TrustGraph: Trusted Graphics Subsystem for High Assurance Systems

Hamed Okhravi
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
Email: okhravi2@illinois.edu

David M. Nicol
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
Email: dmnicol@illinois.edu

***Abstract*—High assurance MILS and MLS systems require strict limitation of the interactions between different security compartments based on a security policy. Virtualization can be used to provide a high degree of separation in such systems. Even with perfect isolation, however, the I/O devices are shared between different security compartments. Among the I/O controllers, the graphics subsystem is the largest and the most complex. This paper describes the design and implementation of TrustGraph, a trusted graphics subsystem for high assurance systems. First, we explain the threats and attacks possible against an unsecured graphics subsystem. We then describe the design of TrustGraph, the security principles it is built upon, and its implementation. Finally, we verify our implementation through different levels of verification which include functionality testing for simple operations, attack testing for security mechanisms, and formal verification for the critical components of the implementation. An analysis of the graphics API covert channel attack is presented, its channel capacity is measured, and the capacity is reduced using the idea of fuzzy time.**

***Keywords*-Trusted Graphics; Multi-Level Security; Virtualization; Formal Verification; Covert Channel Analysis**

I. Introduction

High assurance secure systems require strict compliance of the system activities with a security policy. Multiple independent levels of security (MILS) and multilevel security (MLS) systems are two of such systems. A MILS policy usually requires strict isolation between different compartments of the processes and resources in the system with little or no interaction between them. MLS systems on the other hand allow limited communication between different security levels according to the security policy (e.g. Bell-LaPadula [8] and/or Biba [9] policy). High assurance MILS and MLS workstations require strong compartmentalization of the system to ensure that no information leakage or interference can happen between different security levels. Hence, an attractive approach for building such a system is virtualization. A virtual machine monitor (VMM or hypervisor) provides an abstraction of the system hardware to the virtual machines (VM) running on top of it while isolating each VM from another. In MILS or MLS systems each level of security is designated a separate VM. The VMM then enforces the security policy by blocking any interaction between the virtual machines or allowing limited communications according to the policy. When providing strict isolation between the VMs, VMM is also referred to as a "separation kernel."

An inherent issue with virtualization is the problem of I/O. Different approaches have been proposed for virtualizing I/O [20]. In modern systems, three main approaches exist for I/O virtualization. The first approach is to have a separate I/O device for each VM. In this case, each VM has its own subsystem (driver and manager) to interact with its I/O device, hence achieving strong isolation between the VMs. However, this model imposes additional hardware costs to the system and if the number of different security compartments is large, it soon becomes impractical (e.g. a system with ten different security compartments requires ten separate displays.) The second approach is to have one copy of each I/O device with all of the I/O drivers residing in the VMM. The VMM then presents a virtual device to each VM. In this case, the VMs share each I/O device and its driver. KVM [25] uses this model. The downside is that this model makes the VMM complex and large. The third model for I/O virtualization is to have a separate I/O partition (a.k.a. the privileged partition or dom0) on top of the VMM which controls the I/O operations. Any request for the I/O from a VM is sent to this partition by the VMM and the results are forwarded back to the VMs. This model is used by Xen [7].

In the latter two approaches of I/O virtualization, the subsystem that controls an I/O device is shared between the different VMs. Sharing subsystems has an inherent security problem; namely, information can leak and different security levels can interfere through these I/O subsystems. Some of the I/O controllers are simple and tiny which reduces the chance of interference (e.g. the keyboard and mouse drivers). A graphics subsystem, on the other hand, causes many security concerns. It usually consists of a large piece of code which handles the graphic operations and builds the display output (e.g. the X Window System). The inherent complexity of the graphics subsystem and the fact that it handles data from different security levels, imposes a high risk of information leakage and interference. In fact, the applications frequently use the graphics subsystem resources as a means of communication, not regulated by the security policy [24].

This paper describes the design and implementation of TrustGraph, a trusted graphics subsystem for high assurance systems. In TrustGraph, the entities (resources) in the graphics subsystem are labeled with appropriate security tags to prevent unauthorized communication. Moreover, the methods and operations are secured so that they comply with the security policy.

TrustGraph can be deployed in a single secure operating system or in a virtualized architecture. The implementation is evaluated against the attacks and information leakages that are possible under the vanilla graphics subsystems, but are prevented in TrustGraph. In addition, the critical parts of the implementation are verified using formal method techniques (using ACL2). In fact, we have identified several flaws in our initial implementation by formal verification which were corrected. Moreover, an analysis of the potential covert channels using the dynamics of the graphics subsystem is performed and the capacity of such channels is estimated and then reduced in TrustGraph using the idea of fuzzy time. We have also used TrustGraph in a proof of concept end-to-end virtualized system.

The contributions of this work are as follow:

- Enumerating and describing different classes of attacks possible using the API of a graphics subsystem.
- Design and implementation of a secure graphics subsystem on top of a simple and tiny graphics library.
- To the best of our knowledge, TrustGraph is the first graphics subsystem with some of its critical components formally model checked (the policy enforcement and the window manager logic)
- It is also the first graphics subsystem that reduces the channel capacity of the graphics API covert channel attacks.

The rest of the paper is organized as follow. Section II gives a background on the graphics subsystem and the terms and concepts usually used in its context. The background material presented should be enough for those without the knowledge of graphics systems to understand the rest of the paper. Section III describes the threat model as well as the types of attacks and information leakage possible in a graphics subsystem. The details are presented on how an unsecured graphics I/O can violate the security policy and leak sensitive data. We discuss the design of TrustGraph along with the mechanisms used to counter the security threats in section IV. The implementation details are provided in section V. Section VI describes the evaluation of TrustGraph against known attacks as well as the formal verification of the critical components of the implementation. The analysis of the covert channels using the graphics API is also presented in this section. Finally, we discuss the related work in section VII before concluding the paper in section VIII.

