TRUSTED AND HIGH ASSURANCE SYSTEMS

BY

HAMED OKHRAVI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor David M. Nicol
Professor William H. Sanders
Associate Professor Vikram Adve
Professor Nitin H. Vaidya

Abstract

High assurance MILS (multiple independent levels of security) and MLS (multilevel security) 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. This work provides a study of commercial-off-the-shelf (COTS) products to support high assurance MLS systems and designs a candidate architecture based on virtualization and trusted execution to provide strong compartmentalization. We then identify three major security problems in the candidate architecture: the lack of trust in the network, the problem of patch management, and untrusted graphics. We study and solve each of the security gaps in detail. More specifically, we design and evaluate a trusted network architecture for high assurance applications, evaluate an optimal pre-deployment testing time for effective patch management, and finally design, implement, and formally evaluate a trusted graphics subsystem.

To my parents

Acknowledgments

I would like to express the deepest appreciation to my adviser, Professor David M. Nicol, who has always supported me with his excellent guidance, insight, and patience. He has provided me with an excellent atmosphere to do research. Also, I would like to thank Professor William H. Sanders, whose great advice and ideas guided my research for the past several years.

I would like to thank my committee members, Dr. Vikram Adve and Professor Nitin H. Vaidya, for their support and brilliant suggestions that helped me improve my research. Without their guidance this dissertation would not have been possible.

I would also like to thank my parents who have always inspired me and whose love and guidance are with me in whatever I pursue. They are the ultimate role models.

Finally, I would like to thank my loving fiancé, Asal. She has endured this process with me, always offering endless inspiration, support, and love.

Table of Contents

List of Figures

List of Tables

1 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 such systems. 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 and/or Biba 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.

In this work, we first investigate different components available from the commercial and research communities which can be used to build MLS systems. The study of the available commercial-off-the-shelf (COTS) products, the security functionalities they provide, their workings, and readiness status suggest a candidate architecture based on trusted execution, virtualization of the environment, secure IO, and the TPM chip functionalities. In this architecture, MLS security policy can be implemented by the VMM or the separation kernel. Trusted networks and MLS-enabled peripherals (e.g. storage) offer security outside the host boundaries. We then identify a number of major security gaps in the candidate architecture: the lack of trust in the network, the problem of patch/update testing and management, and the problem of I/O and especially the lack trusted graphics.

Although some approaches have been proposed to extend security labeling to the network and make it security aware [1, 2], they do not extend "trust" to the network. Simple network traffic labeling has its shortcomings. Namely, a secured host gets the same network label as an under-patched host with an old anti-virus version, running an outdated version of a software if they have the same MLS level. Moreover, the problem of rogue hosts and

users, unsecured physical access, vulnerable devices, and security device misconfiguration (e.g. firewall rule conflicts) are not addressed in an untrusted labeled network. We propose a network architecture based on new technologies collectively referred to as "trusted networks." Specifically, we study trusted networks in the context of critical infrastructure, one of the most important applications of high assurance systems. Trusted networks allow user and device authentication, configuration validation, automated patch enforcement, device access control, and posture remediation. We study the architecture and requirements of a trusted network; especially for high assurance and high availability applications. An algorithm is proposed to distribute firewall rules across many network access devices in a trusted network in order to avoid introducing new rule conflicts to the network. The architecture is then evaluated against known attack patterns.

An inherent problem with automated patch enforcement in trusted networks is pre-deployment testing, i.e., how much one tests a patch/update before enforcing it in the system. If the pre-deployment testing phase is long, the patch is less likely to introduce a new vulnerability (or break the system), while the system has a longer window of vulnerability, and vice versa. We evaluate the pre-deployment testing period using analytical and simulation models and find an optimal testing time which results in the minimum number of open vulnerabilities at any time.

Finally, a traditional issue with virtualization is the problem of I/O. Different approaches have been proposed for virtualizing I/O [3]. 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 VMs. Although this model can be practical for network or storage devices, it does not work for keyboard, mouse, or graphics. The reason is that it is difficult or impractical for a workstation to have multiple copies of these devices. The second approach is to have one copy of each I/O device with all the I/O drivers in the VMM. The VMM then presents a virtual device to each VM. In this case, VMs share each I/O device and the driver that controls it. KVM [4] 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 (privileged partition) on top of the VMM which controls

I/O operations. Any request for 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 [5].

In the latter two approaches of I/O virtualization, the subsystem that controls an I/O device is shared between 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. 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, along with the fact that it handles data from different security levels, imposes a high risk of information leakage and interference. In fact, applications frequently use the graphics subsystem resources as a means of communication that is not regulated by the security policy [6].

We describe the design and implementation of TrustGraph, a trusted graphics subsystem for high assurance systems. In TrustGraph, entities (resources) in the graphics subsystem are labeled with appropriate security tags to prevent unauthorized communication. Moreover, methods and operations are secured so that they comply with the security policy. TrustGraph can be used in a single secure operating system or in a virtualized architecture. TrustGraph is implemented using DirectFB [7], a thin graphics library with integrated windowing system. The implementation is evaluated against attacks and information leakages that are possible under X or vanilla DirectFB, but are prevented in TrustGraph. In addition, 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 and corrected them. In order to narrow the gap between the model of the system that is formally verified and the actual C implementation, compiler-based techniques are used to perform static analysis of the implementation of TrustGraph. The static analysis checks the implementation for security violations or bad coding choices which may result in a breach of control flow integrity (CFI) of the code and deviation between the model and the implementation. Moreover, we provide an analysis of possible covert channel attacks that use the graphics API as the means of communication, we measure the channel capacity of

those covert channels, and reduced the channel capacity using the idea of fuzzy time.

The rest of the dissertation is organized as follows. Chapter 2 studies the COTS products that can be used to build high assurance multilevel security workstations, the services they provide, and their limitations. A candidate architecture for high assurance MLS systems based on virtualization and the security gaps in such an architecture are described in Chapter 3. Chapter 4 studies trusted networks in the context of process control systems while the problem of patch management and pre-deployment testing is evaluated in Chapter 5. Chapter 6 details the design, implementation, and evaluation of TrustGraph. We describe the design of a simplified version of TrustGraph and its formal evaluation in Chapter 7 before concluding the work in Chapter 8.

2 COTS Products and Available Technologies

2.1 Introduction

A multilevel system architecture implements an MLS security policy. For such architecture to be secure, there must be no path to bypass or disable the security mechanism. This goal can be achieved only when every component of the system is trusted or can be verified by a trusted component. Consequently, a secure MLS system cannot be built with secure memory and untrusted I/O or with secure end devices and vulnerable network architecture. It is important to remember that a system is at most as secure as its weakest component.

In this chapter, we discuss a comprehensive MLS system architecture which is built using trusted components that support MLS functionality. Whenever a device is not trusted or can be tampered with, it must be verified by a trusted component and must be tamper evident. Any unsecured component can lead to data leakage or security policy violation.

Figure 2.1 shows various components of an MLS system. Each component is discussed in a separate section. In each section, we study the mechanisms to secure that component, different COTS devices which support MLS functionality, their workings, and the extra features they may provide.

2.1.1 Contributions

The contributions of this chapter are as follows:

- Different COTS components and devices are studied.
- The functionalities, workings, and features of each device are studied.

