial-of-service attacks driven by client applications and can guarantee the service of sensitive client applications with regard to their GUI. The choice of buffers and views as Nitpicker's graphical abstractions was motivated by the seamless window-integration technique described in Chapter 3, which enables the use of existing commodity window systems and their applications alongside low-complexity security-sensitive applications. Because Nitpicker relies

on server-side client representation, which I identified as prerequisite for achieving temporal quality of service, Nitpicker's design is compatible with the periodic execution model, redraw splitting, and redraw merging as introduced in Chapter 2. Hence, the advantages of extremely low source-code complexity, full client isolation, the trusted-path security measure, bounded output latency, and the compatibility to existing GUI applications are consolidated into one GUI-server design.

From a purely functional point of view, this state of the GUI server design fulfills my initial goals. The applicability of the solution in practice, however, depends also on non-functional qualities, foremost the graphics throughput. Nitpicker performs graphical output via software routines, which consume precious CPU time and bus bandwidth. Hardware-accelerated graphical functions as provided by graphics cards operate orders of magnitude faster than software routines and relieve the CPU from handling pixel data. Utilizing hardware acceleration for a GUI server as simple as Nitpicker is almost trivial because the used graphics operations are limited to the blitting of pixel data, the drawing of boxes, and the rendering of simple text strings. Among these operations, blitting is the most performance-critical one because it operates on bulk pixel data. This operation is provided by most hardware-accelerating graphics cards and it is very simple to utilize by software. Therefore, enhancing the GUI server to support hardware-based blitting provides a simple yet effective optimization over purely software-based graphics. On the other hand, hardware-accelerating the graphical operations of the GUI server alone is not sufficient for achieving overall high-performance graphics, in particular 3D graphics. For achieving maximum performance, also the GUI clients must be enabled to exploit the available hardware-graphics capabilities. This raises the problem of how to safely multiplex the physical resources provided by the graphics card between the GUI server and its clients. Chapter 5 addresses this problem.

4.6 Related work on securing GUI servers

The GUI-server design presented in this chapter contributes solutions for three categories of problems that are also addressed by related work: protecting and isolating GUI clients from each other, assuring GUI integrity, and minimizing source-code complexity. This section presents related work for these categories. It does not cover related work on preventing resource-exhaustion-based denial-of-service attacks at the GUI level because—to the best of my knowledge—this problem is not addressed by GUI servers other than Nitpicker.

4.6.1 Protecting and isolating GUI clients

With the creation of Trusted X [23] in 1991, J. Epstein pioneered a GUI architecture that enables mutually untrusted applications to share one user interface. As described in Section 3.4, this solution employs one untrusted X server for each protection domain, which—in the context of Trusted X—corresponds to a security level. Trusted X supports inter-domain communication via copy-and-paste but it subjects these interactions to policy defined via a privileged configuration interface (Trusted Shell). The design of Nitpicker inherits the idea of hosting an entire GUI session as a single client from Trusted X to uphold compatibility to existing applications. In contrast to Trusted X, which reuses the complete X server almost unmodified in the form of untrusted single-server instances, there exist several projects such as XTSol and XC-Security to incorporate security into the X server itself. In 2005, these efforts culminated in the XACE [18, 69, 70] extension with the goal to provide mandatory access control at the fine granularity of a single X client. Before executing a potentially risky operation, the X server calls a central policy manager for permission and supplies information about the subject (the

GUI client), the object (e. g., the affected X resource), and the access type as decision criterion. The X server executes the operation only when approved by the policy manager. Securing the X server with XACE has two facets, the policy-enforcement mechanism and policy definition.

To let the X server enforce access control, it must be enhanced to perform policy requests before doing critical operations by inserting hooks into the code. The starting point is an X server without any hooks, which corresponds to a system based on a default-permit policy. The XACE concept promotes the methodology to successively add hooks to the code and thereby make the system more prohibitive. The places for inserting hooks into the code must be carefully chosen because policy requests require communication to the central policy manager and there is a trade-off between requesting policy decisions at a high rate and performance. To become fully effective for isolating GUI clients, all potentially critical operations within the highly complex source code of the X server must be identified. In contrast, Nitpicker does not perform inter-client communication and does not identify resources via global ID name spaces. Thereby, it keeps its clients fully isolated at all times. If employing a drag-and-drop mechanism as described in Section 4.3.5, the policy is completely encapsulated in the negotiator component.

By adding policy-lookup hooks to the X server, the security problems of the X server are translated to a policy-definition problem. XACE provides 15 different hook types to govern the access to different resources, devices, properties, and extensions. In the X window system, there exist numerous resources, extensions, and properties, which must be known by the policy manager. As learned from SELinux [19], security policies for complex systems become complex as well. Unfortunately, applications partially rely on the overly permissive default policy, for example when using the X clipboard mechanism. Finding a restrictive policy that still upholds application compatibility is challenging.

In contrast to systems based on a central policy, capability-based systems such as EROS [66] implement policies in a decentralized way, employ a default-deny policy and thereby facilitate the principle of least privilege from the start. The EROS Window System (EWS) [67] is the first GUI server specifically tailored for a capability-based system. EWS reduces the number of object types provided by the GUI server to only sessions and windows. Nitpicker shares this simplicity and the orientation toward capability-based systems with EWS. As mentioned throughout this chapter, EWS and Nitpicker address the same problems differently. In contrast to Nitpicker, EWS still contains a substantial amount of built-in policy for arranging and decorating windows. Furthermore, the authors of EWS spent no effort to accommodate existing applications because EROS does not provide execution containers for legacy software. I consider the support for seamlessly integrating legacy window systems into a Nitpicker session as pivotal. Nitpicker's flexible buffer-and-view mechanism described in Section 4.3.2 supports both isolating complete instances of untrusted GUI servers as promoted by Trusted X and fine-grained isolation of native GUI clients as facilitated by XACE and EWS.

4.6.2 Assuring GUI integrity

As explained in Section 4.3.4, protecting the integrity of displayed information on screen is required to counter Trojan Horses. Secure window labeling as a protection measure was first introduced for Compartmented Mode Workstations [62] to make security domains distinguishable on screen. In [20], J. Epstein discusses several labeling techniques employed by Trusted X. At the beginning of the 1990's, monochrome displays and 8-bit color displays had been the state of the art, a dimming technique as implemented by Nitpicker was not applicable. Instead, the proposed labeling mechanisms tried to ensure the visibility of labeling information by rather obtrusively displaying labels not only at the window title but at all sides of each window. Today, with true-color displays being pervasive, dimming techniques

have become feasible. In contrast to the Exposé function for Mac OS X, which uses a dimming effect for usability reasons rather than for security, Gnome's `gksudo` and Windows Vista's User Account Control (UAC) employ dimming to support security features. Gnome's `gksudo` is a GUI-based variant for the `su` command, which is commonly used to execute a process with `root` privileges from a unprivileged user session. To authorize the privilege elevation, `gksudo` dims and freezes the current screen content and presents a password request. On Windows Vista, UAC is used to elevate user privileges and also uses a dimming effect to highlight its security-critical functionality.

However, both solutions suffer from the following two problems. Because the credentials prompt is triggered by the application rather than the user, it may be issued at arbitrary times. Hence, a malicious application can attempt to spoof the security by using a full-screen window that mimics the Credentials Prompt and thereby obtain the password. UAC tries to hinder such attacks by presenting the user name and a customized icon. In contrast, `gksudo` presents only a plain password request. Furthermore, both implementations rely on a global modal dialog that blocks the output of all other application on screen. This modality is not just an inconvenience but an attack vector for denial-of-service because the points in time for UAC requests are controlled by software.