II. Background

A graphics subsystem is a software system responsible for providing a graphical user interface (GUI) for the applications and building the display output through the graphics card. It allows the applications to interact with the graphics hardware through an application programming interface (API) which we simply refer to as the *interface* hereafter. The graphics subsystem also handles the inputs from the input devices such as mouse and keyboard and directs them to the appropriate application. Examples of the graphics subsystems include the X Window System [34] for Linux-like operating systems, DirectFB for Linux and embedded systems, and Quartz [38] for Mac OS X. The graphics subsystem often includes an integrated windowing system which is responsible for handling and managing the windows on the screen.

The following terms and concepts are used in the context of graphics. We use the generic terms with a bias towards those adopted by DirectFB [1].

- Window: A window is a visual area, usually rectangular, which displays the graphical outputs and accepts the inputs for an application.
- Layer: Each layer represents an independent graphics buffer in the system. Different layers are blended into the final image using the transparency information for each layer (i.e. alpha blending). For example, one layer can be used for the background, another one for an application window in the middle, and yet another one for a video playing on top.
- Surface: A surface is a reserved piece of memory (from the video card or the system memory) which holds the pixel data for a window. All drawing operations requested by the application are done directly on this piece of memory.
- Event Buffer: An event buffer holds all of the input events for a window (e.g. key strokes or mouse events).
- Data Buffer: A buffer which holds the image or video data to be displayed on a window.
- Window Manager: A piece of software that manages a set of windows. The window events such as resizing, reordering, or moving call the appropriate window manager methods. The window Manager is also responsible for redirecting the input events to the appropriate window.

In this paper, we refer to the set of all windows (*W*), surfaces (*S*), event buffers (*EB*), and data buffers (*DB*) as the *graphics resources* (*R*) (or simply the *resources*.)

Figure 1 shows the simplified architecture of a graphics subsystem. A typical sequence for establishing a GUI starts from the application creating an interface (called the main interface) to the graphics subsystem. The main interface can optionally create a data buffer to load image or video data. The main interface then creates one or more windows

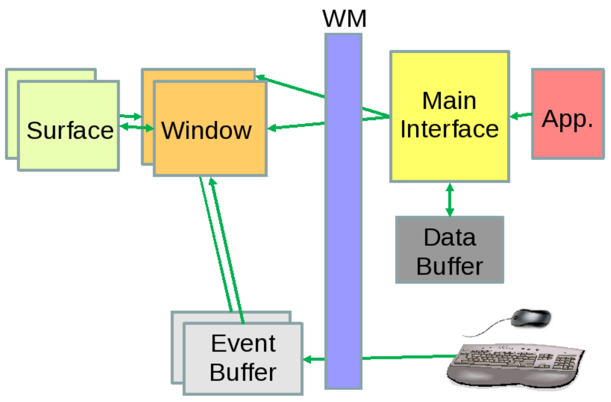

Figure 1. The architecture of a graphics subsystem

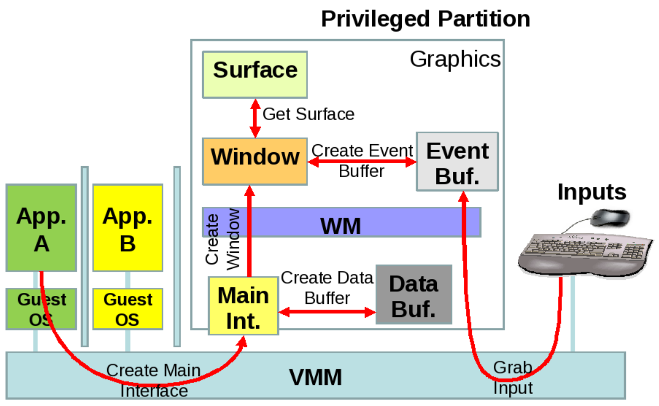

Figure 2. Label flow in TrustGraph

for that application through the window manager. Each window then creates a surface and an event buffer to hold its pixel values and the input events respectively. The window manager then forwards the input events to the appropriate application through its event buffer and eventually its main interface.

Resource creations and acquisitions are done through a set of graphics methods (*M*) (hereafter referred to as *methods*) provided by the graphics subsystem through the interfaces. For example, one method (Create-Surface) creates a surface for a window ($f : W \rightarrow S$) and another one (Get-Event-Buffer) assigns an already existing event buffer to a window ($f : W \rightarrow EB$.) A complete list of the graphics methods is provided in section IV-C1. There are also global operations (*OP*) (hereafter referred to as *operations*) which operate on a set of resources. For instance, taking a screenshot of the display is an operation which operate on the set of all windows and their surfaces to dump an image of the display. Also a graphics subsystem provides a set of drawing functions (*D*) which are called by a window to manipulate its surface ($f : S \rightarrow S$.) As an example, when an application wants to draw a rectangle on its window, a drawing function is called which gets the surface of the window and changes its pixel values to include a rectangle. A graphics subsystem (*GS*) is a 5-tuple that includes the set of all graphics resources ($R = \bigcup(W, S, EB, DB)$), the graphics methods (*M*), the operations (*OP*), the drawing functions (*D*), and a window manager (*WM*); i.e. $GS = \langle R \mid M \mid OP \mid D \mid WM \rangle$.

III. Threat Model

The threat model used in this paper assumes that the applications running at different security levels are not trusted. They have the potential to leak information and violate the security policy as a result of intentional malicious behavior or unintentional bugs or programming errors. The trusted computing base (TCB), on the other hand, includes the hardware of the system (CPU, main memory, and devices), the video card, and the logic used to provide the separation between the processes at different security levels. This logic can be a VMM (hypervisor or separation kernel) as explained in the introduction or a trusted operating system; although, it is easier to establish the isolation property of a tiny VMM than an entire operating system. The security policy can be any arbitrary policy such as MLS, MILS, or type enforcement (TE). As a proof of concept, we implement a Bell-LaPadula-like [8] security policy for MLS systems.

We assume that the only interaction between the applications and the graphics subsystem is done through the interface defined by the graphics subsystem. The subsystem itself is protected from modifications by residing in a lower software layer (i.e. in the VMM) or in a privileged virtual machine (e.g. dom0 of Xen.) The goal is to prevent the applications from violating the security policy by using the graphics subsystem to communicate.

