A Nitpicker's guide to a minimal-complexity secure GUI

Norman Feske and Christian Helmuth
Technische Universität Dresden
{feske,helmuth}@os.inf.tu-dresden.de

Abstract

Malware such as Trojan Horses and spyware remain to be persistent security threats that exploit the overly complex graphical user interfaces of today's commodity operating systems. In this paper, we present the design and implementation of Nitpicker—an extremely minimized secure graphical user interface that addresses these problems while retaining compatibility to legacy operating systems. We describe our approach of kernelizing the window server and present the deployed security mechanisms and protocols. Our implementation comprises only 1,500 lines of code while supporting commodity software such as X11 applications alongside protected graphical security applications. We discuss key techniques such as client-side window handling, a new floating-labels mechanism, drag-and-drop, and denial-of-service-preventing resource management. Furthermore, we present an application scenario to evaluate the feasibility, performance, and usability of our approach.

1 Introduction

Spyware and Trojan Horses are crucial security problems of today but they are not widely addressed by developers of operating systems (OS) and graphical user interfaces (GUI), who are afraid of breaking compatibility with existing commodity applications.

In former times, applications behaved friendly and cooperated with each other to please the user. GUIs designed twenty years ago do everything to facilitate communication among applications without bothering application programmers with security protocols. Commodity GUIs of today still expect applications to be nice. Unfortunately, the security assumptions about the good behavior of applications do not hold anymore. Today, computers are ubiquitously connected to networks and it is usual practice to download applications and applets from untrusted sources. This practice brings along the risk of falling prey to malware. Once installed on a system, malicious code can furthermore exploit the networking facilities to communicate.

In 2004, the overdue discussion about what OS designers can do about GUI security was brought back to life by J. Shapiro et al. with the presentation of the EROS Trusted Window System (EWS) [22]. However, the presented approach supports only a dedicated set of applications compiled for the particular platform while the broad range of mass-market software is not available.

In contrast, current OS and virtualization techniques provide powerful ways of securing applications while retaining support for legacy software. Xen [11] provides coarse-grained isolation of concurrently running virtual machines and heads toward multi-level security. Virtual-machine-monitor (VMM) technology gets an additional spin by Intel's release of Virtualization Technology [5] and AMD's release of the Pacifica specification. Microsoft's Next Generation Secure Computing Base (NGSCB) [20] and its recently announced hypervisor architecture [6] will use these technologies to separate trusted services from a monolithic Windows OS by partitioning the underlying platform.

The isolation mechanisms implemented by these platforms enable the safe execution of security applications aside of a sandboxed legacy OS. With our work, we complement these techniques by an extremely minimized secure GUI to fight security problems such as spyware and Trojan Horses. Instead of discussing system policies such as session management, which vary a lot among different platforms, we focus on low-level mechanisms that are applicable to a large variety of target platforms. We address the following classes of malware:

Applications spying on other applications: No application should be able to obtain sensitive information from another application via the GUI, for example by taking a screen shot. Section 2.2 tells you about our mechanisms to isolate client applications while still supporting existing commodity software.

Applications observing the user: Spyware should not be able to sample key strokes and mouse actions. In Section 2.3, we show how we make applications receive only those input events that are meant for the particular application.

Applications obtaining data from the user by fraud:

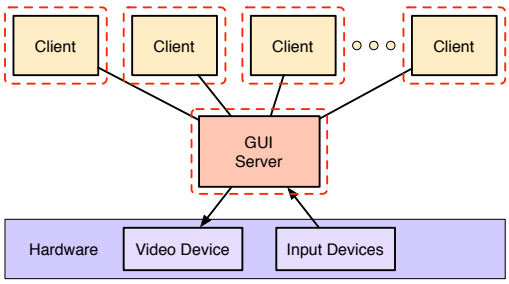

Figure 1. General scenario.

Trojan Horses mimic trusted applications to wrest sensitive information from the user. In Section 2.4, we pull out our weapon to fight Trojan Horses.