Nitpicker's X-ray mode combines dimming and labeling such that each area on screen is watermarked with the corresponding labeling information. In contrast to UAC and `gksudo`, Nitpicker puts the user into control to decide when the GUI-integrity measure is activated. Thereby, Nitpicker's X-ray mode does not interrupt the user's work flow with a global modal dialog. Compared to UAC and `gksudo`, Nitpicker's X-ray mode is hard to spoof. First, an attacker would need to predict the point in time when the user activates the X-ray mode, which is unlikely. Second, Nitpicker has a reserved screen area for presenting authentic labeling information. The background color of this area indicates when the X-ray mode is active. Because no client can alter the reserved area, a spoofed X-ray mode would be easy to detect. Third, because Nitpicker enforces full isolation of its clients, no client is able to acquire a screen shot, which is a precondition to spoof the dimming effect.

An alternative concept to defy spoofing attacks without the need for GUI-integrity measures is the secure-attention-key mechanism as used by the Windows NT login screen. Rather than asking for the user's credentials, the login screen presents a message asking the user to press the key combination for a system reset. A Trojan Horse implemented as a plain Windows application cannot catch this key combination and would be killed. The login screen, however, is part of the OS and handles this key combination by presenting the real login screen. The only way to circumvent the secure-attention-key mechanism is to boot an alternative OS. Even though it can be circumvented, this security measure is beneficial because, launching an attack via an alternative OS requires more effort than creating a simple Windows application.

To proof the authenticity of the loaded software stack including the login screen, authenticated booting can be combined with remote attestation. The remote attestation of the loaded software stack must be issued by an external device such as a mobile phone that is trusted by the user and features a display to present the result of the attestation process [35].

4.6.3 Minimizing complexity

Section 4.1.2 presented the crucial role of software complexity for security and constituted a minimal-complexity TCB as design goal. The first GUI server that considers TCB complexity as design criterion is Trusted X, which achieves a complexity as low as 12,700 SLOC.

In [46], the creators of Nemesis identified that shared servers inherently imply QoS crosstalk, which complicates QoS scheduling. To minimize QoS crosstalk, shared servers

should generally be avoided. Only for multiplexing physical resources, the authors of Nemesis devise the construction of simple shared servers with low abstraction from the hardware. Even though this argumentation was made in the context of QoS, it applies to the problem of protecting and isolating clients. To minimize the likelihood for unwanted information flow between clients, a shared server such as a GUI server should be as minimal as possible. The Nemesis Window System as mentioned in Section 2.4 pursued the combination of client-side window handling with client isolation to minimize the complexity of the GUI server and thereby prevent QoS crosstalk.

In [67], J. Shapiro quantifies the source-code complexity of EWS with only 4,500 SLOC. This achievement by the authors of EWS and the lessons learned from their work motivated my work on Nitpicker, which takes the ambition to minimize complexity one step further. Nitpicker is comprised of less than 1,500 LOC. This complexity reduction compared to EWS is mostly attributed to the design decision for client-side window handling.

Chapter 5

Hardware-accelerated Graphics

In contrast to the previous chapters, which provided techniques for addressing functional requirements posed on GUIs, maximizing performance is not a functional requirement but an optimization. This optimization, however, is universally regarded as mandatory to enable the GUI workload on today's desktop computers. In effect, the importance of high graphics performance has driven fundamental design decisions of today's GUI servers partially by sacrificing functional aspects that I consider as essential, in particular security.

The bias of the design of today's GUI servers toward maximized performance rather than security results from the historical market evolution for commodity desktop computers. Section 5.1 presents a timeline of the major innovations in the field of hardware-accelerated graphics that determined the feature sets provided by today's graphics devices and exploited by today's GUIs.

Because these commodity GUIs exploit hardware-accelerated graphics, users generally expect to be able to fully utilize the hardware capabilities. The GUI-server implementations that I presented in the previous chapters perform graphical operations by software executed on the CPU. Not considering hardware-accelerated graphics seems to disqualify my proposed design from real-world usage. This chapter addresses this concern by systematizing the existing approaches for hardware-accelerating GUIs according to the compliance to our security goals. For reviewing the existing approaches for utilizing hardware-accelerated graphics, I subordinate performance to the functional requirements compiled by the previous chapters. Rather than approaching the question of how to maximize the utilization of the graphics device, I strive for exploiting hardware capabilities for optimizing performance as far as possible but without violating my functional goals, foremost maintaining security. Section 5.2

translates this rather abstract problem to a technical level by providing an overview over the physical resources and the programming model of today's graphics cards resulting in a concrete problem statement. Section 5.3 presents the range of designs by elaborating on existing work.

An apparent solution is provided by the principle design of the Windows Device Driver Architecture explained in Section 5.4. This architecture, in turn, served as the primary motivation for recent hardware improvements as presented in Section 5.5. Section 5.6 presents how these hardware improvements qualify as apparent solutions for incorporating hardware-accelerated graphics support into the TCB at a low complexity footprint.

5.1 Timeline of hardware-accelerated graphics

The idea of offloading graphical operations from the CPU to specialized hardware was first introduced to the commodity desktop-computer market by the Commodore Amiga 1000 home computer in 1985. The Amiga hardware design was based on the Agnus chip that functioned as a memory-bus arbitrator, a flexible DMA controller, and a video-output device. The DMA controller contained a block image transfer unit (blitter), which provided functions for copying pixel data between memory locations including on-screen memory, filling memory with bit patterns, and drawing of lines. The pixel-copy capabilities were optimized for bit-plane-organized screen modes and therefore provided functions for applying boolean functions and bit-shift operations to the pixel payload. By using the blitter, system software was able to relieve the CPU from long-taking pixel-copy operations and thereby to accelerate the overall graphics output. At the time of release, the Amiga competed against the Apple Macintosh. Thanks to the combination of the blitter with the Copper video unit, the Amiga was able to outperform its competition in terms of graphics capabilities.

From the software perspective, the blitter is accessed via a set of memory-mapped registers. These registers expose the entire state of the blitter including attributes for the current blitting operation and the device status. The programming model for setting up a blitting operation is synchronous, which means that each blitting operation must be issued by the CPU separately. For issuing a blitting operation, the software must

1. Wait until the blitter is ready,

2. Set up the attributes of the operation, which are source and destination memory addresses, the boolean operation (incorporating source data, destination data, and bit patterns from halftone registers), the bit shift to be applied to the payload, and geometry information,

3. Start the operation by accessing a special register.

Once started, the blitter operates on the memory bus independent from the CPU. The completion of the operation is signalled by an interrupt and by a bit in a blitter register that can be polled by the CPU.

The 2D hardware-acceleration functions as provided by the blitter were subsequently introduced to other home computers and professional graphics cards.

Silicon Graphics extended the approach of hardware-accelerating graphics to the 3D domain by introducing the Personal Iris workstation in 1988. This workstation featured a raster engine that enabled real-time polygonal 3D graphics. Until the mid-1990's, Silicon Graphics was the innovation leader for hardware-accelerated graphics in the professional workstation domain.

At that time, the gaming industry started to incorporate 3D graphics capabilities into consumer devices. In 1993, Atari introduced the Jaguar game console featuring hardware-accelerated polygonal graphics, an effect processor, and a programmable object processor for 3D calculations. The subsequently released gaming consoles of other vendors featured hardware-accelerated 3D as a mandatory capability for enabling modern games.