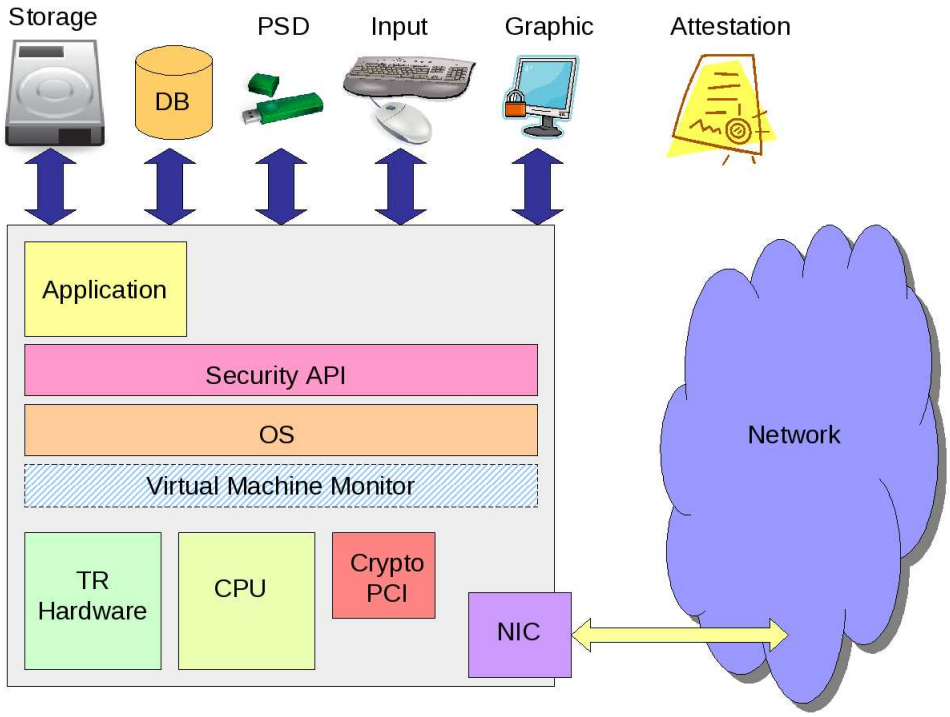

Figure 2.1: MLS sample architecture

- The assurance level (or certification) and the readiness of the components are investigated.

The chapter's goal is to widely study the available technologies and the services that they can provide, and it serves as the introduction to the subsequent chapters.

2.2 Tamper Resistant Hardware

A secure MLS system requires support from a tamper resistant hardware. Since it is sometimes necessary to verify the integrity of even the lowest layers of software including the OS kernel and drivers, these operations cannot be done in software. Furthermore, there are secret values in a secure system (such as the root secrets) which have to be irremovable from the platform in order to prove the identity of the system. However, software cannot keep the root secrets permanently. Warm resets or system crashes may result in the loss of root keys. Consequently, these operations and storage of platform secrets must be done in hardware. Two major commercial attempts offer such functionalities.

2.2.1 IBM 4758 Secure Coprocessor

The IBM 4758 secure coprocessor [8] has been one of the major commercial attempts to implement a tamper resistant hardware. It supports high-speed cryptographic functions for data encryption or signing. It has a set of embedded certificates which are used for platform attestation applications. It implements PKCS#11 and the IBM Common Cryptographic Architecture standards which support DES and T-DES encryption, single and double key message authentication codes (MACs), SHA-1 and MD5 hash functions, and symmetric or asymmetric key distribution algorithms.

The coprocessor can be used for large volume data encryption, data, software, and OS integrity verification using MACs, trust establishment, and attestation.

The IBM 4758 is tamper resistant. It senses hardware tamper attempts using a randomly placed grid of wires inside its casing. It responds to tampering by zeroing its secrets and changing its state. Penetration, temperature extremes, voltage variation, and radiation are examples of events which trigger tamper alarm.

The coprocessor consists of different layers of software. The first layer is loaded into the on-board ROM, boots first, and loads the next layer. Other layers reside in a battery-backed RAM and each is loaded by the previous layer.

The 4758's hardware (RAM, ROM, Flash...) and its firmware (post, miniboot) can only be modified or upgraded by IBM; its software (loaders, kernel, OS, and applications), on the other hand, can be upgraded by the officer of each layer. The officer of each layer has a unique signing key and is appointed by the previous layer by signing its public key.

The boot sequence starts from the ROM; ROM loads the OS and it, in turn, runs the applications. To enforce the access control in this strict fashion, IBM 4758 has a number of ratchet locks. These locks are strictly increasing counters which are advanced whenever a layer is loaded completely and is ready to load the next layer. Access decisions are made based on the counter value.

The coprocessor also has a PRNG unit which generates all the random numbers needed. Note that the coprocessor generates all the seeds to these PRNGs internally. IBM 4758 also has a feature called "targeting": commands from the officers can target the coprocessor units

with specific configurations. The targeting feature facilitates operations and maintenance of the device. The operating system running on the coprocessor is a modified version of CP/Q which includes device drivers for different hardware units of 4758. The 4758 also comes with a C API which is suitable for application development.

IBM 4758 is the world's first product to be certified at NIST FIPS 140-1 Level 4. The new versions of IBM 4758 are produced under the new "IBM 4764" name.

2.2.2 Trusted Platform Module (TPM)

TPM [9] is another widely used commercial product which presents a tamper resistant hardware for trusted systems. It is used for applications such as protected storage, data integrity, trusted execution, secure booting, sealing, attestation, and trust establishment. TPM is currently certified with Common Criteria at EAL 3.

The architecture has three roots-of-trust and some trusted building blocks (TBBs) built around them. Roots of trust include the root of trust for measurement (RTM) used for integrity management and platform attestation, the root of trust for storage (RTS) which manages secure storage of data and keys outside the primary trusted boundary, and the root of trust for reporting (RTR) which is used for secure reporting and monitoring.

Integrity measurement in TPM is done using a layered approach in which each layer verifies the next one by computing hashes and securely storing the reference values. These values are stored iteratively in platform configuration registers (PCRs). TPM also has five types of credentials which are used for authentication and attestation purposes: endorsement (EK), conformance, platform, validation, and identity (AIK) credentials.

Along with normal encryption and signing, TPM supports sealing and sealed-signing which essentially means that the remote platform must have specific values in its PCRs in order to decrypt the data. This enables TPM to apply strong policies for data access.

The API which supports the TPM and fills the gap between the software and the chip is called the TCG Software Stack (TSS). TSS provides a layered stack of APIs which interact with the higher layer and provide an interface to the lower layer. At the top of the stack is the application which requires TPM support. The layer beneath the application is called

the TCG Service Provider (TSP) which provides functions used by the application to TPM operations. Under TSP, there are three other layers called TSS Core Services (TCS), TCG Device Driver Library (TDDL), and the TPM Device Driver each facilitating TPM services for its upper layer.

TSS can be used to develop TPM-based applications at different software layers. For instance, user applications use TSP as an API to interface with TPM. Kernel applications, on the other hand, can use TCS or TDDL APIs. Lower layer kernel codes can directly interface with the device driver.

For the sake of perspicuity, the usages of IBM 4758 and TPM in MLS system are not described here. They are presented throughout the future sections depending on the component deploying them.

2.3 Processor

Modern CPUs provide security functionalities which facilitate the implementation of MLS policies. In this section, we study four major technologies in new CPUs which can be deployed to build an MLS system.

Intel Trusted Execution Technology (TXT or LaGrande)

Intel Trusted Execution Technology [10, 11, 12] is a new feature on Intel CPUs which provides a trusted execution environment for secure applications. It requires support from the I/O subsystems, the OS kernel, and the TPM chip, and it provides functionalities such as data and program protection, attestation, and secure I/O.

Intel TXT Capabilities

Intel TXT delivers the following on-chip capabilities to build a trusted environment:

- Protected Execution: Enables each secure application to run in its own secure and isolated environment. It prevents malicious applications or malware from observing

and/or interfering with the execution of a secure application. Each application receives its dedicated share of resources.

- Sealed Storage: Provides information isolation and data security. Sealed storage essentially means encrypting the policy along with data. Using this feature, a piece of data is encrypted along with the environment properties (e.g. hash value of the kernel or value of a set of registers). Data can be decrypted only in the same environment. It prevents an attacker from tampering with devices, OS, or memory to steal encryption keys and decrypt data.

- Protected Input: Provides a secure path from input devices (e.g. mouse and keyboard) to the application. When using this feature, the application domain's input manager shares a key with the input device. Any data (e.g. keystrokes) is encrypted with this key before it is communicated to the machine. Only applications who own the key can decrypt the input data. It prevents attacks such as key logging.

- Protected Graphics: Provides a secure path between the application and the output display. Display information is encrypted before the application sends it to the display context (e.g. window object). It prevents malicious applications from observing or interfering with display data.

- Attestation: Provides assurance about the applications and the trusted execution environment to other parties. Using this feature, another party can get a signed list of processes running in the system, the hashed values of some specific applications (e.g. drivers), or environment measurements (e.g. hash of the kernel or register values). Using attestation, different parties can establish mutual trust and can get assurance that the other party is authentic or has a secure environment.

- Protected Launch: Provides controlled launch of OS and system software components (e.g. OS kernel, kernel modules, etc.).