Denial of service of the user interface: Current overly complex GUIs are prone to denial-of-service attacks driven by client applications. Such applications can infinitely grab the mouse pointer or open a full-screen window that captures all input events to make the user interface inaccessible. Furthermore, a GUI server that allocates resources on client request is at the mercy of its clients not to exhaust the available resources. When relying on high availability, malicious or faulty applications are an unbearable risk. Our approach for managing server-side resources is explained in Section 2.7

After presenting the design of our minimal-complexity GUI server that we call Nitpicker in Section 2, we outline its large application space by a selection of potential target platforms in Section 3. Our implementation for one particular platform is described in Section 4 and evaluated in Section 5. After revisiting additional related work in Section 6, we conclude the paper with Section 7.

2 Design for Nitpickers

Figure 1 provides an overview about the components that we discuss. One GUI server exclusively accesses the hardware and serves a number of clients. All clients and the GUI server are executed within dedicated isolation domains, enforced by the underlying OS kernel. In the following, we refer to these client applications as clients.

High complexity of system software implies a high risk for the robustness and security of dependent software components. The GUI server is a crucial part of system software on which end-user applications absolutely need to rely. Consequently, the design of the GUI server described in this section follows the principle of minimal complexity. We only integrate mechanisms for enforcing security and a core of mandatory GUI functions that cannot be implemented in another system component. As we will see, this condition applies for a surprisingly low amount of GUI functionality.

All desired functionality that is not provided by the server must be implemented in each client. This does not mean 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.

Let us begin our survey with the discussion of one key decision leading to the low complexity of Nitpicker.

2.1 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 extremely high complexity, for example, the Qt toolkit consists of more than 300,000 lines of C++ code.

There are window systems that implement widget handling on the server side, for example Fresco [3], and DOpE [14]. While server-side widget handling has advantages with regard to responsiveness and consistency, the authors of EWS state that complex widget management should be implemented on the client side. From the security perspective, this declaration is valid because widget toolkits do not enforce security policies at all. Consequently, EWS only provides windows but no buttons, menus and other widgets. In DOpE however, a window is implemented as a widget. This leads to the question of why not implement the window widget on the client side as well? Should a window enforce a security policy and provide means to protect accessibility?

J. Shapiro et al. [22] answer the latter question with yes. They argument that 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 accessibility 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 (X11) 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, our usage of the term "client-side window handling" refers to managing windows by each client itself.

2.2 Buffers and views

In this Section, we describe Nitpicker's mechanisms for representing graphical applications on screen while enabling the client-side implementation of windows and other widgets. Nitpicker deals with only two kinds of objects: *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 Nitpicker via shared memory. The pixel format of every buffer is equal to the pixel format of the current screen mode. Nitpicker does not perform color-space conversion because converting color spaces is no security-relevant functionality. Each client must be aware of the pixel format provided by Nitpicker.

Nitpicker 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. Nitpicker provides a much simpler object type that we call *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 viewport on the buffer by specifying a vertical and horizontal offset. Note that there may exist multiple views on one and the same buffer whereas each view can have an individual size and position on screen and presents a different region of the buffer. Each time a client changes the content of a buffer, it informs Nitpicker about the affected buffer region. Nitpicker then updates all views that display the specified buffer region. 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 entitled 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 views, the global stacking order of all views is only known to Nitpicker. This fulfills our initial security goal that one client can neither obtain information about other clients nor manipulate other clients.

2.3 Input handling

The buffers and views mechanism presents clients on screen and let them communicate to the user. For enabling the save communication in the other direction—from the user to the client—Nitpicker needs to route mouse and keyboard events to the addressed client while hiding the user input from other clients (e. g., 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. Nitpicker 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 scancode. 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, Nitpicker 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 client-side window handling. 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. We ensure that the referred window is able to catch all events that belong to the resize operation.