In the commodity PC market, 3DFX initiated the rapid evolution of 3D-capable graphics cards by introducing the Voodoo graphics chip in 1996. The Voodoo PCI card was an add-on device that complemented existing VGA display adaptors and was the first low-cost PC hardware that provided advanced graphics capabilities such as textured polygonal graphics. Both the 2D blitting unit of the graphics card and the 3D unit of the Voodoo card had not been used in combination but for entirely different workloads. Windowed GUIs made the benefits of the 2D unit available to their applications via the Graphics Device Interface (GDI) on Windows and respectively the X Acceleration Architecture (XAA) on the X window system. In this architecture, the GUI server accesses the 2D unit exclusively, uses hardware functions to speed up its internal graphics functions, and lets its GUI clients benefit from the improved performance transparently through the client API. In contrast to the 2D unit that was indirectly used by multiple concurrently executed GUI clients, the 3D unit was used by only one application at a time in a low-resolution full-screen mode. Games as the most prominent workload for the 3D unit facilitated direct access to the device to achieve maximum throughput. APIs such as Glide and OpenGL had been used to ease software development and to facilitate code re-usability but through these libraries, the 3D application directly accessed the device.

The 2D unit and 3D unit had two characteristic differences, the sustainability of their hardware interfaces and their programming model. The feature set of the 2D unit remained static over years but 3D units were expanded by new functionality with each new device generation driven by the graphical advances of the games. As the 2D unit was an evolutionary descendant of the blitter, its programming model retained the synchronous mode of operation that required the CPU to program device registers for each single operation. In contrast, the programming model of the 3D unit was laid out asynchronously to enable the execution of huge bulks of small graphical primitives without CPU intervention. The 3D unit uses direct memory access (DMA) to fetch batches of commands from CPU's memory, executes the whole batch of commands, and triggers an interrupt after completing the command execution. To let the hardware operate in such a decoupled way from the CPU, the 3D unit has to hold complex state information in hardware. Compared to relatively simple 2D operations, 3D operations such as polygon rendering are subject to a large number of attributes such as the pixel-sampling method, the texture layout, the alpha values, the colors, and the scan-line-interpolation properties. The set of attributes affecting graphics operations is called *rendering context*.

Even though the 3D unit offered the functionality needed to implement a windowed GUI, commodity GUIs at that time kept relying on the 2D unit. Compared to 2D operations, the batched processing of 3D operations would have introduced higher latencies and the 3D output resolution was not considered to be sufficient for windowed GUIs.

In 1998, NVIDIA introduced Riva TNT as the first graphics card with an integrated display adapter, a 2D unit, and a 3D unit. Other vendors followed and extended the functionality by introducing advanced filtering methods, bump mapping, MIP mapping, transform-and-lighting, vertex shaders, and pixel shaders. The rapid development of the graphics-device market led to a high functional diversity among different graphics-card devices and vendors. With the huge increase of the power of graphics devices, 3D applications other than games became available and facilitated the need to combine 3D acceleration with a windowed GUI. DirectX for Windows OS and DRI for the X window system introduced time sharing of the 3D

unit between multiple GUI clients to make 3D acceleration exploitable not only by full-screen games but also by GUI clients. To provide the highest performance to the newest generation of games, the driver-software stack had to provide access to the latest device features and had to maximize the graphics throughput. Under these circumstances, subordinating security to performance in the system design can be justified because, in contrast to 3D performance, system security was no selling point.

Deducing from the observations that the output latencies of 3D units sufficed for the interactivity of games and that the resolution of 3D games was no more lower than typical GUI resolutions, Apple Computer introduced Quartz Extreme in 2002 as the first windowed GUI server of a commodity OS utilizing the 3D unit. The vendors of the other major commodity GUI servers followed this path of utilizing the 3D unit for 2D window composition. Both the Desktop Window Manager of Windows Vista and the AIGLX/Compiz extension of the X.org X server solely rely on the 3D unit and thereby degrade the 2D unit of PC graphics cards to a legacy. The remaining part of the document does not address the 2D unit but only the 3D unit being called *graphics processing unit* (GPU).

5.2 Device overview

In a windowed GUI with multiple GUI clients and the GUI server being active at the same time, making efficient use of the GPU for accelerating graphics corresponds to the problem of how to efficiently share the physical device. The display architectures of today's commodity GUIs provide solutions for this problem. But by taking security as a premise as stated in the introduction of this chapter, this problem becomes the challenge of how to share the physical graphics device in a secure and safe manner. Understanding this challenge requires the following basic technical knowledge about the characteristics of graphics devices.

Today's graphics devices provide three different resources to be multiplexed among multiple GUI clients:

Local graphics memory

Graphics devices that are not integrated into the chip set of the mother board feature local memory. Integrating memory local to the GPU maximizes the bandwidth between the GPU and the graphics memory. Because this bandwidth is a determining factor for the rendering throughput, the bus between GPU and local graphics memory is optimized for GPU access patterns.

On typical graphics devices, the operands such as source textures and the results of GPU operations are restricted to be located in local graphics memory. In contrast to the bandwidth between GPU and local graphics memory, the bandwidth between the CPU and the local graphics memory is orders of magnitude lower because such data transfers rely on the peripheral bus of the CPU. To efficiently use the GPU, accesses by the CPU to the local graphics memory must be avoided.

Graphics devices that are integrated into the chip set of the motherboard utilize host memory as local graphics memory. Although this architecture avoids the use of a slow peripheral bus between GPU, CPU, and graphics memory, the achievable bandwidth between GPU and the graphics memory is limited by the capacities of the host-memory controller.

Frame buffer

The frame buffer is the portion of the graphics memory that corresponds to the image displayed on screen.

GPU

The GPU is the graphics processor that executes graphical primitives. It operates on the graphics memory but also supports the fetching of data via DMA from host memory, for example for loading a source texture into the local graphics memory prior to using the texture for a graphics operation.

In the following, programs that utilize these resources are called GPU clients. With this notion, both the GUI server that uses the GPU for window redrawing and GUI clients that use the GPU for rendering operations are GPU clients.

Sharing the physical resources of the graphics device involves the following security risks. If one GPU client is able to observe state of another GPU client without authorization, for example by reading from arbitrary graphics-memory locations, the confidentiality of processed information cannot be assured, thereby breaking isolation between GPU clients and enabling the operation of spyware. Analogously, a GPU client that can arbitrarily modify the state of other GPU clients may violate the integrity of the processed information and thereby introduce attack vectors for Trojan Horses. If the time sharing of the GPU and the allocation of graphics memory is not subjected to policy such as quotas, one GPU client may exhaust the available resources and put the availability of the graphics subsystem at risk. As observed by my personal experience and supported by [25], commodity graphics devices are prone to robustness problems when used in a wrong way. Wrong usage may result in an inconsistent state or a lock-up of the device making the GUI unavailable.

The majority of commodity graphics devices do not provide hardware support to avoid such problems. In contrast to the CPU's memory that can be safely used by multiple untrusted programs via the memory-management unit (MMU), there is no MMU for graphics memory. Each program with direct access to the GPU is able to issue GPU operations that operate on arbitrary graphics-memory locations and thereby potentially violate the integrity, confidentiality, and availability of the entire GUI.

The constraints of graphics devices on the one hand and the security requirements on the other hand implicate the fundamental technical challenges of how to partition graphics memory in such a way that isolation between GPU clients is maintained and how to temporally multiplex the GPU between GPU clients such that liveliness of each GPU client is guaranteed. By introducing the endeavour of minimizing the complexity of application-specific TCB in Section 4.1.2 as primary non-functional design guideline, the code complexity on account of the performance optimization through GPU usage has to stay in relation to the purely functional parts of the GUI-related TCB.

5.3 Design space for multiplexing graphics hardware

This section discusses two approaches that span the design space for multiplexing a graphics device, API-level multiplexing and device-level multiplexing. By revisiting existing implementations, the security-related limitations of both approaches become apparent.