To implement the above functionalities, Intel TXT uses processor, chipset, mouse/keyboard, graphic, and TPM features. The processor enhances event handling and provides instruc-

tions for working with protected execution environments and for a more secure software stack. The chipset provides memory protection, enhancement to protected data access from memory, secure IO and graphics, and TPM interface. Keyboard, mouse, and graphic subsystem must support secure channels with the application. Finally, the TPM chip provides encryption, sealing, and attestation services.

Intel TXT Workings

Intel chips that support TXT technology have two modes of operation: standard and protected. In a standard partition, applications run normally as they do in an IA-32 environment. This preserves the backward compatibility for old applications and those which do not require security.

A protected partition, on the other hand, provides isolation and non-interference services for the applications running inside it. An application running in a protected partition is allocated its own memory pages encrypted with keys known only to the application. Unauthorized applications cannot read or write to these pages. The chip also blocks any DMA request to the protected memory pages. Moreover, information paths to and from the applications in a protected partition are secured against sniffing or modification by encrypting data from input devices (e.g. mouse/keyboard) and to output graphic display. The software code that provides isolation between different partitions is called the domain manager.

Secure applications can run entirely inside a protected partition. On the other hand, they may have standard sections that run normally in a standard partition and critical, secure sections that run in the protected partition. This facilitates adding secure functionality to existing applications and decreases protection overhead.

When an application needs to run in a protected partition, it executes a SENTER instruction. Some pages of memory are allocated to this application and they are protected from being read or written by other applications. The processor loads an authenticated code module into this memory, stores its hash into PCR registers in the TPM, and invokes it. The authenticated code checks for proper hardware configuration, enables memory protection for the domain manager, authenticates the domain manager, and invokes it.

To exit protected execution, the domain manager first sanitizes the protected partition from any sensitive data. It then executes a SEXIT instruction to disable protected execution and free up the protected memory. Intel TXT allows exceptions and interrupts only within the protected partition in order to prevent these events from leaking sensitive data. The processor also watches for unexpected events such as sudden resets and wipes the protected memory in those cases before it can be accessed by any other application.

Intel TXT application in MLS

To implement information segregation using Intel TXT, each application with a specific security level must be executed in a separate domain. This isolates the applications and prevents information leakage or malicious observation of state.

However, isolating applications in this manner prevents any flow of information between different security levels. MLS and information flow security policies define a set of rules that allow some form of data exchange between different domains and block other information flow. To implement such a policy using Intel TXT, there are two possibilities. One is to use shared domains. In this case, secure applications run under the same shared domain. A domain manager (e.g. the OS kernel) enforces the security policy here. In this model, although secure applications are well protected against other applications, they can observe or manipulate each other's state (memory content). Furthermore, this model does not use TXT functionalities to implement information flow security; rather, it is the domain manager which implements the policy.

Another approach is to virtualize the execution environment. In this model, each virtual machine is executed in a separate domain. A virtual machine monitor (VMM) manages these virtual machines and implements the security policy. This model can potentially provide better segregation since the applications run in different domains and no direct interference is allowed between them except through the VMM. Because VMMs are usually smaller than OS kernels, it is easier to make sure that they are secure or to verify them formally.

Intel TXT Extra Features

Intel TXT architecture also delivers attestation and trust establishment services using the TPM chip. To do this, Intel TXT uses the permanent keys embedded in the TPM chip to handshake with a remote agent in order to establish trust. As the TPM chip is bound to the hardware (attached to the motherboard for example), this proves the true identity of the system.

To provide attestation, TPM stores system measurements (e.g. hash values of specific software, driver, or kernel modules) in its PCR registers. To prove the authenticity of the system, TPM sends a signed version of these registers to remote parties.

2.3.1 Intel Virtualization Technology (IVT)

Virtualization [13] is another feature that can be deployed to implement secure MLS- enabled systems. It is supported in most of the new Intel processors as well as many of the older ones.

IVT application in MLS

Recall that virtualization can be deployed along with trusted execution to implement application segregation and MLS policy. More details about this model are discussed in Intel TXT section. A detailed discussion of the IVT is beyond the scope of this work. Here, we briefly point out some of its features that are specifically useful for secure applications.

- DMA: IVT supports complete DMA remapping. This makes the VMM much lighter weight since it does not have to deal with tracking DMAs in the software. Lightweight VMMs are generally easier to verify and are considered more secure. Note that if virtualization is used the MLS policy is implemented by the VMM, so it is important for the VMM to be (provably) secure.

- Interrupt: IVT also remaps interrupt tables completely. As in the case of DMA, interrupt remapping helps to reduce the size of the VMM.

- Multilevel page table: Adds an extra step in address translation process. It remaps the virtual memory for the virtual machines. Multilevel page table also facilitates virtualization and removes the burden of memory tracking from the VMM.

2.3.2 AMD Secure Virtual Machine (SVM or Pacifica) and Trusted Execution

SVM

AMD SVM [14] also provides virtualization features. It facilitates DMA and interrupt handling by intercepting and remapping them. Furthermore, SVM improves the virtual machine performance by offering state switching mechanisms. It saves the state (register values and pointers) of a host OS and loads the state of the guest OS when switching from host to guest and vice versa.

In addition, SVM provides an additional virtual address translation step, called nested page table, along with a tagged TLB to further simplify the VMM.

Trusted Execution

Some of the AMD processors provide TPM and trusted execution support. AMD secure startup offers attestation and memory protection services which are similar to those offered by Intel TXT. AMD security features, however, do not offer memory encryption or secure IO.

The processor starts establishing a protected environment when a SKINIT instruction is issued. Secure loader (same as Intels domain manager) initializes the SVM hardware, authenticates an application, and invokes it. To provide memory protection, the processor blocks any other application from accessing the pages allocated to this domain. The memory, in contrast, is not encrypted.

TPM keeps the application and environment measures as in the case of Intel TXT and can provide attestation of the environment to other parties.

In addition, secure startup clears protected memory pages on sudden exits. Called Au-

tomatic Memory Clear (AMC), this processor unit clears memory content in case of a warm reset or a system exception. It prevents information leakage by imposing intentional system exceptions. Intel TXT has the same feature.

2.3.3 Intel vPro Technology

Intel vPro [15] is a recent technology which enhances security and manageability of computers. It allows remote entities to connect to the processor directly when the machine is shut down, hibernated, or even when the OS is absent or has crashed. This facilitates asset discovery, inventory checks, automated/remote updating, diagnosis and repair, and patch management. Intel vPro also provide security services which are described below.

Intel IVT and TXT

Intel vPro technology has trusted execution and virtualization technologies built in. It offers all IVT features such as DMA and interrupt remapping and virtual memory. It supports general purpose virtualization which essentially has a guest OS running on top of a host OS.

In addition, there is another mode of operation named special purpose virtualization or virtual appliance. In this mode, a software stack on top of the VMM virtually produces an appliance and its services. For example, a virtual appliance can offer intrusion detection/prevention service. In this model, the virtual appliance runs on top of the VMM as a piece of software and is connected to the virtual network interface card of the guest OS. Any traffic to the guest OS is passed through the virtual appliance by the VMM. The virtual appliance can inspect traffic to the guest OS and block it in case it detects any anomaly in the packets.

Intel vPro also supports complete TXT functionality. The mobile version of the vPro technology, Centrino Pro, however, does not yet support TXT.

Support for Cisco NAC

Intel vPro supports IEEE 802.1x and Cisco NAC technology for network security. It can be used to send security posture information about the host to posture validation servers and to keep credentials in the hardware. More details about the Cisco NAC technology are explained in the network section.

Intel vPros application is MLS

Intel vPro technology can be used to support an MLS system in different ways. Its Cisco NAC and 802.1x compliance can be used to build trusted networks which are necessary for communicating tagged information, preserving tags, and enforcing network MLS policy. Moreover, TXT and IVT technologies can be deployed, as described earlier, to build MLS systems.

A more important usage of vPro technology in MLS systems can be the virtual appliance concept. A virtual appliance can have arbitrary control over the traffic to and from a virtual machine. This makes virtual appliance a good candidate for implementing separation kernel. A virtual appliance can implement arbitrary information flow security policy for the virtual machines and provide information segregation. As the control over the communication and virtual appliance comes directly from the processor, this provides a safer solution than implementing the separation kernel on the same host as an application or inside the OS kernel. It also keeps the VMM lightweight and provides a separation of duty between the VMM and separation kernel.