There are two magic keys that are exclusively in use by Nitpicker 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 let the user pick a client to lock out from the Nitpicker session. It is the emergency brake for a misbehaving client. The other key that we call *X-ray* will be explained in the following Section.

2.4 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, we address the 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 X11 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 targeted by Trusted X [13], 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 in Nitpicker 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 [12], 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 fullscreen window could present a picture of a trusted client, including the faked labeling information. This example illustrates that we 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. We desire a more noticeable way to present labeling information.

Another idea to preserve a unique capability for presenting labeling information is to cut the color space into two regions. 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.

For Nitpicker, we combine the reserved area and dimming techniques with a new label placement strategy: *Floating labels*. Nitpicker dimms 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, Nitpicker analyzes the arrangement of visible views and places all labels in a way that they are visible. Nitpicker 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 while always covering a part of the view's content. All labels are drawn in the same color as 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 faze 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 MLS systems, Nitpicker could tint unfocused views of different classification levels with different colors. For application areas where extremely paranoid security policies are needed, the dimming may completely blend out the content of unfocused clients. Just for the sake of nitpicking, we must consider that the dimming technique does not prevent Trojan Horses from faking trusted clients that use only dark colors. Still, the view borders and labels cannot be faked.

There are other application areas, where productivity is needed. For example, a user wants to watch a full-color movie while coding. In this scenario, dimming would reduce inspiration and consequently, lower his efficiency. For this, we introduce a way to toggle two modes by using a magic key. In *Flat mode*, no labels, no borders and no dimming is displayed. The only visible part of Nitpicker is a red shaded bar at the top of the screen that displays the labeling information of the focused view. In *X-ray mode*, dimming, floating labels, and the view borders are active. The bar at the top of the screen is shaded gray, 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, for highly secure systems, switching to Flat mode could be disabled.

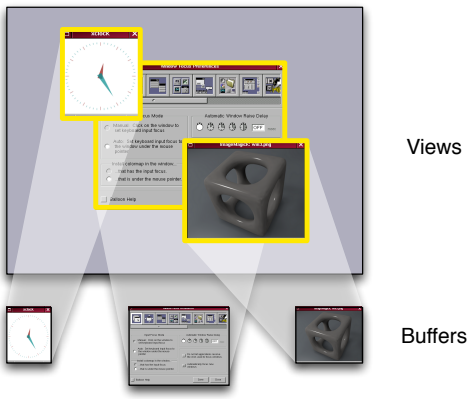

Figure 2. One buffer per view.

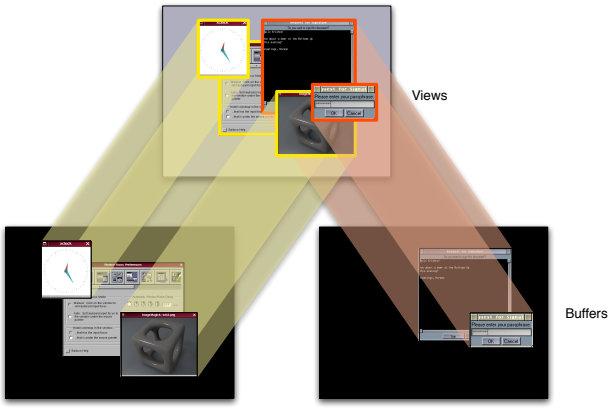

Figure 3. Two window systems as clients.

2.5 Bring Nitpicker to life

After describing raw mechanisms, we outline 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 2 illustrates this approach, which basically corresponds to the implementation of EWS and Apple Quartz. The obvious advantage is simplicity. The performance of moving windows and changing the stack layout is great because no rerendering of windows is needed in such situations. The performance only depends on the blitting operation of Nitpicker. For resizing windows, reallocation of the buffer and a new render process is needed. Of course, the buffer-per-view approach implies high memory requirements. 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 argument that modern graphics cards provide an abundance of memory. On the other hand, one can argue that graphics memory should be available to applications instead of the window system. Additionally, when looking at mobile platforms and embedded devices, graphics memory is a precious resource.