5.3.1 API-level resource multiplexing

As mentioned in Section 5.1, when 2D graphics acceleration was introduced, system software made the performance benefit available to applications in a completely transparent way via a system API. The design of the 2D-hardware features was driven *by* existing APIs such as Windows' GDI and the X protocol. From the opposite perspective, an API provides an abstract and convenient interface to the hardware facilities and hides device diversity. An API with

support for client contexts (e. g., a device context for GDI, or a virtual workstation for GEM) implicitly facilitates the use of hardware acceleration by multiple clients through the indirection of the software stack implementing the API. API-based multiplexing worked well for 2D acceleration because the hardware facilities had been simple and only moderately diverse among different devices and vendors.

In contrast, 3D hardware is highly diverse with regard to implemented features and initial 3D APIs such as Glide lacked support for client contexts. Therefore, these APIs did not facilitate device multiplexing at the API level. Approaching API-level multiplexing of the 3D hardware raises the question of which API to choose as a basis for the GUI and all graphical applications. As APIs try to provide an abstraction from the hardware, not all hardware features may be sufficiently exposed to the programmer. For example, NVIDIA introduced pixel shaders into the GeForce 3 graphics devices in the year 2001 but the OpenGL API provided full support for this feature not before 2006 when revision 2.1 was released. The other way, if an API provides a superset of the functionality supported by the device, software fallbacks are required. On account of the high diversity of devices, a perfect correspondence between a generic high-level API and a specific device is unlikely. Despite of that inherent compromise, with Direct3D and OpenGL, there exist established APIs that are regarded as both universally usable and well-supported by graphics hardware.

The programming model of OpenGL as the de-facto industry standard is a state machine that holds a set of attributes forming the *rendering context*. OpenGL commands as issued by an application programmer fall into one of three categories:

- Configuration of the rendering context by assigning rendering attributes such as the color and lighting,
- Execution of a graphical primitive according by the current state of the rendering context, or
- Special control functions such as `SwapBuffers` that are independent of the rendering context.

For the X window system, there exists GLX as an extension of the X protocol. GLX is a simple wire protocol of API-level OpenGL commands. The X server holds the rendering context for each GLX client and exposes the complete OpenGL API as client interface. Internally, the X server serializes the GLX streams received from its clients and transforms the resulting sequence of OpenGL-commands to the device level. Because the X server has a global view about the state of all GLX clients, it is able to implement the switching of rendering contexts and thereby time-multiplexes the graphics device. By using a stream representation of OpenGL commands, GLX preserves the network transparency of the X window system.

The usage of streamed OpenGL commands was taken further by the WireGL project [17]. WireGL optimized the stream protocol for minimizing network bandwidth. Instead of holding the rendering contexts only in the server, WireGL lets each client track state changes of its rendering context and lazily send updates to the server only when needed. Furthermore, WireGL handles the rendering context not just as a flat set of attributes but it represents all attributes in a hierarchy. The hierarchy alleviates the costs of calculating the difference between rendering contexts and thereby enables efficient lazy updating of the rendering context at the server. Furthermore, this technique facilitates the use of soft context switching inside the server. Instead of saving and restoring the rendering contexts at device level, the server can efficiently apply only the differences between the old and new context at the API level. As presented in [17], soft context switching is able to improve context switching times by orders of magnitude and it does not require any hardware support.

The Chromium project [44] conducted the multiplexing of rendering hardware as one aspect of distributing OpenGL workload across networked cluster nodes. Based on the experiences made with WireGL, this work further generalized the use of streamed sequences of OpenGL commands. Each node in the cluster receives one or more incoming OpenGL command streams, applies stream transformations, and emits outgoing streams. By using this terminology, the execution of rendering commands is regarded as a stream transformation, for example a node may receive OpenGL geometry commands (e. g., `glVertex3d`) as input, execute rendering commands, and produce OpenGL raster commands (e. g., `glDrawPixels`) as output of the transformation. A rendering node that receives multiple OpenGL streams must apply a *serializing* transformation to the incoming streams to produce one stream of commands to be passed to the rendering hardware. Consequently, such a serializing portion of a rendering node multiplexes the device at the API-level.

VMGL and the Blink project [36] applied the streaming of API-level commands to the domain of virtual machines and addressed the problem of utilizing graphics hardware by multiple untrusted guest OSes hosted on the same machine. Blink relies on one complete OpenGL protocol stack contained in Xen's [15] trusted domain called Dom0. On top of the OpenGL stack, a display manager provides a custom designed command streaming interface to be accessed by applications running within different unprivileged virtual machines. A subset of Blink's command streaming interface resembles the OpenGL API, which is made available to client applications via a custom `libGL` shared library installed in the guest OSes. This way, Blink facilitates backward-compatibility to OpenGL. With a desktop workload on Xen, the primary constraint is not the bandwidth of the virtual network between virtual machines but the context-switching costs between virtual machines and the response latency with respect to the user activity. Blink facilitates the reduction of the number of context switches between virtual machines by enabling clients to offload parts of their program logic (e. g., mouse-focus highlighting) into server-side stored procedures that are executed locally in the server without involving the client. Of the mentioned projects, Blink is the solution that is closest to the workload I am addressing.

From a practical point of view, API-level multiplexing as implemented by Blink is economic with regard to engineering costs because it reuses existing 3D infrastructure such as device drivers for the GPU, the OpenGL protocol stack, and the display capabilities of the host OS, for example the X window system or the Linux frame buffer. Through API-level multiplexing, Blink is able to provide isolation and safety on a conceptual level by relying on the premise that the TCB complexity of the 3D protocol stack is not a problem. As I discussed in Section 4.1.2, however, I object that this premise is valid.

5.3.2 Device-level resource multiplexing

When performing API-level resource multiplexing, the API protocol stack must be trusted. It is complex but at the same time it is shared by all GPU clients. The preservation of security depends on how strongly the protocol stack enforces the isolation of its clients and its robustness against resource exhaustion. Because an API such as OpenGL is a generalized abstraction from hardware, it imposes constraints to the exploitation of special device features. Furthermore, a resource-multiplexing protocol stack is an indirection between the device and a GPU client and, therefore, it implicates overhead, for example caused by the context switches between the GPU server and a GPU client when processing consecutive batches of commands. In contrast, a program with direct access to the device is not subjected to the constraints of an API with regard to the use of hardware features.

Device-level multiplexing aims at preserving the richness of devices for multiple GPU clients and, at the same time, maximizing the performance by avoiding the indirection be-

tween the GPU client and the hardware. The Direct-Rendering Infrastructure (DRI) project [55] is an implementation of device-level multiplexing on UNIX. In this architecture, the OS kernel time shares the access to the physical graphics device between all GPU clients in a round-robin fashion. Furthermore, the OS kernel manages the allocation of graphics memory and arbitrates the supply of commands to the GPU. Each GPU client brings along driver code and an API protocol stack of its choice, for example the Mesa library stack including a back-end driver for the installed graphics card. DRI assumes that applications behave cooperatively and use the device in a compliant way. This assumption is reflected by the design decision of making the complete device including the entire graphics memory, frame buffer, and all control registers accessible to each GPU client. As described in [26], the asynchronous execution of GPU commands by the graphics card makes explicit synchronization of software rendering by GPU clients inevitable. For this synchronization, DRI relies on a global lock that is managed cooperatively by all GPU clients. DRI provides special measures against deadlocks caused by a crashing application that is the current lock holder but there are no measures against applications that misbehave maliciously. Combined, these design properties subvert the security goals expressed in Chapter 4. Because spatial partitioning of graphics memory and the frame-buffer in particular is not enforced, the integrity and confidentiality of displayed information cannot be guaranteed. Because each GPU client is able to monopolize access to the graphics device by grabbing the global lock or by exhausting graphics memory by unrestricted allocations, the liveliness of GPU clients is put at risk. DRI puts the availability of the host GUI at the mercy of all GPU clients because the X window system itself relies on DRI as one member among all GPU clients.