2.4 Operating System

Another important component of an MLS system is the operating system (OS). An MLS enabled OS launches secure applications, provides memory protection, and implements an MLS security policy. It fills the gap between a secure application and a trusted hardware. Technologies such as Intel TXT's secure launch require some form of OS support. In this section, we study major operating systems with MLS functionality and explain their features.

2.4.1 SE-Linux

Security Enhanced Linux [16] supports a variety of mandatory access control (MAC) security policies to restrict accesses and control the flow of information. It implements Type Enforcement (TE), Role-based Access Control (RBAC), and MLS security policies. To do that, SE-Linux adopts a MAC framework called FLASK.

The TE policy is one kind of policy in the SE-Linux policy language. In this policy, a type is associated with each subject (often called the "domain" of the subject) and each subject can only access the objects of the same type.

Role-based access control is another type of policy implemented in the SE-Linux. In an RBAC system, there should be two different mappings: the first is the mapping between users and roles, and the second is the mapping between roles and permissions. In the SE-Linux RBAC, the first mapping is defined using the "user" construct, but the second mapping uses the TE as its basis. This generates a mapping between roles and types, and access control rules are defined only based on types.

Another type of MAC policy supported by SE-Linux is MLS policy. SE-Linux defines a "security level" for each subject (a user, process, or application) and each object (a file or process) in the system. It implements the Bell-LaPadula (BLP) security policy for confidentiality which blocks any writing to a lower and any reading from a higher security level (no-read-up and no-write-down rules). Currently, it does not support Biba security policy for integrity which blocks any writing to a higher and any reading from a lower security level (no-read-down and no-write-up rules). It is recommended that type enforcement is used, instead, to provide integrity. More sophisticated information flow policies can also be supported by a combination of SE-Linux TE and MLS.

A "security level" in SE-Linux is defined as a sensitivity (e.g. top secret, secret, confidential, and unclassified) and a set of compartments (e.g. a subset of Asia, Europe, Africa, America). For instance, the security level of a very sensitive document about Asian and European affairs can be (Top secret, Asia, Europe). A security level dominates another level if it has a higher sensitivity and a superset of its compartments.

SE-Linux confidentiality rule allows only reading from a dominated object and writing

to a dominant one, but not vice versa. MLS policy in SE-Linux applies to every activity in the operating system (even copy-pasting from different windows).

Security levels in SE-Linux are stored in object or subject tags. They are ASCII characters describing the security level of the element. Security levels can only be modified through specific, restricted commands.

SE-Linux, however, does not provide network object labeling because it cannot preserve the security labels across the network. This has to be worked around using other mechanisms which are described in the network section. SE-Linux is evaluated at Common Criteria EAL 4+.

2.4.2 Trusted Solaris

Trusted Solaris [17, 18] also supports a limited version of MLS policy. It supports BLP confidentiality policy and network labeling. It, however, does not provide integrity or arbitrary information flow security policy in general.

To implement MLS, Trusted Solaris defines different "zones." Each process and file system object is assigned a zone. A zone defines a security label and is isolated from any other zones. A process is only allowed to read the files from a lower zone and write to files from a higher zone.

Nevertheless, there is a difficulty in assigning a zone to each process or file. Some processes may need to access multiple zones and some files may need to be available to processes from various zones. For example, access to the /tmp directory is needed by different processes. If a single zone is assigned to it, some processes may not be able to access it. To solve this problem, Trusted Solaris virtualizes the entire application environment into different instances and assigns a label to each of them. This is called poly-instantiation.

Trusted Solaris also provides network labeling functionality to preserve object security labels across the network. It is evaluated at Common Criteria EAL 4+.

2.4.3 Trusted BSD, SEBSD, and Trusted IRIX

Two versions of BSD provide MLS functionality. Trusted BSD [19] includes a module, named MAC Framework, which can enforce arbitrary MAC policies. MAC Framework has different components such as the policy registration, composition, labeling, and system calls. Policies are implemented as modules. Examples of policies which can be implemented using Trusted BSD are Biba, MLS, TE, BLP, etc.

Trusted BSD labels OS elements such as process, file system, network, and IPC objects in order to enforce the MAC policy. It can further compose multiple policies if there is more than one available in the system. Consequently, it can support complicated information flow security policies. Trusted BSD provides APIs for label-aware applications to work with the OS labels. Trusted BSD conforms to different Common Criteria protection profiles, though it has not been evaluated for any of them yet.

SEBSD [19] is the port of the FLASK Type Enforcement framework to the BSD OS. Hence, its working is very similar to that of SE-Linux. In SEBSD and SE-Linux, Linux Security Module (LSM) functions similar to the MAC Framework in Trusted BSD. Although versions of SEBSD are available, it is still under development.

Trusted IRIX [20] is yet another flavor of Linux which supports MLS policies. Developed by the Silicon Graphics, Inc. (SGI), IRIX is a UNIX-like operating system for SGI clusters. Trusted IRIX implements MLS policy by labeling files and processes and restricting accesses by mediating access requests as in Trusted BSD. In addition, it offers network labeling using standard protocols and an extra session manager to enforce additional session level policy. Trusted IRIX 6.5 has been validated to conform to the National Security Agency Information Systems Security Organisation's (ISSO) Labeled Security Protection Profile (LSPP) and the ISSO's Controlled Access Protection Profile (CAPP).

2.4.4 Trusted Linux

Another version of MLS-enabled Linux is called Trusted Linux [21]. It is sometimes described as Linux plus TPM. It provides client integrity by deploying loadable kernel modules: TPM, Extended Verification Module (EVM), and Simple Linux Integrity Module (SLIM). It stores

the measurements of data files and executables in the TPM at the first open or execution and verifies their integrity on each access.

Trusted Linux supports MLS confidentiality (also referred to as secrecy) and integrity policies. It has a fixed set of pre-defined labels for files, namely, "sensitive," "user," "public," and "exempt" for confidentiality and "system," "user," "untrusted," and "exempt" for integrity. Moreover, each process has four security classes:

- Integrity Read Access Class (IRAC)
- Integrity Write/Execute Access Class (IWXAC)
- Secrecy Write Access Class (SWAC)
- Secrecy Read/Execute Access Class (SRXAC)

Trusted Linux enforces the following rules to enforce confidentiality and integrity. These rules are slightly more general versions of BLP and Biba security policies. If IRAC=IWXAC and SWAC=SRXAC, the rules reduce to Biba and BLP.

Read:

```
IRAC(process) <= IAC(object)
SRXAC(process) >= SAC(object)
```

Write:

```
IWXAC(process) >= IAC(object)
SWAC(process) <= SAC(object)
```

Execute:

```
IWXAC(process) <= IAC(object)
SRXAC(process) >= SAC(object)
```

Trusted Linux, nonetheless, does not support more flexible security policies or arbitrary types and labels. Trusted Linux is pending evaluation by the Common Criteria.

2.4.5 Microsoft Next-Generation Secure Computing Base (NGSCB or Palladium)

NGSCB [22] is the kernel designed by Microsoft which drives the Intel TXT. It uses TPM as the platform for its trusted operations and integrity checks. NGSCB provides Secure Startup as a means to ensure boot integrity, protect data in an offline system, and increase the efficiency of TPM deployment.

NGSCB's Secure Star protects the system against offline attacks by encrypting data using keys derived from TPM root secret. Furthermore, it prevents OP exploitation and software hacks by verifying the integrity of the kernel and master boot record (MBR) through measurement values stored in TPM. It also supports key escrow for key recovery and debugging purposes.

In contrast, NGSCB does not provide any mechanism to protect against hardware tampering, insider attack, post login attack, BIOS reflashing, poor maintenance, and network or online attacks.

It protects applications and data integrity by deploying Intel TXT's protected memory and trusted execution functionality. It launches secure applications in protected areas of memory, called "curtained memory," not accessible to any other application even the OS itself.

NGSCB was first proposed to be implemented in Windows Longhorn and later on Windows Vista. The majority of the features, however, are expected to be released well after Windows Vista. The only NGSCB feature already available in Windows Vista is BitLocker which encrypts a whole volume of the file system using keys derived from the TPM secrets.

2.5 Network

Network support is crucial for a secure MLS system. A secure network must prevent the leakage of sensitive information communicated between two processes on two machines. This requires that the network preserves the security labels of data and different machines on the network have the same perception of the security labels. Moreover, a secure network

must provide security services such as confidentiality, origin integrity, and data integrity. In general, it is more difficult to provide security across the network than on a single host. Various open protocols such as TCP/IP used across a typical network were not designed with security concerns in mind; hence, they are usually vulnerable to a wide variety of attacks and exploits. In this section, network technologies supporting MLS systems are discussed. The solutions are divided into two major sections: untrusted technologies and trusted ones.