Another way to deploy Nitpicker's mechanisms is to use only one buffer and render a complete windowed desktop into this buffer. The client is indeed 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 same time. If all *state changes* of client windows are consistently applied to Nitpicker's views, the stacking layout of the views is equal to the stacking layout of the client window system.

If multiple client window systems come into play as illustrated in Figure 3, each client window system manages its local desktop and its local stack of views while isolation between clients is preserved. Nitpicker alone knows the global stacking order that consists 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. This technique and a number of applications are described in more detail in [15]. With regard to memory consumption, this technique scales well with the number of windows on screen because all windows of one Nitpicker 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.

Nitpicker enables the usage of both techniques by different clients at the same time. A Nitpicker 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 thus, providing convenience to application programmers at the cost of increased complexity.

2.6 Drag-and-drop

Drag-and-drop is a widely used paradigm to transfer data from an application to another by dragging an item with the mouse. Nitpicker 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

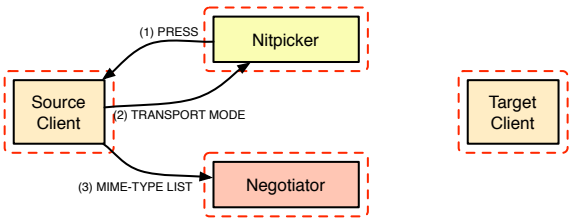

Figure 4. Picking an item.

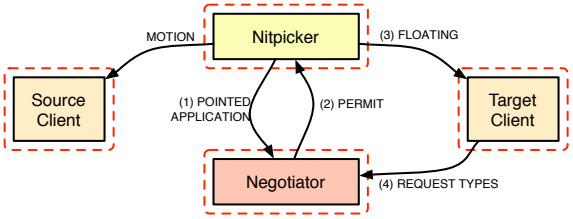

Figure 5. Dragging the item.

policy, described in Section 2.3. More challenging is the use of drag-and-drop for establishing communication between different clients of Nitpicker.

In [22], 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. In this section, we present a drag-and-drop protocol that is derived from EWS and refined for Nitpicker.

Communication via drag-and-drop is restricted by the action of the user *and* global policy, for example the permitted information flow in a MLS system. We introduce a dedicated component—the *negotiator*—for representing the global policy.

Our 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): 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 Nitpicker 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 5): 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 border, Nitpicker tells the *negotiator* about the new *pointed client* (1). In turn, Nitpicker receives the policy decision about the information flow from the source to the target client (2). If permitted and the user moves the mouse, Nitpicker sends motion events to the source client and *floating* events to the potential target client (3). When a

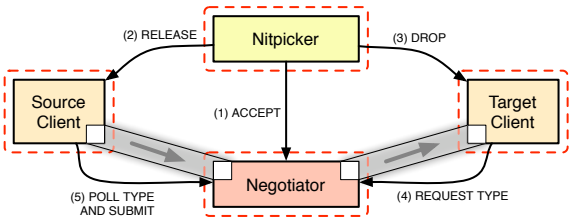

Figure 6. Releasing an item.

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 Nitpicker. 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 6): When the mouse button is released, Nitpicker tells the negotiator that the user accepts the transaction (1). Subsequently, Nitpicker 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.

Nitpicker neither deals with type negotiation, nor implements the policy of information flow, and is not involved in payload transfer. The whole job of Nitpicker 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 Nitpicker's mechanisms are the same for all 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 requests. However, the proposed protocol keeps the involved clients anonymous and the channel is bounded by user action.

Beside 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, we do not discuss cut-and-paste within this paper.