DRI addresses security merely by the means of access control. The access right to DRI as a whole via the `/dev/dri` device can be granted or denied per process. Once a process is permitted to access `/dev/dri`, there is no restriction of how the device can be used [25]. The majority of commodity Linux distributions facilitate the assignment of the access right to `/dev/dri` on a per-user basis by introducing a UNIX group for DRI and only assigning selected users to the DRI group. Because most users desire the benefit of hardware-accelerated 3D graphics, this access right is typically granted to all user applications. Therefore, the GUI-related TCB includes all applications executed by the end user.

In [53], Frank Mehnert secured network-device and IDE-device programming by untrusted driver code by faithful device virtualization. The work was based on the observation that only a small portion of the device interface, namely the DMA engine, is security critical whereas the access to the major parts of the device interface by untrusted clients imposes no risk. A small trusted mediator was introduced as an indirection between the raw device and the untrusted driver code. To its untrusted client, the mediator provides a virtual device interface with a register layout that corresponds exactly to the physical device interface. The mediator installs itself as the page-fault handler for the security-critical portion of the virtual MMIO registers. For uncritical virtual MMIO registers, it creates a direct mapping of physical to virtual MMIO registers. This way, the mediator intercepts all critical hardware accesses, validates DMA requests, and performs the actual programing of the physical DMA registers. This concept was applied for each of the Digital DS2114x Tulip fast-ethernet device, the Intel PRO/1000 gigabit ethernet device, and an IDE controller. For those devices, the required code complexity of the mediator turned out to be lower than 1,000 SLOC, which is factor 5-10 smaller than the reused untrusted driver code taken from the Linux kernel version 2.2.26.

In contrast to the investigated network and IDE devices, commodity graphics devices such as ATI's Radeon device family usually have security-critical registers and uncritical registers co-located at the same physical page preventing a selected interception of only critical device accesses. Consequently, a large subset of the uncritical registers must be intercepted to

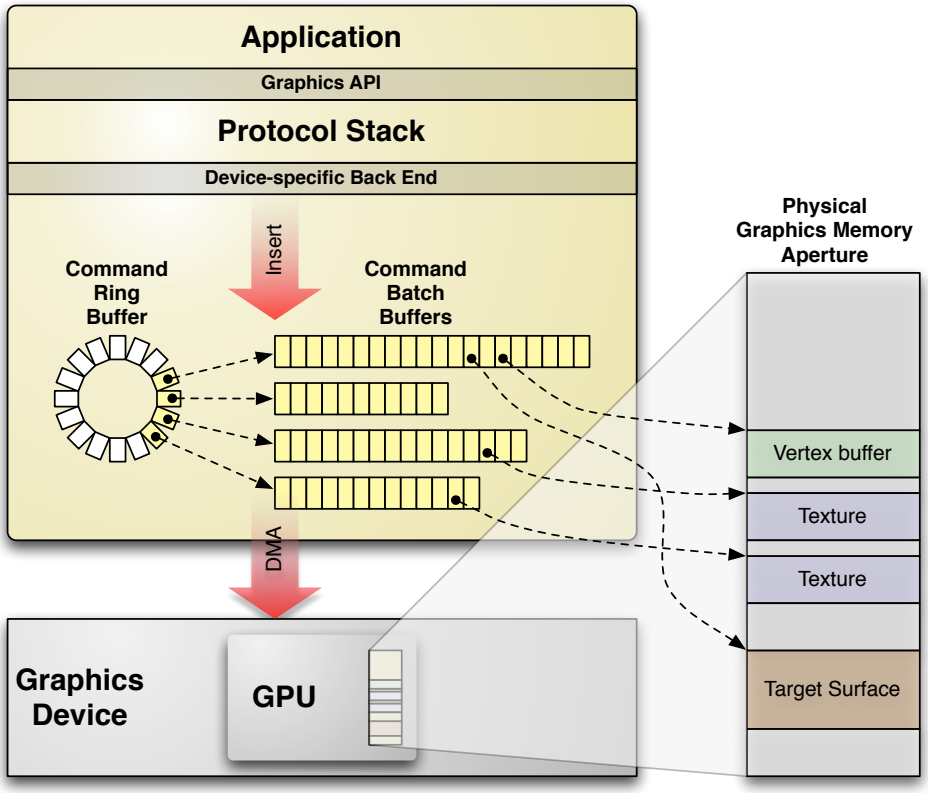


Figure 5.1: A single application directly operates on the graphics device by using the GPU's command-stream interface.

capture the critical accesses. Trapping uncritical registers such as drawing-attribute registers does not contribute to security but spoils the performance.

On the one hand, faithful graphics-device virtualization at register level seems difficult and inefficient. On the other hand, the device-register interface is not used for modern 3D workloads. Instead, GPUs are programmed via batches of higher-level commands. The Windows Device Driver Model introduced with Windows Vista exercises GPU multiplexing based on the abstraction level of GPU commands streams. The following section explains its principle design, which ultimately motivated recent improvements of graphics devices outlined in the subsequent Section 5.5.

5.4 GPU command-stream multiplexing

Figure 5.1 illustrates the interaction between software and hardware for an application that uses the graphics device directly. The application uses the programming model of an abstract graphics API such as Direct3D or OpenGL. The protocol stack translates the graphics API calls to device-specific commands. To perform this translation, the protocol stack has to have the knowledge about both the semantics of the graphics API and the capabilities of the graphics device. If a particular API feature is not supported by the device, the protocol stack has to provide software-based fall-back functions. Hence, graphics protocol stacks are complex. The back end of the protocol stack emits batches of device-specific GPU commands, which are made available to the GPU via DMA. Figure 5.1 presents the mechanism as provided by Intel's Graphics Media Accelerator (GMA) chip sets. The driver software sets up a com-

mand ring buffer in DMA-capable memory and signals new GPU commands by updating the ring buffer's tail-pointer register. During command execution, the GPU maintains the ring buffer's head pointer accordingly such that both software and hardware can synchronize. In a typical mode of operation, subsequent graphics commands are not immediately enqueued into the ring buffer but stored in batch buffers. If a batch of commands is completely assembled by the protocol stack, the driver submits the whole batch by inserting a special command referencing the batch buffer into the command ring buffer. GPU commands as contained in batch buffers fall into the following categories:

- Graphics primitives
- Setup of rendering attributes such as pixel formats, vertex formats, filtering methods, colors, and shader programs
- Setup functions for source textures, vertex buffers, and the current target surface including the definition of the clipping area
- Synchronization and control-flow commands

Commands of the latter two categories contain references to the physical graphics memory. If sharing the device between multiple applications in a DRI fashion, the physical graphics memory is shared by these applications. Therefore, GPU commands referencing physical graphics memory are critical for maintaining the isolation between applications. In contrast, GPU commands of the first two categories are uncritical because they contain no direct references to physical memory locations.

The majority of commands produced by typical workloads are graphics primitives and attribute definitions. Compared to the occurrence of these uncritical operations, critical commands are sparse. The Windows Device Driver Model described in the following section exploits these characteristics for virtualizing graphics memory in software.