2.5.1 Untrusted MLS Network Technologies

There are untrusted network technologies in use which can support MLS systems and provide some form of security against network-based attacks. Although they are secure against some classes of attacks, these technologies are "untrusted" in the sense that they cannot establish the *true* identity of each party bound to its platform. For example, IPSec provides confidentiality and integrity services for messages communicated between two parties, but it does not establish the true identity of either.

IPv4, Commercial IP Security Option (CIPSO), and Revised IP Security Option (RIPSO)

The traditional solution for preserving data labels across the network uses IPv4 and encodes label information in the IP header. In this case, the "option" field in the IP header is used to carry the label for the information in the payload. Each security level is encoded by a byte value and placed in predefined locations in the IP options field.

Two Requests for Comments (RFCs) standardize the usage of IP header to carry label data. Commercial IP Security Option (CIPSO) specified in RFC 1108 standardizes such usage and describes the bit combinations and their location in the IP header.

A revised version of CIPSO is described in the Revised IP Security Option (RIPSO) RFC 1038. It provides more specific details of the standard label values.

Trusted Solaris, Trusted BSD, and Trusted IRIX support CIPSO/RIPSO for preserving labels across the network. SE-Linux does not have such a capability; however, extensions to SE-Linux which can support these standards are possible.

CIPSO/RIPSO does not encrypt label data, so the implicit assumption is that no one can modify the packet as it traverses the network. IPSec relaxes this condition.

IPSec

A more secure solution for encoding security labels across the network is using IPSec. In this model, the security label is still placed in the IP options field, but it is protected by the IPSec AH and ESP headers. AH ensures that the label is not changed across the network (integrity) and ESP provides label confidentiality. A security association (SA) is assigned to each security label, i.e., each combination of sensitivity plus compartment.

On the outbound, the security label is inserted in the IP options field; an appropriate SA is selected based, among other things, on the label; and an IPSec packet is generated and sent across the network.

On the inbound, the packet is received, its IPSec header is extracted, it is processed using the appropriate SA, the label is extracted, and finally the MLS security policy is enforced based on the label value.

Trusted Solaris and Trusted BSD support network labeling using IPSec. Extensions of SE-Linux can support labeled network over IPSec.

2.5.2 Trusted Network Technologies

Trusted networks [23, 24, 25, 26, 27] extend the concept of trust to the network architecture. Using existing standards, protocols, and network appliances, they address security concerns such as rogue devices, rogue users, non-complaint devices, configuration errors, unsecured physical access, and bypassing of the security mechanism. The complete description of a trusted network working and the details involved is well beyond the scope of this work. Nevertheless, we provide a brief explanation of them and describe how they can augment MLS architectures with their security services.

The trusted network architecture is proposed by the Trusted Computing Group's (TCG) Trusted Network Connect (TNC). Commercial implementations of such networks are available from Cisco TrustSec [24], Cisco CleanAccess [25] (also known previously as Cisco Net-

work Admission Control or NAC [26]), and Microsoft Network Access Protection (NAP) [27]. The commercial trusted networks are even interoperable. For instance, Cisco NAC and Microsoft NAP can work side by side in a network.

Trusted networks provide strong authentication, security posture validation (e.g. patch, OS, and antivirus verification), automatic remediation of noncompliant machines, directory service, auditing, and port-based network access control. They also offer the security services inherent in the protocols they use. For instance, confidentiality and integrity services can be provided by using IPSec. Trusted networks are a move away from the traditional perimeter network security model. They provide network security based on trust and from within the network.

Whenever a client device needs to join the network, the network access device (NAD) authenticates the device using EAP over 802.1x protocol [28]. The NAD sends the device credentials to the Authentication, Authorization, and Access control (AAA) server using RADIUS protocol. The AAA server refers the client device to a set of posture validation servers (PVSes). PVSes check the compliance of the device with security requirements (e.g. having a properly patched OS, updated antivirus, and/or authentic version of software). They send the results to the AAA server using HCAP protocol. Based on the compliance results, the AAA sends an appropriate network security policy to the NAD for enforcement. The NAD enforces the policy for any traffic from the device's physical port. Inter-machine communication is secured using IPSec over UDP or IPSec over IPv6 which can take place only after the above authentication and trust establishment steps. Trusted networks can use TPM for secure authentication and posture validation. Figure 2.2 shows different components of a trusted network. There are various details involved in the workings of these protocols and the trusted network which are omitted for the sake of space.

Trusted MLS Network

Trusted networks can implement an MLS security policy in a secure and resilient manner. Traditional solutions can still be supported by placing security labels in IPSec packets. A trusted network, nonetheless, can directly implement an MLS policy to achieve a secure MLS

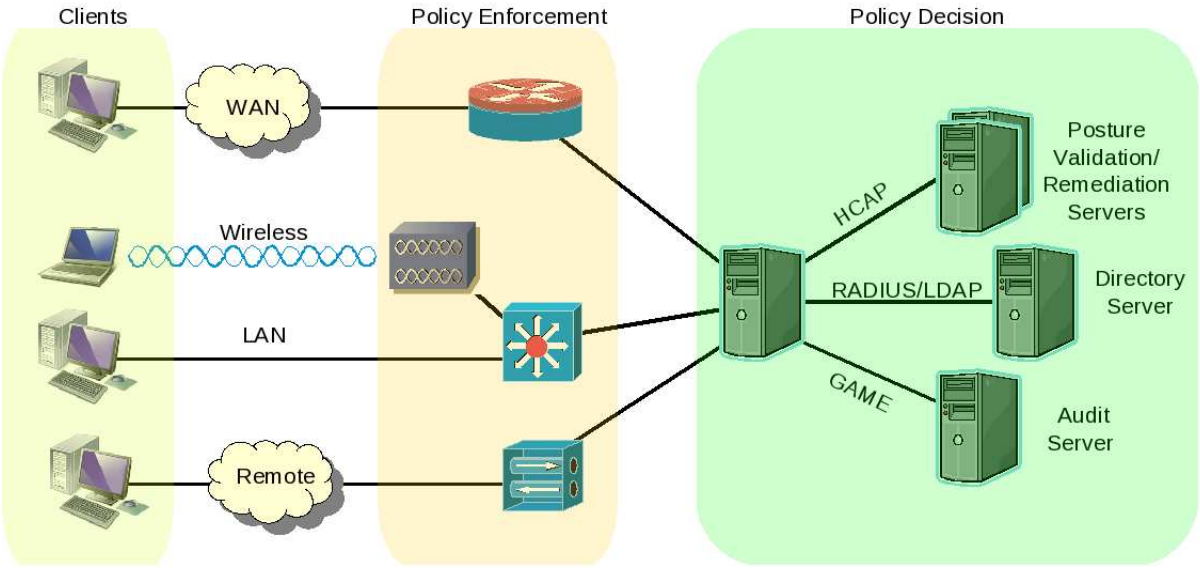

Figure 2.2: Trusted network as an alternative to traditional perimeter security

network in a more wildcard manner.

The directory server inside the trusted network holds the role or attributes of each client in its database. These values are used by the AAA server to decide what traffic policy the NAD enforces for the traffic from the client. The directory can store the security level of each user in the network. It sends back this information to the AAA server during the posture validation process using the LDAP protocol. Based on the user's security level, the AAA server can then send an MLS policy to the NAD to be enforced on that user's traffic. For example, a user with "secret" security clearance must be denied access to a "top secret" database machine attached to the network.

Since the security level of each entity is determined based on its true identity established during the authentication phase, it is very difficult, if not impossible, for untrusted users or devices to get around the security mechanism. Security label tampering or false labels are prevented using the wildcard enforcement of the policy by the NAD. Note that in this model, we do not rely on the OS to preserve or provide security labels. The traffic from the client is tagged by the NAD itself. This remedies OS hacks that cause false labeling or bypassing the security engine.

Other Features of Trusted Networks

Besides implementing an MLS policy, trusted networks offer other beneficial security services. They prevent configuration errors by compliance verification, mitigate malware attacks by automatic update/patch/antivirus management, and block untrusted devices and users using strong hardware based (TPM) authentication. Moreover, they prevent bypassing of the security mechanism by applying the MLS policy to the physical port of the client device.