2.7 Resource management

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

In the case of Nitpicker, critical server-side resources are the heap that holds client-specific data structures and the processing time that is consumed to serve a client. For providing robustness against heap exhaustion, a client must donate memory to Nitpicker. If a client requests the creation of a new view, Nitpicker returns an out-of-memory error. The client resolves this error by donating a memory region to Nitpicker. Thereafter, the memory area is accessible for Nitpicker only and the client cannot revoke this memory area from Nitpicker. Nitpicker uses this memory region exclusively for this particular client and frees the memory region on client exit.

Nitpicker consumes CPU time on request of its clients. Serving a buffer refresh request requires a significant amount of processing time and bus bandwidth because pixel data must be copied. Long-taking atomic operations may delay other clients and the used CPU resources could wrongly be accounted on Nitpicker. With our past research on the DOpE [14] real-time window server, we addressed these problems by introducing *redraw dropping* and *redraw splitting*. These techniques can be applied to Nitpicker as well.

3 Target platforms

The design of Nitpicker is applicable on all platforms that provide isolation between security domains. These platforms reach from virtual machine monitors (VMM) providing coarse-grained isolation of virtual machines (VM) to fine-grained multi-server OSs.

VMMs such as VMware [9] enable the execution of multiple guest OSs on one host OS at the same time. However, there exists no convenient way of user interaction with multiple VMs. The graphical output of different VMs is either displayed in separate windows of a host window system or in separate virtual consoles. With Nitpicker running on the host OS and exporting its client interface as a network service, guest OSs are able to use the view mechanism via a virtual network device. The guest OS could run a service that forwards window state changes to Nitpicker and therefore, achieves a tight integration of its GUI with other guest OSs running on separate VMs. Still, isolation between different guest OSs is maintained.

Hypervisor architectures such as the Xen VMM [11] support the execution of multiple isolated (para-) virtualized OSs. In [16], the developers of Xen describe techniques for the reuse of legacy drivers in dedicated virtual machines and give an outlook toward Xen as multi-level secure system. As Xen's primary focus is server consolidation, the supported device families are network and mass-storage devices. However, the mechanisms for supporting legacy drivers are suitable for video, too. Nitpicker, executed within an I/O isolation space, could be one special virtual video device that multiplexes the graphical output of multiple VMs and performs the access to the physical video hardware. The upcoming Windows Virtualization Architecture [6, 20] envisions a microkernelized hypervisor that executes drivers within de-privileged protection domains alongside unmodified legacy OSs and protected secure applications. According to [7], such applications will have only limited GUI support (e. g., no support for overlapping windows). Nitpicker however, could provide a seamless integration of such applications with the full-fledged GUI of the Windows OS while preserving the required isolation.

Nitpicker could enhance the application area of remote desktop protocols (RDP). For example, German embassies deploy the Secure Inter-Network Architecture (SINA) [8] for processing classified information. The SINA Thin-client as end-user component handles different classification levels by separate RDP sessions on virtual consoles—each virtual console providing a full-screen GUI. Nitpicker could make the user interaction with different classification levels more natural by integrating them into one desktop. Nitpicker's capability of tinting views of different classification levels with different colors and the floating-labels mechanism provide a convenient way to distinguish different domains.

Multi-server OSs such as EROS [21] deploy confinement to decompose OS functionality into small components that can be evaluated independently. For EROS, there exists EWS [22] as GUI. In comparison to EWS, Nitpicker provides the advantages of lower complexity and a higher flexibility for applications. Compatibility to EWS could be maintained by executing EWS as client of Nitpicker. Different compartments could even use dedicated instances of EWS. However, the biggest strength of Nitpicker—the support for legacy window systems as clients—remains largely unused because EROS provides no means to execute legacy software. Coyotos [2] as the successor of EROS will provide support for legacy software by a Linux emulation layer. When Coyotos supports X11, Nitpicker would do well with integrating X11-based legacy software with EWS windows.