5.4.1 Windows Device Driver Model

The Windows Device Driver Model (WDDM) [11, 10, 59] removes the high-complexity graphics protocol stack from the TCB and maintains the isolation of GPU clients by virtualizing the graphics memory at GPU command level. As shown in Figure 5.2, the graphics protocol stack is split into a trusted kernel-mode driver[1] shared by all GPU clients and an untrusted user-mode driver linked to each GPU client. Each UMD produces GPU commands as if programming the device directly except for commands that refer to graphics memory locations. Instead of using physical addresses for these commands, the UMD inserts virtual addresses for memory objects as obtained from the KMD's memory manager. Each time when a GPU client submits a batch of GPU commands to the KMD, the batch has to be filtered for potentially critical commands. An example for such a critical command is `STORE_DWORD_IMM` as provided by the Intel GMA chip sets. This command issues a DMA write operation to an arbitrary physical address on systems without IOMMU support. To maintain security and robustness, such commands must be detected. After having successfully validated each command, the KMD schedules the batch by inserting its reference into the command ring buffer and replacing all embedded virtual graphics memory locations by the corresponding physical addresses. The mapping between virtual and physical addresses is maintained by the KMD in software.

[1] Despite of its name, the kernel-mode driver does not need to be part of the OS kernel. On a microkernel-based OS, the KMD would be realized as user-level component with no enhanced privileges other than direct access to the GPU.

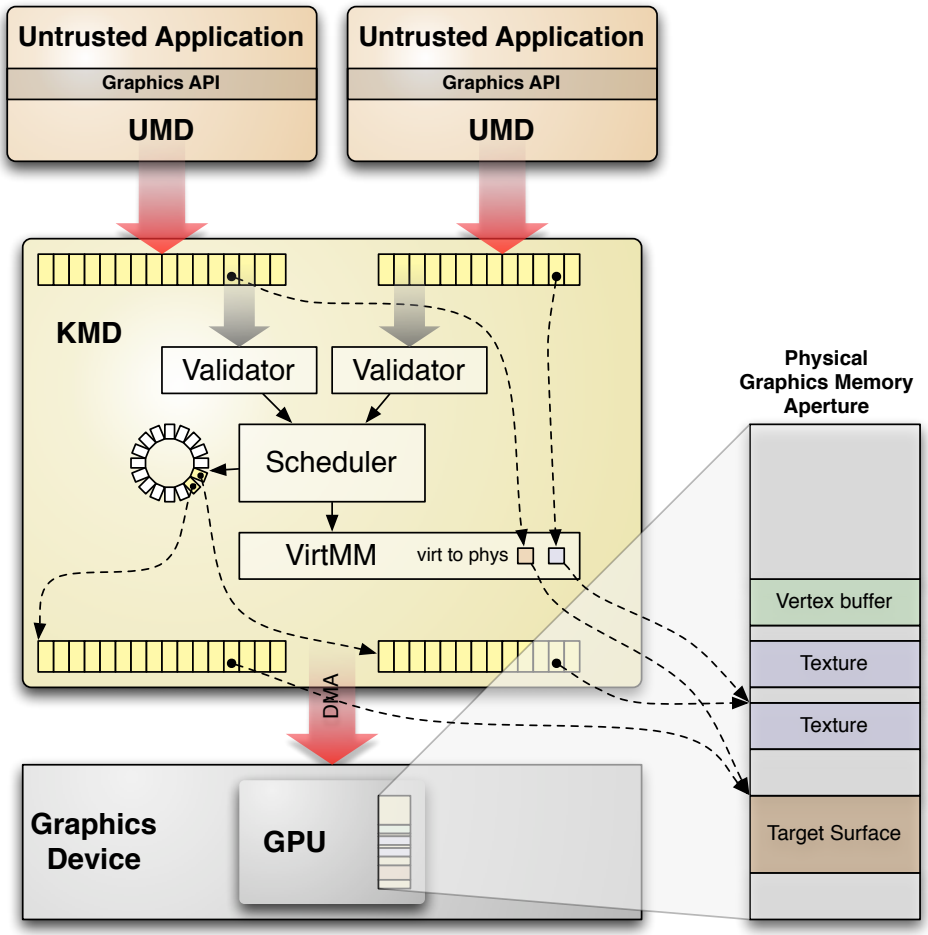


Figure 5.2: Windows Device Driver Model

With Windows Vista's Desktop Window Manager (DWM) being a Direct3D application, the problem of securely multiplexing the GPU among multiple GPU clients becomes inherent. The original WDDM architecture resolves this problem on the GPU command stream level by virtualizing graphics memory and performing GPU scheduling in software. Thereby, WDDM reduces the TCB for maintaining GPU client isolation to the KMD. The UMD as the major contributor to the complexity of the graphics protocol stack can be regarded as untrusted. In contrast to the UMD, which may be frequently enhanced by new graphics features, the KMD's functionality remains rather fixed. By decoupling UMD and KMD, the UMD can be updated at runtime similar to other user-level libraries. Furthermore, different UMDs can be concurrently active for different applications.

The drawbacks of multiplexing GPU command streams solely by software are the costs for scanning GPU command streams submitted by the untrusted GPU clients and the manual address translation. For example, GPU commands as present in Intel's GMA chip sets have variable lengths and multiple opcode fields. For parsing streams of such commands and to protect the system from critical operations, the KMD requires intimate knowledge about the GPU commands potentially issued by UMDs. To overcome these drawbacks, Microsoft facilitated hardware improvements for graphics devices by introducing virtual graphics memory and hardware contexts as mandatory features for devices compliant to DirectX version 10.

5.5 Hardware-supported GPU-context management

With the GMA X3000 chip set released in 2006, Intel introduced the security-related hardware improvements demanded by DirectX version 10. This chip facilitates the management of multiple rendering contexts in a similar way to how Intel's Virtualization Technology (TV) [9] CPU extension supports the management of virtual-machine state. Analogously to how VT maintains CPU state in an opaque memory area called virtual-machine control structure (VMCS), the GPU is able to save the current rendering context to an opaque memory area and restore a new context from memory respectively. On the GMA X3000 chip, the graphics driver steers the switching of rendering contexts by inserting `MI_SET_CONTEXT` instructions into the GPU command stream to select the current rendering context. Only one context can be active at a time. In addition to the support of temporal isolation between GPU clients via hardware-based context switching, the chip set supports spatial isolation via two levels of memory-address translation. Inherited from the previous chip-set generation, a global graphics-translation table (GTT) describes the composition of the graphics-memory aperture from physical memory pages. The GTT is a flat table of GTT entries. Each entry describes the 4KB graphics-memory aperture page corresponding to its table index and it holds a reference to the assigned physical memory page. With the GMA X3000, Intel introduced an additional address-mapping via the per-process GTT (PPGTT). If enabled, the PPGTT describes the virtual graphics memory as visible by the current context performing GPU operations. Each PPGTT entry refers to a page in the GTT address space. In contrast to the GTT, which includes the globally needed mapping of the physical frame buffer and the DMA buffers for GPU commands, the PPGTT can be tailored to be more restrictive, including only the memory regions used by the rendering operations of the current context. If the GPU accesses an invalid memory address, the device signals the access fault via an interrupt at the CPU and halts the GPU. Driver software can then resolve the fault by modifying the GTT and PPGTT accordingly or by taking a scheduling decision. With the support of rendering-context management and per-process virtual graphics memory in hardware, the GMA X3000 chip set enables the execution of untrusted user-mode device drivers without the need for parsing GPU commands by the KMD software. This hardware improvement maintains native ren-

dering performance and frees the KMD from knowledge about complex rendering operations offered by the GPU.

The drawback of Intel's GMA X3000 solution is bad scalability with regard to the number of rendering contexts. Because graphics page-fault resolution is done in a synchronous way, the GPU is halted until the KMD has resolved the fault. If the fault resolution involves long-taking operations, the GPU utilization drops. Analogously to VT's `vmsave` and `vmrestore` operations, the mechanism provided by the `MI_SET_CONTEXT` GPU command copies large state and therefore, it becomes costly when used for fine-grained scheduling. Furthermore, the programming model requires the driver to explicitly insert preemption points into the command stream or to apply scheduling decisions at the granularity of complete batch buffers, which implies potentially high scheduling latencies.