2.6 Database

There are commercial database systems available which have built-in MLS support. In this section we briefly explain two major ones.

2.6.1 IBM DB2

IBM DB2 [29] supports MLS policies by labeling objects and subjects in the system. In DB2, subjects refer to users, tasks, batch jobs, and UNIX daemons. Objects refer to data sets, a row in a DB2 table, commands, terminals, printers, Direct Access Storage Device (DASD) volumes, and tapes.

A security label in IBM DB2 is composed of a security level and a set of categories. Recall the definition of these terms from previous sections. As in the case of SE-Linux, there is a strict order defined on the security labels (domination notation). It implements BLP MLS policy; i.e. it denies read-ups and write-downs. "Read" and "write" are generic terms used here for similar privileges. DB2 standard privileges are "alter," "delete," "select," "insert," "update," etc.

DB2 also supports wildcard access control rules for finer grain information flow control. This is done through DB2s "GRANT" command. Furthermore, DB2 implements MLS TCP/IP policy rules which make it even more flexible in expressing flow control policies. Permissions can be defined based on IP addresses, protocol, and security labels.

IBM DB2 has been evaluated at Common Criteria EAL 4.

2.6.2 Oracle Label Security and Virtual Private Database (VPD)

Oracle databases [30] implement MLS using label based access control (LBAC). As in other MLS systems, a label has a level and a set of compartments. It has privileges similar to those defined for DB2 and enforces MLS confidentiality policy for accesses.

Virtual Private Database (VPD) is another technology implemented in Oracle databases which facilitates fine-grained access control for fields of a table inside the database. It defines a shadow of a table (using a SELECT-like command) and defines user privileges to that shadow. For example, it can be used to block user accesses to a column containing Social Security numbers in an employee information table, yet allow access to other columns.

Oracle databases are evaluated to conform to Database Management System Protection Profile at Common Criteria EAL3.

2.7 Storage and Personal Storage Drives (PSDs)

The IBM Tivoli Framework [31] is a major system and storage management platform that provides MLS support. It defines security level and category as LDAP object attributes. More details about IBM Tivoli can be found in its manual [31]. It has been evaluated at Common Criteria EAL 3.

Another research effort [32] proposes a high assurance MLS file server model which can implement both BLP and GWVr2 policies. Since it uses formal methods for designing the file system, it can meet Common Criteria EAL 7. It has an MLS message router (MMR) which handles access requests to the file system. The MLSFS mediates all access requests to preserve confidentiality, uses MMR to enforce information flow policy, and enforces a modified version of the BLP security policy. In this policy, writing is strictly limited to one's own level instead of higher levels in the original BLP policy.

Trusted Computing Group (TCG) also extends its trust model to storage devices [33]. The TCG model provides three main security benefits: (1) it establishes trust between storage devices and hosts through mutual authentication and trust establishment. (2) It provides secure control over storage security features which support storage protection and encryption

for storage devices. (3) Finally, it creates a secure communication channel between hosts and storage devices. TCGs trusted storage offers fine-grained storage security, improved storage access control, efficient device-level encryption, and automated backup functionalities. TCG also makes recommendations for lifecycle management of such storage. MLS labels can be encoded in the trusted storage access control mechanism.

3 Candidate Architecture

3.1 Introduction

In this chapter, we design and study a candidate architecture for high assurance, multilevel security workstations using available and customized technologies. The reasoning behind some of the choices is explained, but the architecture is not unique. There may be other designs for a high assurance MLS system such as a process-based system or application virtualization systems.

3.1.1 Contributions

The main contributions of this chapter are as follows:

- A candidate architecture based on virtualization for high assurance MLS is designed and studied.
- We study Intel TXT technology in depth—especially, how it can be deployed in a trusted system.
- A trusted boot is designed and implemented to measure the hypervisor and virtualization environment.
- Different security gaps in the candidate architecture are identified and explained. The gaps identified in this study provide the foundation for the subsequent chapters, each of which solves one of the identified gaps.

3.2 Candidate Architecture

In a multilevel security system, each entity has a security level. These levels can be the traditional "top secret," "secret," "confidential," and "unclassified" levels or a more sophisticated set of levels and compartments. Accesses to resources are restricted using these levels. For instance, "read-ups" and "write-downs" are denied for confidentiality reasons. An MLS policy can be a simple Bell-LaPadulla policy or a more sophisticated information flow policy.

3.2.1 Different Approaches

There are two different approaches for building a multilevel security computer system: process-level and machine-level architectures.

Process Level MLS

The first approach to build an MLS system is to have different processes with different security levels inside a machine. In this model, the operating system enforces the security policy (separation policy) when an access to an object (e.g. file) is requested by a process. Examples of this model are SE-Linux and Trusted Solaris operating systems. Figure 3.1 shows this model. The benefit of using this model is that the operating systems provide many facilities and flexible policy frameworks to implement MLS. In this model, however, it is hard to achieve assurance requirements. SE-Linux, for instance, protects against unauthorized accesses, but it does not ensure that the isolation is never breached.

To achieve such isolation assurance, the Linux Security Module (LSM) must be modified to support TPM. TPM can be used to isolate process spaces (compartmented attestation) [34]. This is, however, difficult to achieve and the operating system must be modified intrusively.

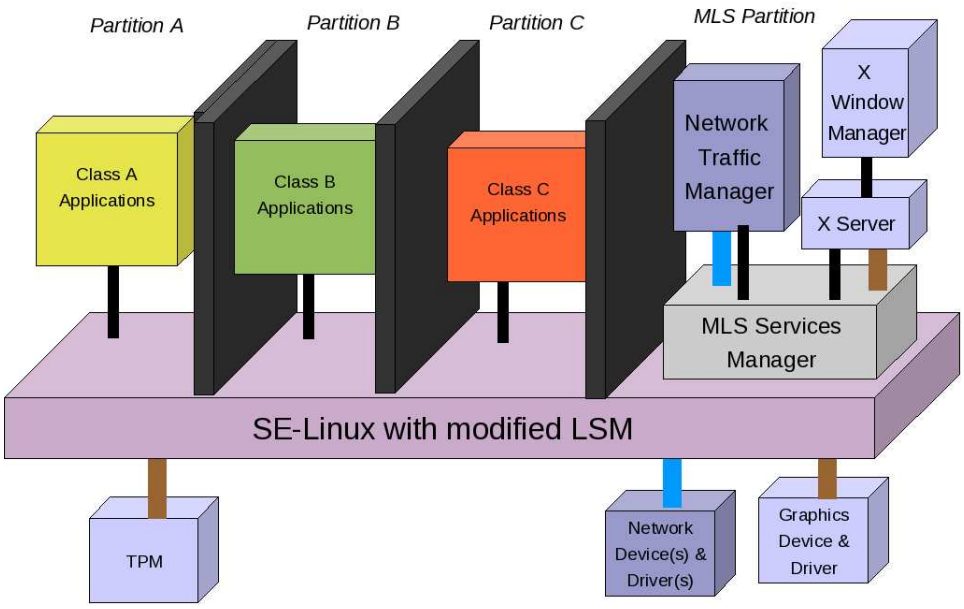

Figure 3.1: Process-level MLS architecture

Machine Level MLS

The other approach to build an MLS system is to have different machines with different security levels inside virtual machines. In this model, each machine is assigned a single security level. The Virtual Machine Monitor (VMM or Separation Kernel) enforces the isolation policy. It also labels the network traffic of each virtual machine with its appropriate level. All of the processes inside a virtual machine have the same security level. Note that in this model the policy is coarser grained because it restricts interactions between virtual machines instead of processes. Figure 3.2 depicts this architecture.

On the other hand, it is much easier to achieve strong isolation in this model. Intel Trusted Execution Technology (TXT) uses this model along with TPM support to isolate VMs. We focus on this approach for its potential to achieve high assurance.