Now we describe various attacks and leakage points which can be used to violate a security policy. A graphics subsystem is a single piece of software which handles data from multiple security levels. This makes it the weak link in the security chain. The first security issue in a graphics subsystem is that all of the resources (windows, surfaces, buffers, etc.) are security agnostic. A surface holding the graphic data from a top secret application is not different from the one holding unclassified data. The applications can dynamically bind to these resources and read the potentially sensitive data from them; thus it is possible for the applications to communicate through these resources or for an application to snoop the graphics data of another application.

Another security threat in an unsecured graphics subsystem is unsecured methods. These methods enable the applications to build their GUI and interact with the graphics hardware. There are two types of unsecured methods: those used for creating or acquiring the resources and those used to handle the inputs. The former can be used maliciously to snoop the graphics data from the security-agnostic resources while the latter can be used to sniff the input events from another window. For instance, an application can retrieve an interface to the surface of another window in a higher security level and read its pixel data (using a method such as GetSurface). Note that applications can enumerate all the windows on the display. This is necessary for facilities such as "alt-tab" or crash recovery applications. Moreover, a window that currently does not have the focus can acquire the input events (using GrabKeyboard or GrabPointer methods). As a result, the window of a malicious application which is sitting behind the other windows and is not even visible can sniff the password typed by the user on a top secret window.

Unsecured operations can also leak information. The global operations such as copy-pasting and taking screenshots (e.g. using the PrintScreen key of the keyboard) can easily leak data across the security levels.

In addition, overlapping windows can endanger the confidentiality of the system. A window can make itself transparent (or partially transparent). There are two security concerns when a lower security window sits on top of a higher one. First, a transparent window on top can get the pixel values of the window behind it, hence accessing the sensitive data. Second, there can be access control mechanisms in place which limit the visual access of the user to different windows. For instance, a face detection camera can identify the user in front of the monitor and allow or deny his access to different windows. In such a system, a low clearance user gets access to the top window with the unclassified information, while in fact he can see the content of a higher security window behind it using transparency.

Finally, the large code size and complexity makes the graphics systems such as X inherently a bad choice for trusted graphics. X was developed back at the time when computer graphics had low color depth and there was no hardware acceleration [26]. Years of enlarging the code base and adding new features to X have resulted in a large and inefficient graphics system. In fact, the code size of X is comparable to that of the kernel itself. The obsolete features and components of X exacerbate the situation. Many of such resources can be used as the communication channels not regulated by the security policy. In short, X is too large and complex for secure graphics.

IV. Design

A. Principles

TrustGraph is built upon a number of security design principles. We first explain these principles and then describe the design of TrustGraph. We explain how the design decisions comply with the security principles.

The following security design principles are used when building TrustGraph:

I. Simplicity: It is important for secure systems to be as simple as possible. Complex design, large code base, and/or unknown or unused features are the sources of vulnerabilities. Simple and small systems are easier to design, understand, and verify.

II. Complete Mediation: Access mediation must be applied to any access or communication attempt in the system in order for the security policy to be satisfied. In fact, the graphics subsystem is one of the components in which either mediation is not done or is not complete.

III. Principle of Least Privileges: Each entity in the system must have the smallest set of privileges that allows it to do its tasks unimpeded. Hence, TrustGraph must allow the applications to lower the privileges of their GUIs.

IV. Least Common Mechanism: Shared resources in the system must be as minimal as possible to avoid overt or covert communications between the subjects using those resources. As discussed earlier, it is impractical to have a different video output for each security compartment. However, TrustGraph limits the sharing of the resources to prevent such vulnerabilities.

V. Open Design: Finally, secure systems must have an open design for them to be verifiable. The design of TrustGraph is described in details to adhere to this principle.

B. Labeled Resources

For the graphics methods and operations to comply with the security policy, all of the resources in the graphics subsystem have to be labeled with a security tag. The main interfaces, data buffers, windows, event buffers, and surfaces must all be labeled with security tags. As a proof of concept, TrustGraph implements MLS levels and categories as the security tags and uses the Bell-LaPadula model augmented with declassification as the security policy. The design, however, is not limited to MLS or the BLP policy. More general security policies such as type-enforcement, role-based access control, or attribute-based access control can be used in TrustGraph.

C. Secure Methods

The next step is to secure the methods in the graphics subsystem. Two types of secure methods exist in TrustGraph: the methods used to create or acquire resources (the label-propagating methods) and the methods used to grab inputs (the input grabbing methods).

1) Label-Propagating Methods: Any method that is used to securely create a new graphics resource or to securely obtain an existing one is called a label-propagating method. These methods must check and propagate the security tags appropriately. The label-propagating methods are as follow:

- Create Main Interface: The method is used to create the main interface to the graphics subsystem. It must propagate the security tag of the application to the main interface. It receives the security tag from the operating system or the VMM.

- Create Window/Surface/Data Buffer/Event Buffer: These methods are called from the main interface. They create a window, a surface, a data buffer, or an event buffer respectively and propagate the main interface security tag to them.
- Get Window: This method is called from the specific layer interface of the main interface to acquire an already existing window. If the main interface security label does not dominate that of the window, it can result in information leakage from a higher security level into a lower one. Thus, in such a situation, the access is denied. A main interface can only read from a window if the security label of the former dominates that of the latter. If so, the window acquisition is granted to the main interface and its security label is propagated to the window. This is done because an application writes its graphical data which has the same security label as the application to its window. Before assigning the window to a new main interface, it is released from any other interface so that only one interface has access to the window at a time.
- Get Surface/Event Buffer: These methods are called from a window interface to acquire an already existing surface or event buffer. If the window has a dominating security label, the access is granted and its security label is propagated to the surface or event buffer.

A security label in TrustGraph is an MLS label which contains a *level* followed by a set of *categories*. As defined in the BLP model [8], a label dominates another one if it has a higher level and its set of categories is a superset of that of the other label. For example, the label $\langle TopSecret, \{ProjectA, ProjectB\}\rangle$ dominates the label $\langle Secret, \{ProjectB\}\rangle$. The label flow is shown in figure 2. Each arrow is marked with the method used to create or acquire the corresponding resource. The label flow is shown in the context of a virtualized system with the graphics subsystem running in the privileged partition.