4 Implementation

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

The basis platform was provided by the L4/Fiasco [18,

19] microkernel that enables us to safely execute one or multiple L^{4}Linux kernel servers [17] along with native L4-based applications at the same time. All L4 applications—including L^{4}Linux—are running in user mode and are executed within isolated protection domains. Communication between protection domains is performed by L4 interprocess communication (IPC) only. For handling mouse and keyboard input, we ported the input subsystem of the GNU/Linux kernel version 2.6 to L4/Fiasco. We realized the graphical output 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.

To evaluate the support for legacy applications, we ported two legacy window systems—DOpE and X11—to Nitpicker (Figure 7). For porting DOpE, we added 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. For the support of X11, we implemented drivers for screen (400 LOC) and input (250 LOC) as modules for XFree86 [10]. We propagate window events from X11 to Nitpicker by using a custom X11 client (200 LOC) that scans all windows and registers itself as an event handler for window events at the root window. We did not require changes of the X server at all. X11 and DOpE client applications are executable in this setup without modification. Figure 7 displays Nitpicker in X-ray mode with X11 and DOpE as clients. Note that the translucency effect, which is implemented by DOpE does not display X11 windows. DOpE has no access to data of the X11 session and therefore, cannot incorporate X11 windows for the computation of the translucency effect.

5 Evaluation

At the beginning of Section 2, we introduced low complexity as first-grade design goal. Our implementation of Nitpicker consists of merely 1,500 human-written lines of C code (LOC). This is only a fraction of EWS's size (5,400 LOC) and an order of magnitude smaller than Trusted X (30,000 LOC) and X11 ($>$ 80,000 LOC without drivers).

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 eases 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 X11 on Nitpicker exactly in the same way as on native X11.

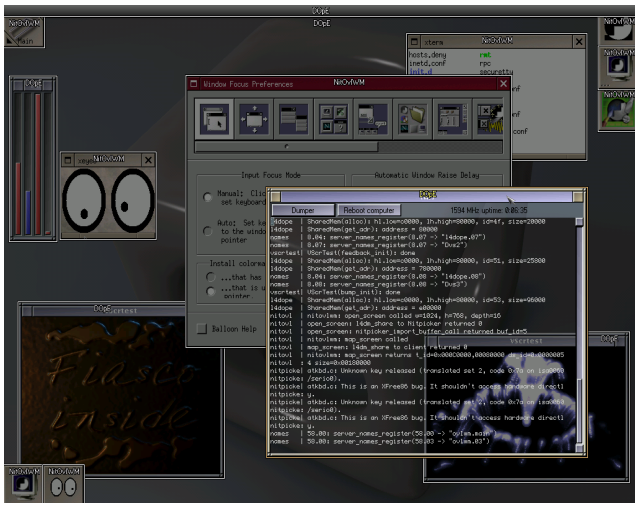

Figure 7. Screenshot of Nitpicker.

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 lines of IDL code, the generated stub code comprises about 1,000 lines of C code. Comparing this number to the complexity of the human-written code highlights the critical role of compilers and tools for secure systems.

We estimated Nitpicker's performance by comparing the CPU demand of DOpE running as Nitpicker client against native DOpE. In both scenarios, we 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 for another DOpE window. For the tests, we used an Intel Celeron PC clocked at 900 MHz. Nitpicker does not require additional copying of pixels. We 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. This is caused by the *dimmed* blitting function that is not optimized for performance, yet. When switching to Flat mode, the additional load drops to less than one percent. Although this is just a showcase, the observed low overhead matches our precedent estimations and indicates the feasibility of Nitpicker's design with regard to output performance.

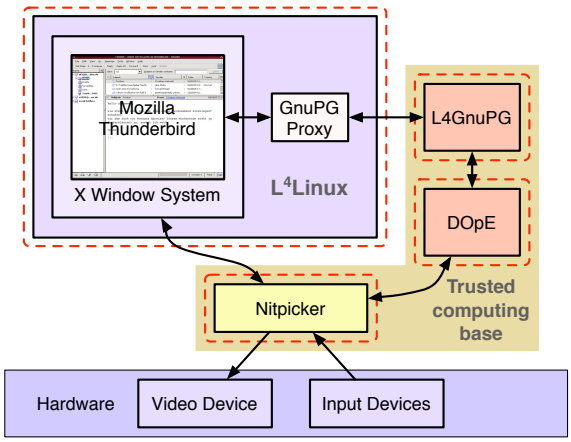

Figure 8. Application scenario.