3.2.2 Trusted Execution Technology

The candidate architecture uses Intel's TXT [10, 11, 12] to achieve isolation. In this model, different VMs are launched on top of a VMM. TXT measures the VMM to ensure that it is not tampered with. It also protects the memory of each VM, isolating it from other VMs,

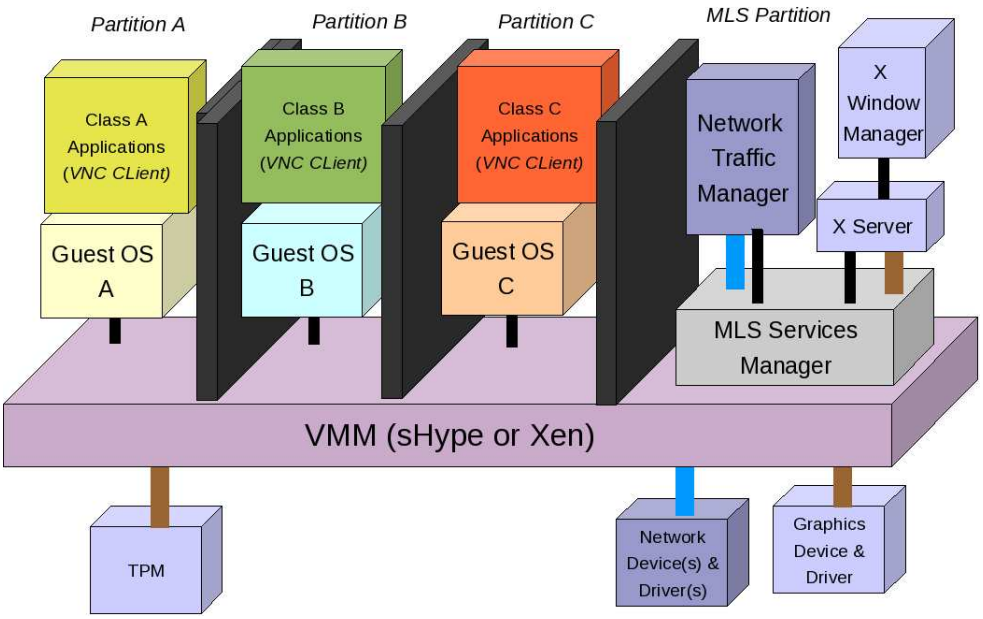

Figure 3.2: Machine-level MLS architecture

seals the sensitive data, protects input and output devices, and provides attestation.

Attacker Model of TXT

Intel's TXT protects against a wide range of attacks:

- Memory Manipulation: TXT prevents memory manipulation attacks by isolating VMs and encrypting their memory spaces. This means that each VM is launched within its own memory space, inaccessible to other VMs. TXT also controls DMA accesses to prevent DMA-based memory manipulation.

- Input Manipulation: TXT secures any input channel to the VMs. Input devices such as mouse and keyboard encrypt data using a session key before it is sent to the application. This prevents attacks such as changing the keyboard driver or keystroke logging.

- Output Manipulation: Output channels are also protected by the TXT. Display output is encrypted before sending to the graphic card to prevent attacks that change the display driver. Screen captures and phishing are also blocked in this fashion.

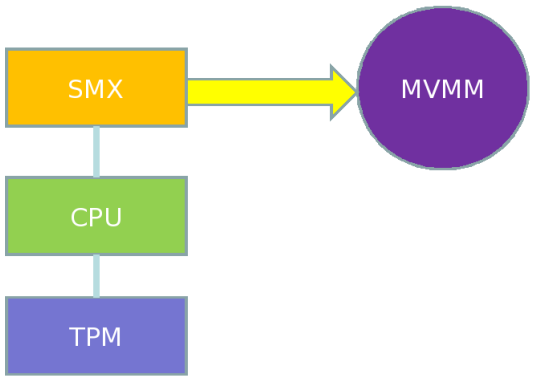

Figure 3.3: Measuring VMM using SMX

3.2.3 Candidate Architecture Using TXT

To protect against the above mentioned attacks and to guarantee isolation, the candidate architecture uses the TXT's *chain-of-trust* model.

Chain of Trust

First, the CPU chip (hereafter referred to as "chip") measures the VMM (separation kernel) to ensure that it is not tampered with. If the VMM passes this stage, it will be called a *Measured Virtual Machine Monitor* or *MVMM*. A measurement is a cryptographic hash of the VMM. The result of the measurement (hash value) is stored inside the TPM. Before each launch, the measurement value is compared with this stored value inside the TPM.

If the VMM passes the measurement (MVMM), the control is passed to it. The MVMM "mediates" all VM activities after this point and essentially implements the policy.

The chip uses Safer Mode Extension (SMX) instruction to measure the VMM and pass control to it (as shown in Figure 3.3). Note that at the beginning, there is a guest OS running on top of the CPU. After running the SMX instruction, the VMM takes the control and virtualizes the environment, including the starter guest OS.

To make sure that the VMM has never been tampered with during its lifetime, measurements are done incrementally. Incremental measurements essentially keep a history of all changes to the VMM inside the TPM. In TPM, these measurements are stored in *Platform Configuration Registers (PCRs.)* Hence, a PCR is computed as follows:

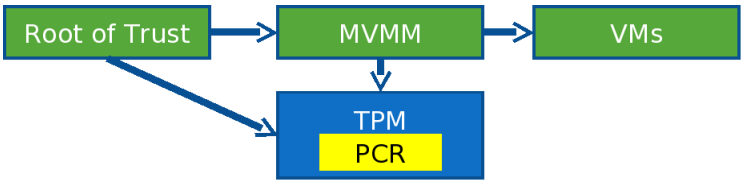

Figure 3.4: A chain of trust using TPM

$$PCR = SHA - 1(OldPCRvalue, NewValue) \tag{3.1}$$

Consequently, any deviation from the desired measurement in the history of the VMM is reflected in its current value. Xen has recently been supported as an MVMM. It can use SMX instruction to initiate a secure launch of the VMM.

After the launch of the MVMM, it starts to measure the VMs using the Virtual Machine Extensions (VMX) instructions. These instructions, in turn, measure the VMs, store the measurements in the TPM, and run the VMs. In essence, TXT establishes a chain-of-trust in which three steps are done repeatedly:

1. Measure the next entity
2. Store the measurement in TPM
3. Pass control to the next entity

This chain-of-trust has a *root-of-trust* which is the master key stored inside the TPM in the factory. Figure 3.4 shows a chain of trusted established by the TPM.

Sensitive pieces of data, in this model, are *sealed* with the desired PCR values. As a result, if the environment (PCR values) changes, the data cannot be accessed.

Memory Protection

Another important requirement of the candidate MLS architecture is memory protection. Lack of memory protection is why operating system based isolation cannot achieve high assurance. TXT provides page protection and memory encryption to isolate VMs and ensure I/O protection. In TXT, there are four types of memory access:

1. MVMM page access: TXT blocks any direct access to the memory from the VMs. Any page access must be requested from the MVMM. Furthermore, MVMM prevents VMs from accessing each other's memory spaces. Each VM is confined to its encrypted pages.

2. DMA: Devices can access the memory through DMA. Although devices are hardware, many of them are configurable through software. As a result, an attacker can misconfigure a device to access restricted memory areas. TXT solves this problem by defining a *NoDMA* table. MVMM controls the NoDMA table to make sure that sensitive memory areas are not accessible through DMA.

3. System Management Interrupts (SMI) transfer mode (STM): STM prevents accesses to prohibited memory regions by normal SMI handling code. It blocks attacks which modify the SMI handling routines to manipulate memory. STM is a peer to MVMM and the policy (i.e. the location of sensitive regions) is negotiated between them.

4. Trusted Graphics Translation Table (TGTT): Display adapters use DMA to efficiently access physical memory pages holding graphical data (frame buffers). To provide assurance that the display output is actually what the applications have sent, TXT uses TGTT. TGTT provides frame buffer protection by dynamically translating between the buffer address and physical memory address. MVMM establishes the TGTT, assigns memory pages to it, and configures the display adapter to use it. TGTT is protected by NoDMA.

Figure 3.5 depicts memory protection in Intel TXT.

I/O Protection

The candidate architecture uses TXT to protect input/output channels as well. Recall that I/O manipulation is part of the attack model.