WDDM version 2.1 addresses these drawbacks by facilitating further hardware improvements. In November 2007, AMD introduced its first graphics devices based on the RV670 chip compliant to WDDM 2.1. The major improvement of this device generation is the execution of muliple GPU contexts in parallel. If one GPU context faults, the other contexts remain active and can proceed with rendering operations. Managing multiple contexts locally at the graphics chip effectively eliminates the costs for context-switching. Furthermore, the latency for switching contexts is minimized by making the GPU processing interruptible at the granularity of a single pixel. This way, long-taking fragment shaders as employed by increasingly popular non-graphical GPU clients [2] do not impose unbounded delays on concurrently active graphical GPU clients.

5.6 TCB complexity on account of hardware-accelerated graphics

The recent introduction of MMU mechanisms into the current generation of graphics devices apparently resolves the GPU multiplexing problem. Thanks to these hardware improvements, enabling hardware-accelerated graphics does not inherently imply that highly complex driver code is contained in the TCB. By utilizing hardware-supported virtual graphics memory and context management, the trusted part of the graphics protocol stack can be reduced to the GPU scheduler, the GPU page-fault handler, and the memory manager. It does not need to parse GPU commands because GPU command streams as produced by untrusted user-mode drivers contain only virtual addresses. For ensuring the isolation between GPU clients, there is no knowledge about the highly complex graphical primitives needed and thanks to the fine-grained interruptibility of the GPU, the KMD is not required to validate fragment-shader programs to maintain bounded scheduling latencies.

Now that documentation for current-generation GPUs (AMD's R6xx series and Intel's GMA X3000) has become publicly available, a practical evaluation of the achievable TCB minimalism becomes feasible and poses an interesting challenge for future work.

My practical experiences with programming various graphics devices (Matrox, ATI Rage, ATI Radeon, Intel GMA) and existing source code of open-source drivers already allow for the estimation that an achievable low complexity of the trusted driver portion lies in the order of 5,000 to 10,000 SLOC.

Chapter 6

Conclusion

When I started my work on GUIs, I identified the four major design challenges quality of service, compatibility, security, and high performance, which seemed to contradict with each other. With my work, I strived for solving this conflict using a combination of existing and novel techniques.

Moving the client representation to the GUI server makes the costs for redrawing jobs predictable and enables the application of real-time scheduling. By exploiting the special characteristics of redraw jobs, optimization techniques such as lazy updating the client state on screen and redraw dropping become feasible. These optimizations ultimately lead to a design that considers the GUI quality of service as a global system parameter and eliminates the need for temporal client models. The spatial partitioning of the frame buffer as performed by my proposed design maintains full isolation of GUI clients from each other and thereby protects the confidentiality of the presented information. My solution for the unforgeable watermarking of screen regions enables the user to counter attacks by Trojan Horses. Combined with secure routing of user input, the GUI server ensures a bidirectional trusted path between each single application and the user. To prevent resource-exhaustion-based denial-of-service attacks, I combined resource donation with a novel heap-partitioning technique. With the help of recent hardware improvements, my design becomes able to benefit from hardware-accelerated graphics without compromising security.

I validated my design propositions with extensive experiments. My DOpE GUI server is the first GUI server modelled as a periodic real-time process. Thereby, it is able to provide quality of service guarantees and it prevents overload situations by design. My most remarkable implementation work is Nitpicker, which is a GUI server that accommodates complete existing windowing systems and low-complexity security-sensitive applications side by side.

The most distinctive property of Nitpicker is its source-code complexity of less than 1,500 lines of code, which is only a fraction of previously existing GUI servers.

As a composition, the techniques presented in this document overcome the apparent conflict between the different goals posed on GUIs. Supporting quality of service, accommodating the existing wealth of GUI-based applications, and achieving high performance while maintaining security on the GUI level are not contradicting each other.

Outlook

This thesis does not mark the end of my work in the field of secure GUIs. Instead, I regard it as the foundation of my upcoming activities [3] of pairing the presented techniques with my OS architectural work [31]. By combining both, I strive to prove that a TCB complexity for graphical applications in an OS environment with support for general-purpose desktop workloads can be in the order of tens of thousands rather than millions of lines of code, which is the state of the art.

Bibliography

[1] Apple Mac OS X website. Apple Inc., URL: http://www.apple.com/macosx/. 47

[2] General-Purpose computation on GPUs website. URL: http://www.gpgpu.org. 71

[3] Genode Labs website. URL: http://www.genode-labs.com. 74

[4] Hatari website. URL: http://hatari.sourceforge.net. 31

[5] SLOCCount website. URL: http://www.dwheeler.com/sloccount/. 21

[6] Tcl/Tk website. URL: http://www.tcl.tk. 21

[7] VMware website. URL: http://www.vmware.com. 26, 35

[8] Wine website. URL: http://www.winehq.org. 26

[9] Intel Vanderpool Technology for IA32 Processors (VT-x). Intel Corporation Order Number C97063-001, January 2005. 40, 70

[10] ATI Catalyst Graphics Drivers and WDDM whitepaper. ATI Technologies Inc., URL: http://ati.amd.com/products/wp/atiwddmwhitepaperfinalv38.pdf, 2006. 68

[11] MSDN Library: Windows Vista Display Driver Model introduction. Microsoft Corporation, URL: http://msdn2.microsoft.com/en-us/library/aa480220.aspx, July 2006. 4, 68

[12] L. Abeni and G. Buttazzo. Integrating Multimedia Applications in hard Real-time Systems, December 1998. 23

[13] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In Proceedings of the 1997 IEEE Symposium on Security and Privacy, pages 65–71., May 1997. 41

[14] Paul Barham. Devices in a Multi-Service Operating System. July 1996. 23

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*, pages 164–177. Bolton Landing, NY, October 2003. 26, 65

[16] Paul T. Barham, Mark Hayter, Derek McAuley, and I. Pratt. Devices on the Desk Area Network. *IEEE Journal of Selected Areas in Communications*, 13(4):722–732, 1995. 23

[17] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking Graphics State for Networked Rendering. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 87–96, August 2000. 64

[18] Alan Coopersmith. Open Source Multi-level Security in X. Slides from talk at the Desktop Developers' Conference, URL: `http://people.freedesktop.org/ alanc/ddc-2006.pdf`, 2006. 38, 54

[19] Tresys Technology David Caplan. SELinux Policy Analysis - Concepts and Techniques. SELinux Symposium, 2005. 55

[20] Jeremy Epstein. A prototype for Trusted X labeling policies. In *Proceedings of the Sixth Annual Computer Security Applications Conference*. Tucson, AZ, USA, December 1990. A discussion of visible labeling issues, not specific to X, but applicable to any windowing environment. 46, 55

[21] Jeremy Epstein. A High-Performance Hardware-Based High Assurance Trusted Windowing System. In *Proceedings of the 19th National Information Systems Security Conference*, October 1996. 22, 34

[22] Jeremy Epstein, John M c Hugh, Rita Pascale, Hilarie Orman, Glenn Benson, Charles Martin, Ann Marmor-Squires, Bonnie Danner, and Martha Branstad. A Prototype B3 Trusted X Window System. In *Proceedings of the 7th Annual Computer Security Applications Conference (ACSAC)*, December 1991. 34

[23] Jeremy Epstein, John McHugh, Hilarie Orman, Rita Pascale, Ann Marmor-Squires, and Bonnie Danner et al. A high assurance window system prototype. 46, 54

[24] Jeremy Epstein and Marvin Shugerman. A Trusted X Window System server for Trusted mach. In *Proceedings of the USENIX Mach Conference, Burlington, VT, USA*, October 1990. 34