5.1 Application scenario

For highlighting the benefit of Nitpicker in conjunction with widely used commodity applications, let us present an application scenario.

Mail readers such as Mozilla Thunderbird are popular because of their rich features (e. g., spam filtering, powerful searching functions) and good usability. This convenience comes at the cost of an enormous complexity of the application and the needed OS support. With regard to the confidentiality of private keys for signing emails, such applications are a nightmare. For the concrete example of using Mozilla Thunderbird on the GNU/Linux platform, the complexity of the Linux kernel, the privileged daemon processes, the X window system, Mozilla Thunderbird and concurrently running user processes of the same user accumulate to millions of lines of code that potentially put the secrets of the user at risk.

In fact, only a small fraction of this code—the GNU Privacy Guard (GnuPG) [4]—actually needs the private keys for operation. We ported GnuPG to the L4 platform, creating L4GnuPG, and complemented it with a trusted text viewer. We interfaced L4GnuPG with Thunderbird by creating a L^{4}Linux proxy process that redirects Thunderbird's calls of GnuPG to L4GnuPG. L4GnuPG uses DOpE as its widget set, which is running within an isolated address space. In this scenario, L4GnuPG is the only process in the whole system that can access the confidential signing key of the user. Figure 8 presents an overview about the components of this scenario. When the user activates the signing function of Thunderbird, our L^{4}Linux proxy process transfers the email to L4GnuPG. L4GnuPG presents this email in a DOpE window that is displayed within a corresponding view of Nitpicker. The user can now decide to sign the email or cancel the operation. If he decides to sign the email, L4GnuPG requests a pass-phrase, signs the email and transfers the result to Thunderbird via the L^{4}Linux proxy process.

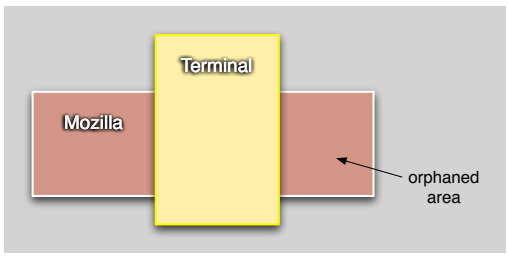

Figure 9. Orphaned area on screen.

In the presented scenario, the confidentiality of the signing key depends on only 105,000 LOC including L4/Fiasco (15,000 LOC), trusted L4 services (35,000 LOC) and L4GnuPG (55,000 LOC). The isolation of the legacy X11 window system and the GUI of the trusted application depends only on the L4/Fiasco kernel and Nitpicker (1,500 LOC). We obtain the powerful features and great usability of a commodity application while extremely minimalizing the trusted computing base (TCB) of a security-sensitive function with regard to its GUI. The scenario underlines the biggest strengths of Nitpicker: low complexity and the support of legacy graphical user interfaces.

5.2 Current limitations

After presenting the strengths of Nitpicker, we review the limits of our current implementation.

Nitpicker attaches exactly one label to each view. There are view layouts that leave orphaned areas unlabeled on screen (Figure 9). Although the dimming technique in X-ray mode prevents confusion about the focused view, a shading policy as described in [12] could be deployed to encounter such cases by blanking out orphaned areas. This will be implemented in a future version.

Nitpicker performs graphical output via software graphics routines. Making hardware-accelerated graphics usable by Nitpicker and untrusted clients at the same time is a challenging problem and will be an object of our future work.

6 Related work

This section complements Section 3 with related work about techniques and approaches that inspired the design of Nitpicker.