All label flows are internal to the TrustGraph code except one: creating the main interface. This method must receive the application security label from the VMM (or the operating system in a non-virtualized model). This is done through a small piece of code called the "membrane." The membrane is responsible for receiving the application security label from the VMM and delivering it to TrustGraph when the main interface is being created.

2) Declassification: To implement the principle of least privileges, it must be possible for the main interfaces to create windows with dominated security labels. This ensures that if the application wants to perform a low security task, it can open a low security window. However, the problem with declassification of window is that the application can maliciously or unintentionally declassify its sensitive data. This allows other applications which did not have access to the data before, to gain such an access. Consequently, declassification must restrict the access of window to higher security resources (such as a higher level surface), yet it must not allow other applications that did not have access to the window to gain such an access. The same argument applies to a window acquiring a surface or an event buffer.

As a result, each resource is labeled with two labels: a permanent label (PL) and a declassification label (DL). When a resource acquires another resource, it is only granted access if its DL dominates the other resource's PL. Hence, no entity gains new accesses when a resource is declassified. For the main interface, DL is always equal to PL.

Therefore, the label flow is as follow. "DOM" denotes the domination as defined before.

```
Create Resource:
   {Resource PL = Creator PL;
    Resource DL = Creator DL;}

Acquire Resource:
    If (Acquirer DL DOM Resource PL)
         {Resource PL = Acquirer PL;
          Resource DL = Acquirer DL;
          Grant;}
    else
         { Deny; }

Declassify:
    Window DL = L  (only if Main interface PL DOM L)
```

Note that if declassification is not used, the system works as a simple multilevel security system (PL=DL for all resources). The PL can be viewed as the highest possible security level that the resource may contain. Since a window may still contain data with the security level as high as the application when declassified, the PL can never be lowered by the application. The DL, on the other hand, can be interpreted as the highest label which the application believes the window should have access to.

If a more general security tag is used (e.g. types or attributes), declassification should reduce the privileges of the resource being declassified, but it must not grant new access rights to other resources. For instance, if type-enforcement is used as the policy, the main interface can declassify a window to a type which has strictly smaller set of accessible types than the original type. On the other hand, the set of types which have access to the new type must remain the same or become smaller after the declassification. Similar arguments apply for attribute-based and role-based systems.

3) Input Grabbing Methods: Whenever an input grabbing method is called, all the subsequent events of the corresponding input device are delivered to the window, ignoring the focus. These methods can result in input sniffing where a window sniffs all the input events of the window under focus. The sniffer can optionally redirect the event to its proper destination after sniffing it to avert suspicion. There are typically five types of input events: key press, key release, mouse/joystick button press, mouse/joystick button release, and mouse/joystick movement.

For TrustGraph to secure the input grabbing methods, it must redirect input events to the requesting window only if it has the focus. There are four types of input grabbing methods: Grab Keyboard, Grab Key, Grab Pointer, and Grab Unselected Keys which redirect all keyboard events, specific key events, mouse/joystick events, and unselected key events to the application respectively.

The window manager always keeps track of the window under focus, so whenever an input grabbing method is called, TrustGraph checks with the window manager to make sure that the requester is under focus. If the condition is true, the input is granted to the window (*move-to-focus* model). Otherwise, the current status of the inputs is kept intact; i.e., input events are sent to the window which currently receives them. Another model for changing the focus is called *click-to-focus* in which an unfocused window may receive mouse left or right click events. In this case, upon receiving those events, the unfocused window requests the focus in order to receive key stroke events.

One attack that is not addressed by this mechanism is click-jacking where a lower security window suddenly requests focus and the user mistakenly types a few characters or clicks on the wrong window. To mitigate this attack, TrustGraph has a click-jacking prevention (CJP) feature which issues a warning before granting the focus to a window with a different security label than the current window . If the security labels are the same, however, the focus transition is done transparently. For convenience, CJP can be turned on or off at the compile time of TrustGraph.

D. Secure Operations

The global operations are implemented in a graphics subsystem to facilitate the usage and augment the system with additional functionalities. They, however, can cause information leakage in the system. The operations differ from the methods in that they have a more global scope. Two such operations exist in TrustGraph: copy-pasting and taking screenshots.

- Copy-pasting: Copying is done by setting a global container called the *Clipboard Data* through the main interface. The clipboard data includes the MIME type of the data as well as the data itself. To prevent leakages, the clipboard in TrustGraph is labeled with the same security label as the main interface. Consequently, if the interface that gets the clipboard data (i.e. pasting the data) has a dominating security label, the data will be returned. Otherwise, NULL is returned.
- Screenshots: Screenshots can be taken from the screen by pressing the PrintScreen key of the keyboard. Regardless of the focus, the entire display is dumped whenever a screenshot is taken. To prevent leakage of information, an application can only dump the pixel values of the dominated windows. Hence, whenever PrintScreen is pressed, TrustGraph gets the security label of the window which currently has the focus. It then zeroizes the pixel values of any surface that does not have a dominated label in a copy of their surfaces and constructs the screenshot using the copy surfaces. This ensures that no application can get the pixel values of a higher application by taking a screenshot from the entire display.

By controlling all the methods and operations and validating their compliance with the security policy, TrustGraph follows the principle of complete mediation.

E. Window Manager

The window manager controls a set of windows called the window stack. If a window is resized, moved, reordered, or closed, the appropriate method of the window manager is called to rearrange the window stack.

In section III, we explained how overlapping windows pose a threat to trusted graphics. Lacking security controls, a lower security level window could in principle read the pixel values from a higher security one behind it. It might also evade visual access control mechanisms. As a result, the security label dominance imposes the same strict ordering on the windows on the screen. The window manager of TrustGraph imposes this ordering on all windows. The methods used for inserting and reordering windows are modified to always preserve the window ordering.