[25] Rickard E. Faith and Kevin E. Martin. A Security Analysis of the Direct Rendering Infrastructure. Cedar Park, Texas: Precision Insight Inc., 1999. 63, 66

[26] Rickard E. Faith, Jens Owe, and Kevin E. Martin. Hardware Locking for the Direct Rendering Infrastructure. Cedar Park, Texas: Precision Insight Inc., 1999. 66

[27] Norman Feske. A Case Study on the Cost and Benefit of Dynamic RPC Marshalling for Low-level System Components. SIGOPS OSR Special Issue on Secure Small-Kernel Systems, July 2006. 52

[28] 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. 14

[29] 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. 14

[30] Norman Feske and Christian Helmuth. Overlay window management: User interaction with multiple security domains. Technical Report TUD-FI04-02-März-2004, TU Dresden, 2004. 34

[31] Norman Feske and Christian Helmuth. Design of the Bastei OS architecture, subsequently called Genode OS Framework. Technical Report TUD-FI06-07-Dezember-2006, TU Dresden, 2006. 6, 74

[32] Decklin Foster. Aewm website. URL: `http://www.red-bean.com/~decklin/aewm/`. 30

[33] X.Org Foundation. X.org website. URL: `http://www.x.org`. 9, 26, 29

[34] Thomas Friebel. Portierung der libSDL auf DROPS / DOpE. Großer Beleg, TU Dresden, URL: `http://os.inf.tu-dresden.de/papers_ps/friebel-beleg.pdf`, 2005. 26, 31

[35] Scott Garriss, Ramón Cáceres, Stefan Berger, Reiner Sailer, Leendert van Doorn, and Xiaolan Zhang. Towards Trustworthy Kiosk Computing. In *Proceedings of the 8th IEEE Workshop on Mobile Computing Systems and Applications (HOTMOBILE)*, pages 41–45. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-3001-X. 56

[36] Jacob Gorm Hansen. Blink: Advanced Display Multiplexing for Virtualized Applications. In *Proceedings of the 17th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), Urbana, Illinois, pages 15-20*, June 2007. 65

[37] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*. Sintra, Portugal, September 1998. 14, 26, 29

[38] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77. Saint-Malo, France, October 1997. 26

[39] Hermann Härtig. Security Architectures Revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*. Saint-Emilion, France, September 2002. 40

[40] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART)*. Adelaide, Australia, September 1998. 29

[41] Christian Helmuth. Ein Konsolensystem für DROPS. Großer Beleg, TU Dresden, URL: `http://os.inf.tu-dresden.de/papers_ps/helmuth-beleg.pdf`, 2000. 25

[42] Christian Helmuth, Alexander Warg, and Norman Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.CH Security 2005*. Darmstadt, Germany, March 2005. 40

[43] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002. 29

[44] G. Humphreys, M. Houston, Y. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters. Presented at SIGGRAPH, San Antonio, Texas, 2002. 65

[45] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2005)*. San Jose, California, USA, December 2005. 40

[46] Derek McAuley Ian Leslie, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications, 14(7)*, September 1996. 23, 56

[47] SUN Microsystems Inc. Virtual box website. URL: `http://www.virtualbox.org`. 26

[48] SUN Microsystems Inc. UltraSPARC T1 Hypervisor API Specification, January 2006. 40

[49] 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. 29

[50] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42. USENIX Association, Berkeley, CA, USA, 2001. ISBN 1-880446-10-3. 38

[51] Y. Malaiya and J. Denton. Estimating Defect Density using Test Coverage. Colorado State University Tech. Report CS-98-104., 1998. 39

[52] Nicola Manica, Luca Abeni, and Luigi Palopoli. QoS Support in the X11 Window Systems. In *Proceedings of 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), St. Louis, MO, United States*, April 2008. 23

[53] Frank Mehnert. *Kapselung von Standard-Betriebssytemen*. PhD thesis, TU Dresden, July 2005. 66

[54] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005. 51

[55] Brian Paul. Introduction to the Direct Rendering Infrastructure. Linux World 2000, San Jose, 2000. 66

[56] Rob Pike. The blit: A Multiplexed Graphics Terminal. *Bell Labs Tech. J.*, 63(8, part 2): 1607–1631, 1984. 1

[57] Martin Pohlack, Björn Döbel, and Adam Lackorzynski. Towards Runtime Monitoring in Real-Time Systems. In *Proceedings of the Eigth Real-Time Linux Workshop*. Lanzhou, China, 2006. URL `http://os.inf.tu-dresden.de/papers_ps/pohlack06runtime_monitoring.pdf`. 19, 20

[58] I. Pratt. User-Safe Devices for True End-to-End QoS. Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), 1997. 23

[59] Steve Pronovost, Henry Moreton, and Tim Kelley. Windows Display Driver Model (WDDM) v2 and beyond. WinHEC, 2006. 68

[60] Torvald Riegel. A Generalized Approach to Runtime Monitoring for Real-Time Systems. Diploma thesis, TU Dresden, URL: `http://os.inf.tu-dresden.de/papers_ps/riegel-diplom.pdf`, June 2005. 21

[61] Carsten Rietzschel. VERNER - ein Video EnkodeR uNd playER für DROPS. Diploma thesis, TU Dresden, URL: `http://os.inf.tu-dresden.de/papers_ps/rietzschel-diplom.pdf`, October 2003. 20

[62] James A. Rome. Compartmented Mode Workstations. Oak Ridge National Laboratory, presentation at DOE Computer Security Meeting, Seattle, WA, April 1995. 55

[63] John E. Sasinowski and Jay K. Strosnider. ARTIFACT: A Platform for Evaluating Real-Time Window System Designs. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS)*, pages 342–352, 1995. 22

[64] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Trans. Graph.*, 5(2): 79–109, 1986. ISSN 0730-0301. 4

[65] Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt. The Interactive Performance of SLIM: a Stateless, Thin-Client Architecture. In *Proceedings of the 17th ACM symposium on Operating systems principles (SOSP)*, pages 32–47. ACM Press, New York, NY, USA, 1999. ISBN 1-58113-140-2. 26

[66] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast Capability System. In *Proceedings of the 17th ACM symposium on Operating systems principles (SOSP)*, pages 170–185. ACM Press, New York, NY, USA, 1999. ISBN 1-58113-140-2. 38, 55

[67] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the EROS Trusted Window System. In *Proceedings of the 13th USENIX Security Symposium (2004)*, pages 165–178, 2004. 14, 38, 42, 43, 48, 55, 57

[68] Chuck Thacker, Ed McCreight, Butler Lampson, Robert Sproull, and David Boggs. Alto: A Personal Computer. *Computer Structures: Principles and Examples, second edition, ed. Siewiorek, Bell and Newell, McGraw-Hill, 1981*, pages 549–572, 1979. 1

[69] Eamon F. Walsh. X Access Control Extension Specification. URL: `http://people.freedesktop.org/ ewalsh/xace.pdf`, October 2006. 38, 54

[70] Eamon F. Walsh. Application of the Flask Architecture to the X Window System Server. In *Proceedings of the 2007 SELinux Symposium*. Baltimore, MD, USA, March 2007. 38, 54

[71] Carsten Weinhold. Portierung von Qt auf DROPS. Großer Beleg, URL: `http://os.inf.tu-dresden.de/papers_ps/weinhold-beleg.pdf`, 2005. 26

[72] Robert Wetzel. An Acceleration Architecture for DOpE. Diploma thesis, TU Dresden, URL: `http://os.inf.tu-dresden.de/papers_ps/wetzel-diplom.pdf`, 2003. 18, 19