J. Epstein addressed the problem of expressive and unique labeling of windows for the Trusted X11 in [12]. Beside estimating different labeling techniques for marking classified information, he introduces a technique to detect and blank out orphaned window areas. The dimming of non-focused windows was inspired by Apple's Exposé feature in Mac OS X. J. Shapiro described the dimming

of unfocused windows for EWS in [22]. Apple Quartz is an existing implementation of client-side window handling. It is used for integrating X11 with the Aqua GUI of Mac OS X. The X server, provided by Apple, enables the use of any legacy window manager (e. g., WindowMaker) for managing the X windows. Apple significantly changed the XFree86 source base. Each X window is rendered into a dedicated pixel buffer. In contrast, we did not change the X server at all and use only one buffer for all X windows.

7 Conclusion

With the work described in this paper, we hope to advance the discussion of GUI-related security mechanisms of operating systems a step further. With Nitpicker, we minimized the complexity of the security-sensitive GUI server to only 1,500 lines of C code by consequently moving non-security-related functionality from the GUI server to the clients. The achieved low complexity is only a fraction of existing approaches.

When running on a host OS that provides isolated protection domains, Nitpicker maintains the isolation of its clients to prevent applications from spying at each other by exploiting GUI server functionality. In contrast to todays 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 a high flexibility of client GUIs. This enables the user to identify and disarm Trojan Horses. Thanks to the extremely low complexity and the deployed resource management, Nitpicker is robust against denial-of-service attacks driven by client applications and thus, can guarantee the service of sensitive client applications with regard to their GUI. At that time, we facilitate the support for existing legacy applications using the flexible buffers and views technique. This enables the further use of commodity window systems and their application alongside the safe execution of low-complexity security-sensitive applications.

Our implementation proves the feasibility of the presented design. We frequently use our custom software stack for public talks and lectures. At publishing time of this paper, the implementation of Nitpicker will be publicly available under the terms of the GNU General Public License.

We thank Jonathan S. Shapiro for sharpening our minds with regard to (more or less) covered channels and resource management. Furthermore, we want to thank Alexander Warg for the valuable and frequent discussions during the design of Nitpicker.

References

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

[2] Coyotos website. URL: http://www.coyotos.org.

[3] Fresco website. URL: http://www.fresco.org.

[4] GNU Privacy Guard website. URL: http://www.gnupg.org.

[5] Intel Vanderpool technology website. URL: http://www.intel.com/technology/computing/vptech/.

[6] Microsoft's Virtualization Architecture. TWAR05013 at WinHEC 2005.

[7] NGSCB presentation at WinHEC 2004. URL: http://download.microsoft.com/download/1/8/f/18f8cee2-0b64-41f2-893d-a6f2295b40c8/TW04008_WINHEC2004.ppt.

[8] SINAvpn website. URL: http://www.sinavpn.com.

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

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

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*, Oct. 2003.

[12] J. Epstein. A prototype for Trusted X labeling policies. In *Proceedings of the 6. Annual Computer Security Applications Conference*, Dec. 1990.

[13] J. Epstein, J. McHugh, H. Orman, R. Pascale, A. Marmor-Squires, and B. D. et al. A high assurance window system prototype.

[14] N. Feske and H. 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, Dec. 2003.

[15] N. Feske and C. Helmuth. Overlay window management: User interaction with multiple security domains. Technical Report TUD-FI04-02-März-2004, TU Dresden, 2004.

[16] K. Fraser, S. Hand, I. Pratt, and A. Warfield. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS 2004)*, Boston, MA, Oct. 2004.

[17] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, Sept. 1998.

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

[19] J. Liedtke. L4 reference manual. Technical report, Sept. 1996. RC 20549, IBM T. J. Watson Research Center.

[20] M. Peinado, P. England, and Y. Chen. An Overview of NGSCB.

[21] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, Apr. 1999.

[22] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS Trusted Window System. In *Proceedings of the 13th USENIX Security Symposium (2004)*, pages 165–178, 2004.