The ordering is done based on PL, not DL. If the ordering had been based on DL, another window with higher DL than the current window could have been positioned on top of it. Nevertheless, if the window on top had a lower PL, it could snoop higher security level data, resulting in information leakage. Note that when the ordering is done based on PL, a window can snoop data with higher security level than its DL. However, since all the resource acquisition methods check the PL before granting access, this data cannot be leaked to any other application and the system is secure.

If the windows have incompatible security labels (i.e. neither label1 DOM label2, nor label2 DOM label1), they cannot overlap at all. They can only float on the display as separate rectangles without one sitting on top of another one. If the user tries to overlap the windows, the moving window will not move any further than the edge of the other window. Although these measures impose restrictions on the user interactions, they do not make the system unusable. For instance, if a user requires a large window for one application, he can minimize incompatible windows or drag them to the corner of the display. We were able to work with the system despite these restrictions without much inconvenience. The location restrictions can be exploited to form a covert channel between the windows. We discuss the techniques to mitigate graphical methods covert channel attacks in section VI-D.

V. Implementation

TrustGraph is built on top of an existing graphics and windowing software to avoid the redundant effort to code the basic graphics functionality. Although X is the default graphics software in the Linux operating system, it was not chosen for this purpose due to its large code base, complex design, and inefficiencies. Instead, we have modified DirectFB to build TrustGraph. DirectFB is simple and lightweight and it has small overhead. Hence, it is often used in embedded systems. Unlike X, it does not use the client-server model which adds to the complexity of the graphics software. By choosing a simple base graphics system, we adhere to our first design principle: simplicity.

TrustGraph is implemented by modifying DirectFB version 1.2.0. The code size of DirectFB is about 40,000 LOC (lines of code) with a default window manager of about 3,800 LOC. This is significantly smaller than X's 1,837,000 LOC. The entire implementation of TrustGraph requires less than 3,000 LOC of fresh code and modification.

To implement labeling, the following resources are augmented with security labels: IDirectFB, IDirectFBDataBuffer, IDirectFBWindow, IDirectFBSurface, and IDirectFBEventBuffer. Each label comprises an integer security level in the range of $[0, 255]$ and a set of up to five different categories from the set $\{c0, c1, ..., c255\}$.

All of the resource creation and acquisition methods are modified to propagate and check the labels. The input grabbing methods are also modified to ensure focus before redirecting input events to a window. These methods are part of the default window manager. The *wm_grab* method of the default window manager handles all input grabbing requests. CJP is also implemented by modifying the *wm_request_focus* method.

For secure copy-pasting, the *ClipboardData* structure is augmented with a security label. In addition, two methods of the main interface have been modified to enforce the security policy. The *SetClipboardData* method is modified to set the security tag of the ClipboardData to that of the main interface that perform a "copy." The *GetClipboardData* method is also modified to provide the *ClipboardData* to any dominating security label and deny any other request.

Screenshots are handled in the input module of DirectFB. Whenever the PrintScreen key of the keyboard is pressed on any window, the input module filters that event and dumps the display by calling the function *dump_primary_layer_surface*. In order to secure screenshots, this function is modified to zerioze the pixel data of non-dominated windows.

To implement strict security-label-based ordering, the three main functions of the window manager, *wm_add_window*, *wm_restack_window*, and *wm_remove_window*, are modified. Events on the windows call one of these functions to change the window ordering.

A. Compatibility

TrustGraph provides backward compatibility with the X applications using the XDirectFB library. This library enables the applications developed for X to run seamlessly over DirectFB or TrustGraph. It is also possible to develop native applications or to port the existing X applications to use the TrustGraph (DirectFB) API directly. Many applications have already been ported to the DirectFB API, including Mozilla[2].

B. End-to-End Implementation

As a proof of concept, we have implemented an end-to-end virtualized architecture using TrustGraph. We have used Xen [7] and sHype [33] as the hypervisor and the mandatory access control (MAC) module. Xen's access control module (ACM) is in fact an implementation of IBM's sHype and it supports type enforcement and Chinese wall security policies.

In this architecture, TrustGraph runs in the privileged partition (dom0) of Xen and any graphical request by the virtual machines are sent to dom0 via hyper calls (see figure 2). We have modified sHype security labels to carry the MLS labels (i.e. a level and a set of categories.) In our current implementation, the MLS security policy is enforced inside TrustGraph. However, it is also possible for the graphics subsystem to ask the hypervisor's ACM for the access decisions. Note that we do not know whether Xen provides strong isolation and non-interference between the virtual machines or not and these properties are yet to be proven. It is used in our implementation just as a proof of concept.

For the end-to-end implementation, we have used Xen 3.3.1 with XSM and ACM (sHype) features turned on. Fedora 10 is used in dom0 and the virtual machines (dom1) run Fedora 8.

VI. Evaluation and Formal Methods

To evaluate the implementation of TrustGraph, we have performed different levels of testing. First, we have tested the functionality of TrustGraph through a number of applications. Then we have implemented a number of successful attacks on vanilla DirectFB based on the threat model. We show that these or similar attacks are prevented in TrustGraph. For the most critical parts of the implementation, i.e. label flow logic and the window manager ordering logic, we have used formal methods techniques to model check the implementation. Finally, an analysis of the possible covert channels on top of TrustGraph is presented and their capacities are estimated and then reduced using the concept of fuzzy time.

A. Functionality Testing

Testing the functionality of TrustGraph is done by developing a number of native applications. These applications test different modules and functionalities of TrustGraph.

Three applications have been developed to test windowing, input handling, and image/video loading. The windowed application checks the window manager, correct window ordering, and window insertion and re-stacking. The input handling application tests the input grabbing methods and input redirection. Finally, the last application tests the data buffer and the successful loading of image and video modules. The functionality tests have been performed on a Fedora 10 workstation (kernel version 2.6.27.9) with an Nvidia Quadro FX 570M video card. X has been disabled for all the tests.

B. Attack Evaluation

To show the types of attacks possible under a graphics subsystem which are blocked by TrustGraph, we have implemented three sample attacks. Note that these are not ad-hoc attacks; they are designed in a bottom-up approach based on the threat model discussed in section III. In fact, the reader can easily design and implement similar attack under DirectFB or X. The attacks are not indicative of implementation problems of any graphics subsystem. These graphics subsystems, in their vanilla form, were not designed, nor should they be used for trusted systems. On the other hand, the attacks show the necessity of a secure graphics subsystem and how it can block the security policy violations.

Three attacks have been implemented to test TrustGraph. In the first attack, two applications conspire to communicate in the violation of the security policy. One application acquires an interface to the window of another application by enumerating all the windows on the display and retrieving an interface to its layer. It then dumps the surface regularly for the new messages and writes its messages back on the surface. The other application can communicate with the first one by simply reading and writing to its surface. The system has no control over this channel and the two applications can easily communicate.

In the second attack, a window attaches to the event buffer of another window, reads and removes some of the events, and puts a number of false events back to the event buffer. This violates both the confidentiality and the integrity of the other window.

In the third attack, the attacker grabs the specific key events (e.g. the function keys F1-F12 or the escape key) regardless of the focus. Whenever these keys are pressed on the victim window, they are redirected to the attacker. Since the specific operations of the victim application are bound to these key presses, the attacker can infer those operations whenever it receives the key press events.

The attacks are successful under vanilla DirectFB. Similar attacks can be designed and implemented under X. TrustGraph, however, stops these attacks. The first and second attacks are stopped when the first application gets the interface while the third attack is blocked when the attacker grabs the keys.

In fact, while implementing the functionality testing programs, an accidental bug was introduced to the windowed application where a window was trying to get a higher level surface. We observed an abnormal behavior when one of the windows failed to start. Finally, by inspecting the log, we realized that TrustGraph successfully detected and prevented the buggy assignment.

C. Formal Verification

Formal method techniques are used to verify some of the more critical parts of the TrustGraph implementation such as the label flow logic and the window manager ordering logic. We have used ACL2 to describe and model check the correctness of those parts.

1) ACL2: ACL2 (A Computational Logic for Applicative Common Lisp) [23] is an automated reasoning system consisting of a language and a mechanical theorem prover. It is the “industrial strength” successor to the Boyer-Moore theorem prover [22]. Both the ACL2 language and its implementation is built using the side-effect free version of Common Lisp [37].

In common Lisp everything including the code and the data is a list. The lists hold data such as integers, lists, fractions, or characters. For instance, (120) is an integer, (1 2 3 6) is a list of integers, (a) is a character, and (1/6) is a fraction, all represented as lists. The code is also written using lists, usually with the first element representing the operator or function name and the rest of the elements representing the arguments. For instance, (if x y z) is equivalent to “if x then y else z”, or (car (x y z)) represent the first element of (x y z) which is x.

When using ACL2, first the operation or the model is described using this syntax and a number of function definitions (defun.) Then, we describe a number of theorems (defthm) that ACL2 tool tries to prove about that operation or model. The ACL2 theorem prover then tries to prove the theorem using some basic axioms that it has in its libraries and by breaking it into some smaller theorems (subgoals.) Upon successful proof of the theorem, ACL2 outputs the list of rules and axioms it used to prove that theorem. ACL2 theorem prover is sound, but incomplete. As a result, if it proves a theorem, the theorem is always true, but if it fails to prove it, the theorem might be true or false. Detailed description of how ACL2 works or how to check models with ACL2 is beyond the scope of this paper. The readers may refer to the ACL2 book [23] for more information.

2) Formal Verification of TrustGraph: To check the correctness of the logic implemented in the TrustGraph window manager and the label flows, we have performed model checking on them using ACL2. Note that verifying the entire implementation of a graphics subsystem is very difficult if not impossible due to its large size. Hence, we chose to model check the most critical components of TrustGraph upon which the entire security is dependent.

First, we verify the label flow logic. Four main methods are modeled: acquiring window, surface, event buffer, and setting the window level. Some of the ACL2 scripts are provided in appendix A. Each method behaves as described in its ACL2 model. If the acquirer has a declassification label that dominates the resource's permanent label, it is allowed to acquire the resource. When acquired, both labels of the acquirer is propagated to the resource.

To prove the correctness of label flow, we define four ACL2 theorems. The first theorem ensures that no application can acquire a window if it does not have a dominating label. The second theorem proves a similar property for windows and surfaces. The third theorem ensures that no window can attach to a higher security event buffer. Finally, the fourth theorem consider the combined effect; i.e. an application cannot access a higher security event buffer through another window interface. We also model and prove the correctness of the logic of window ordering in the window manager.

Using the model, ACL2 messages, and proof sub goals, we were actually able to find a number of flaws in the initial implementation of TrustGraph. First, a security tag can never be NULL. This bug was found when ACL2 failed to prove the label propagation theorems and the output showed that no assumption can be made about "(CONSP label)". In the ACL2 context, the "CONSP" predicate means that the both elements of a label (the permanent and declassification labels) always exist and are not NULL. To fix this bug, the NULL condition is checked after any creation or acquisition just in case a component fails to set a label as a result of an unpredictable condition.

Also the initial condition of the window manager was not secure. ACL2 messages showed that although adding and restacking functions preserve the correct ordering, there is no guarantee that if we start with a window stack, the ordering is correct initially. Additional checks were put in place to guarantee that when the graphics subsystem starts, the windows are in the correct order.

Finally, another flaw was that there was a type mismatch for some of the security labels. This bug was found when ACL2 failed to prove the label propagation theorems because no assumption could be made about the type of each label (e.g. INTEGERP or CHAR-P predicates of ACL2 corresponding to labels being integer or characters.) This is because if the labels have mismatching types, the comparison cannot be made. For the system to operate correctly, all labels must be type consistent. As an example, if a byte representing the MLS level is unsigned in some portions of the code and signed in other places, it can result in a security breach. A main interface with a signed char level of 120 actually gets access to a resource with an unsigned char level of 251 because when the comparison operator is used, the resource level is interpreted as -5. All security levels in TrustGraph are implemented as unsigned chars. Each security compartments is also an unsigned char and each resource can have a set of up to five different categories.

After correcting the flaws, ACL2 was able to prove the label flow theorems using 29 axioms and rules and by breaking them into 15 sub-goals. The window ordering theorem was also proved by ACL2 using 17 rules and 30 sub-goals. Note that the scripts in the appendix are the corrected versions.

D. Covert Channel Analysis

Covert channel attacks [29] can pose a threat to the security of high-assurance systems by using the shared resources in a way not intended in their design to leak information and violate the security policy. It is often very difficult, if not impossible, to eliminate or even enumerate all possible covert channels in a system. The best current recommendations for dealing with covert channels are specified by the TCSEC [3] and its successor the Common Criteria [5] evaluation schemes. In these schemes it is recommended that first the channel capacities of the possible covert channels are estimated. Then, using some mitigation techniques, the channel capacity must be reduced to an acceptable level (usually 100 bits/sec is considered acceptable.)

We are interested in the covert channel attacks that use the dynamics of the graphics subsystem to communicate information. Such channels use the graphics methods to transfer sensitive data one bit (or a few bits) at a time. To estimate the capacity of a graphics covert channel, we have implemented a prototype covert channel on top of TrustGraph. As mentioned before, the applications can enumerate all the windows on the display. Our covert channel uses this fact to transfer data. At each time slot, if the number of windows on the display is even, it conveys a "0" and if it is odd it conveys a "1". At each time slot, if the sender wants to send a "1" and the number of windows is even, it opens a small dummy window and does not do so otherwise. The dual procedure is done for transferring a "0". We have measured the capacity of our channel by transferring a 10MB file over the channel. The total transfer time was about 10 minutes and 30 seconds which means that the channel capacity is around 127,000 bits/sec. Given that the window opening and closing activity of the system is reasonable (not more that one window per second), the noise level on such a channel is negligible; hence, the channel capacity is equal to the bandwidth. The capacity measured here is well beyond the acceptable rate. Similar channels can be designed using the other resource creation/acquisition methods. As another example, the sender can acquire the window of the receiver at each time slot, releasing its access to the window, thus sending one bit at a time.

To reduce the capacity of such covert channels, we use the idea of fuzzy time[17]. A random delay of maximum 100 ms is imposed on *all* of the resource creation and acquisition methods in TrustGraph. The amount of delay is selected using the Merssene-Twister random number generator[27]. After introducing this delay, we measured the new capacity to be strictly less than 20 bits/sec. Note that the average delay for each method is about 50 ms.

The elegance of this approach is that it does not introduce any delay to the drawing operations. These operations are performed by the application on an already acquired surface, so they cannot leak information across different applications. Hence, the random delay does not slow down the visual effects or the graphics acceleration. For benign applications, it just slows down the window opening by an average of a few tens of milliseconds (depending on how many resources are created) which is masked by the application starting delay and is not even noticeable by the user.

VII. Related Work

There have been a number of efforts in the literature for building trusted MILS and MLS systems using virtualization. IBM's PR/SM [10] and VMM-based security kernel for VAX architecture [21] describe two of the earlier attempts.

NSA's NetTop [28] and MILS [6] and IBM's sHype [33] are three of the more recent VMM-based high assurance systems. Karger [19] studies the requirements of the MLS systems and discusses their implications on the design of VMMs. Terra [16] and Secvisor [35] also build trusted systems through virtualization. Using the fact that VMMs reside in a lower layer than the guest operating system, they provide code attestation and isolation for the virtual machines. Walker et al. [39] use formal techniques to verify the Linux security kernel.

Karger and Safford [20] describe the I/O virtualization complexities and study the performance and security tradeoffs of different I/O models. AMD [4] and Intel [18] support the Input Output Memory Management Unit (IOMMU) approach for assigning I/O devices to virtual machines. However, this approach is not suitable for graphics system which is inherently shared between the virtual machines.

Woodward [42] describes the requirements for a trusted graphics system for Compartmented Mode Workstation (CMW), one of the early attempts to build a high assurance MLS system. Epstein and Picciotto [14] study the security problems of X.

There have been different efforts to build trusted X. Epstein [13], [12], [11] and Woodward [42] describe different trusted X implementations. Picciotto [31] presents two approaches for implementing trusted cut and paste operation in X. Another work by Picciotto and Epstein [32] surveys the architectures and security policies implemented by the trusted X implementations. Finally, the work by Feske and Helmuth [15] is another recent effort on building secure GUI.

A recent attempt for building trusted graphics is done by adding security hooks to X (known as X Access Control Extension or XACE) and extending a two-level trust hierarchy to it [40]. However, simply dividing clients to "trusted" and "untrusted" is too coarse-grained. A more flexible policy model is implemented by extending the SE-Linux policy [36] to X using XACE hooks [41]Nevertheless, this implementation only mediates known channels under X and does not provide any type of assurance [24]. Considering the large size of X and the obsolete features in its code, it is difficult to provide assurance for SE-Linux-enabled X. Moreover, none of the existing trusted graphics subsystems provide capacity reduction for the graphics API covert channels. Finally, Paget [30] describes how design flaws in the Win32 API can be exploited to escalate privilege.

VIII. Conclusion

We have described the design and implementation of TrustGraph, a secure graphics subsystem for high assurance systems. TrustGraph is designed based on a number of fundamental security principles which we adhere to throughout the implementation. Labeled resources, secure methods, secure operations, and a secure window manager provide complete mediation and compliance with the security policy in the graphics subsystem. To adhere to the principle of simplicity, we have used DirectFB instead of X for our implementation which required modest amount of fresh code and modification.

We have performed different levels of evaluation to test TrustGraph. Correct operation is verified through simple functionality tests. Security mechanisms and label checking are verified through a number of simple attacks designed based on the threat model. Finally, detailed formal model checking is done to verify the critical components of the implementation: the label flow and the window manager operations. Throughout the different stages of verification, we have found and corrected a number of flaw in the initial implementation of TrustGraph.

We have also analyzed TrustGraph for the covert channels through the graphics API, estimated the capacity of such channels, and reduced their capacity using the idea of fuzzy time.

TrustGraph can be used as the graphics subsystem in machines with either a single trusted operating system or multiple virtual machines running on top of a VMM. We plan to release TrustGraph to be used as a secure graphics subsystem in high assurance systems.

References

[1] Directfb. `http://www.directfb.org/`.

[2] Mozilla directfb porting. `https://wiki.mozilla.org/Mobile/DFBPorting`.

[3] Trusted computer system evaluation criteria. National Security Institute, 1985. 5200.28-STD.

[4] Amd i/o virtualization technology (iommu) specification. Advanced Micro Devices, 2006. publication no. 34434.

[5] Common criteria security assurance requirements. Common Criteria Recognition Arrangement, 2007. CCPART3V3.

[6] Jim Alves-Foss, Carol Taylor, and Paul Oman. A multi-layered approach to security in high assurance systems. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90302.2, Washington, DC, USA, 2004. IEEE Computer Society.

[7] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[8] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., 1973.

[9] Biba. Integrity considerations for secure computer systems. *MITRE Co., technical report ESD-TR 76-372*, 1977.

[10] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Syst. J.*, 28(1):104–123, 1989.

[11] J. Epstein. A high-performance hardware-based high-assurance trusted windowing system. In *NISSC'96: National Information Systems Security Conference*, 1996.

[12] J. Epstein. Fifteen years after tx: A look back at high assurance multi-level secure windowing. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 301–320, Washington, DC, USA, 2006. IEEE Computer Society.

[13] J. Epstein, J. McHugh, R. Pascale, H. Orman, G. Benson, C. Martin, A. Marmor-Squires, B. Danner, and M. Branstad. A prototype b3 trusted x window system. *Computer Security Applications Conference, 1991. Proceedings., Seventh Annual*, pages 44–55, Dec 1991.

[14] J. Epstein and J. Picciotto. Trusting x: Issues in building trusted x window systems -or- what's not trusted about x? In *Proceedings of the 14th Annual National Computer Security Conference*, October 1991.

[15] N. Feske and C. Helmuth. A nitpicker's guide to a minimal-complexity secure gui. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 193–206, New York, NY, USA, 2003. ACM Press.

[17] W.-M. Hu. Reducing timing channels with fuzzy time. *Research in Security and Privacy, 1991. Proceedings of the 1991 IEEE Computer Society Symposium on*, pages 8–20, 1991.

[18] J. Humphreys and T. Grieser. Mainstreaming server virtualization: The intel approach. IDC, 2006.

[19] Paul A. Karger. Multi-level security requirements for hypervisors. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 267–275, Washington, DC, USA, 2005. IEEE Computer Society.

[20] Paul A. Karger and David R. Safford. I/o for virtual machine monitors: Security and performance issues. *IEEE Security and Privacy*, 6(5):16–23, 2008.

[21] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the vax vmm security kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.

[22] M. Kaufmann and R. S. Boyer. The boyer-moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[23] M. Kaufmann and J. S. Moore. *ACL2*, volume Version 3.4. University of Texas at Austin, August 2008.

[24] D. Kilpatrick, W. Salamon, and C. Vance. Securing the x window system with selinux. Technical report, National Security Agency, March 2003.

[25] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007.

[26] M. Manely. The x window system must die. Featured Articles, July 2000. Linux.com.

[27] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(3), 1998.

[28] R. Meushaw and D. Simard. Nettop: Commercial technology in high assurance applications. *National Security Agency Tech Trend Notes*, 9(4):3–10, Fall 2000.

[29] J. Millen. 20 years of covert channel modeling and analysis. *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 113–114, 1999.

[30] C. Paget. Exploiting design flaws in the win32 api for privilege escalation, August 2002. `web.archive.org`.

[31] J. Picciotto. Towards trusted cut and paste in the x window system. *Computer Security Applications Conference, 1991. Proceedings., Seventh Annual*, pages 34–43, Dec 1991.

[32] J. Picciotto and J. Epstein. A comparison of trusted x security policies, architectures, and interoperability. *Computer Security Applications Conference, 1992. Proceedings., Eighth Annual*, pages 142–152, Nov-4 Dec 1992.

[33] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J.L. Griffin, and L. Doorn. Building a mac-based security architecture for the xen open-source hypervisor. *Computer Security Applications Conference, 21st Annual*, pages 285–295, 2005.

[34] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Trans. Graph.*, 5(2):79–109, April 1986.

[35] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, pages 335–350, 2007.

[36] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. Technical report, National Securoty Agency, May 2002.

[37] Guy Steele. *Common LISP : The Language (LISP Series)*. Digital Press, June 1984.

[38] R. Scott Thompson. *Quartz 2D Graphics for Mac OS X(R) Developers*. Addison-Wesley Professional, 2006.

[39] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the ucla unix security kernel. *Commun. ACM*, 23(2):118–131, February 1980.

[40] E. Walsh. X access control extension specification, 2006. X.org Foundation.

[41] E. Walsh. Application of the flask architecture to the x window system server. In *Proceedings of the 2007 SELinux Symposium*, 2007.

[42] J. P. L. Woodward. Security requirements for system high and compartmented mode workstations. Technical Report MTR 9992, MITRE Corporation, November 1987.

APPENDIX

Some of the ACL2 models and theorems are shown in table 1. The complete scripts are not presented for the interest of space, but are similar to those shown in table 1.

Table I
TRUSTGRAPH LABEL FLOW SCRIPTS.

```
(defun get_win (win int_PL)
   (if (dom int_PL (car win))
       (list int_PL int_PL)   nil))
```

```
(defun dom (resa resb)
   (if (AND (>= (car resa) (car resb))
            (subset (cdr resb) (cdr resa)) ) T nil))
```

```
(defun set_win_label (win int_PL label)
   (if (dom int_PL label) (list (car win) label) win))
```

```
(defthm no_leak  (implies
       (AND (consp win) (NOT (dom int_PL (car win))))
       (= (get_win (set_win_label win int_PL label)
             int_PL) nil)))
```

```
(defthm no_leak2
    (implies
      (AND (consp win) (consp surface)
         (NOT (dom
           (car (cdr (set_win_label win int_PL label)))
           (car surface))))
      (= (get_surface surface
         (set_win_label win int_PL label)) nil)))
```