TXT offers I/O protection through *trusted channels*. Trusted channels are established between the I/O devices and the applications (VMs) by the TPM. Note that TPM does

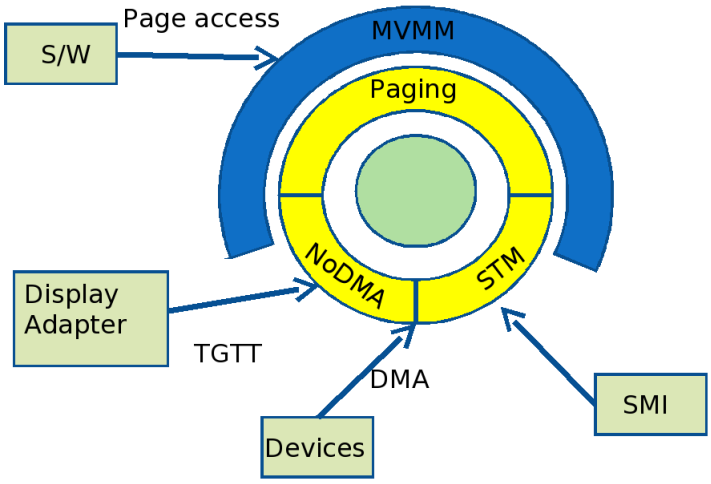

Figure 3.5: Memory protection in Intel TXT

not perform the encryption; It just established the channel at the initial phase, while the applications and I/O devices are responsible for the encryption. There are two types of trusted channels:

1. Cryptographic trusted channels: Usually established between an application and a discrete device, they rely on encryption for the protection of data.

2. Hardware trusted channels: Mostly used for integrated devices, they rely on the difficulty of hardware based attacks for protection.

The I/O devices must be able to support trusted channels for TXT to protect I/O. For instance, mouse and keyboard must encrypt input events (e.g. keystrokes or wheel rotation) before sending them to the system. The receiver application can decrypt these events using session keys established by TPM. Other than this, the workings of these devices are relatively straightforward. The protected graphics, however, are a little more complicated.

Protected Graphics

Graphics protection relies on hardware trusted channels for integrated graphics, and on cryptographic channels for discrete ones. In a regular system, physical memory for graphic display is allocated dynamically. The Graphics Translation Table (GTT) provides mapping

to the physical memory for the graphics engine (such as the X Server). The graphics display writes to the display buffer through GTT and the MUX outputs the data to the display.

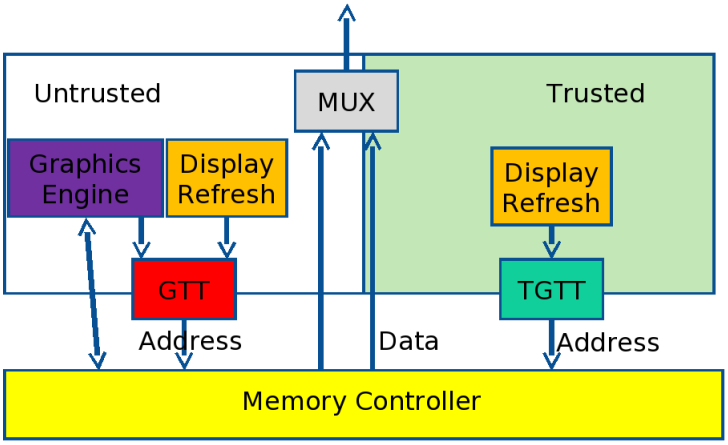

Figure 3.6: Protected graphics model in Intel TXT

In the trusted architecture, the graphic mapping for trusted applications is kept in the *Trusted Graphics Translation Table* (TGTT). This table can only be written by the MVMM. In this model, the applications write their graphical data to their *virtual frame buffers*. The secured driver of the MVMM writes to the actual frame buffer through TGTT (Figure 3.6). This extra level of translation assures that the display output actually comes from the application and that no application can manipulate other applications' data.

The protected graphics builds a trusted surface overlaying the display to show the trusted graphical outputs. This surface is transparent at the regions where untrusted programs are outputting. The MVMM makes sure that this surface has the highest Z value, meaning that it is on top of all other windows. Changes to the entire display (such as resolution or depth changes) result in the trusted surface being torn down and reestablished.

3.3 Trusted Boot

We have implemented a complete trusted boot scheme using TXT and TPM. In this scheme, we verify the authenticity of a VMM (Xen) using trusted boot by measuring (hashing) its binary. If the measurement yields the same value as the value stored in the TPM PCRs, the system loads into the VMM. The VMM then can measure the VMs. Proving the authenticity

of the VMM does not mean that it has desired security properties (e.g., it provides strong isolation.) It can only detect any unauthorized modification to the VMM code or any extra module or rootkit residing in its binary.

3.3.1 Measuring VMM

A small piece of code called the "Authenticated Code" (AC) [35], with a known signature value stored in the mainboard chipset, measures the VMM and sets the NoDMA tables. If the measurement matches the good value stored in TPM NV memory, it passes the control to the VMM. Depending on the chipset model, there are two versions of AC available: X35 and Q35. Our system runs the Q35 version which is a binary code about 25 kB in size stored in the /boot directory of Linux. AC is added as a module to the bootloader and is loaded by BIOS during the boot process before the VMM.

Another piece of code called Tboot [36] performs the verified launch of the VMM (Xen in our case). Tboot is a pre-kernel module that itself is verified by TXT using the Launch Control Policy (LCP). A set of policies called Verified Launch (VL) verify Xen and initrd.

In order to interface with the TPM, we need an implementation of the TCG Software Stack (TSS) API. We use an existing open source TSS from IBM called "TrouSerS" [37]. Tboot uses TrouSers to work with the TPM.

3.3.2 Implementation

To measure the VMM, first we have modified the bootloader to load Tboot before Xen. This is done by modifying Grub configurations as shown in Table 3.1.

We then create the LCP and VL policies as illustrated in Table 3.2. The functions "lcp_mlehash," "lcp_crtpol," and "tb_polgen" are provided by Tboot for creating measurements and policies. Note that the command line is also included in the hash value to ensure that the attacker cannot bypass the booting process by modifying the grub configurations.

Finally, the LCP and VL policies are loaded to the TPM. To do so, we first load a kernel module (TPM driver) and take ownership of the TPM. Then we load the policies into the

Table 3.1: Modifications to grub configuration for trusted boot

```
Original Grub Configuration:
       title Xen 3.2
        root (hd0,1)
        kernel /boot/xen-3.2.gz
        module /boot/vmlinuz-2.6.21.fc8xen root=LABEL=/ ro rhgb
        module /boot/initrd-2.6.21.fc8xen.img
```

```
Modified Grub Configuration:
       title Trusted Xen 3.2
        root (hd0,1)
        kernel /boot/tboot.gz
        module /boot/xen-3.2.gz vtd=1
        module /boot/vmlinuz-2.6.21.fc8xen root=LABEL=/ ro rhgb
        module /boot/initrd-2.6.21.fc8xen.img
        module /boot/Q35_SINIT_16.BIN
```

Table 3.2: Creating LCP and VL policies

```
Launch Control Policy (LCP):
    lcptools/lcp_mlehash /boot/tboot.gz > good_hash
    lcptools/lcp_crtpol -t hashonly -m good_hash -o lcp.pol
```

```
Verified Launch (VL) Policy:
    tb_polgen/tb_polgen --create --policy_type nonfatal --uuid vmm
    --hash_type hash --file vl.pol --cmdline ‘‘module /boot/xen-3.2.gz vtd=1’’
    /boot/xen-3.2.gz

    tb_polgen/tb_polgen --create --uuid dom0 --hash_type hash --file vl.pol
    --cmdline ‘‘module /boot/vmlinuz-2.6.21.fc8xen root=LABEL=/ ro rhgb’’
    /boot/vmlinuz-2.6.21.fc8xen  /boot/initrd-2.6.21.fc8xen.img
```

TPM using Tboot “lcp_writepol” function (Table 3.3).

In order to collect the boot time messages, we have connected a computer to the trusted boot computer using the serial link. Figures 3.7 and 3.8 illustrate the boot time messages for when the verification succeeds and when it fails, respectively. If the measurement does not match the policy, the secure environment is torn down and the system halts.

3.4 Security Gaps in the Candidate Architecture

In this section, we identify major security gaps present in the candidate architecture. Although the usage of TXT and MLS-aware peripherals protects against a wide range of attacks, these gaps still exist in the system. In a highly secure environment, these gaps must

Table 3.3: Loading policies to the TPM

```
Initializing TPM:
    modprobe tpm_tis
    Tcsd
    tpm_takeownership password
```

```
Loading Policies to TPM:
    lcptools/lcp_writepol -i owner -f lcp.pol -p TPM-password
    lcptools/lcp_writepol -i 0x20000001 -f vl.pol -p TPM-password
```

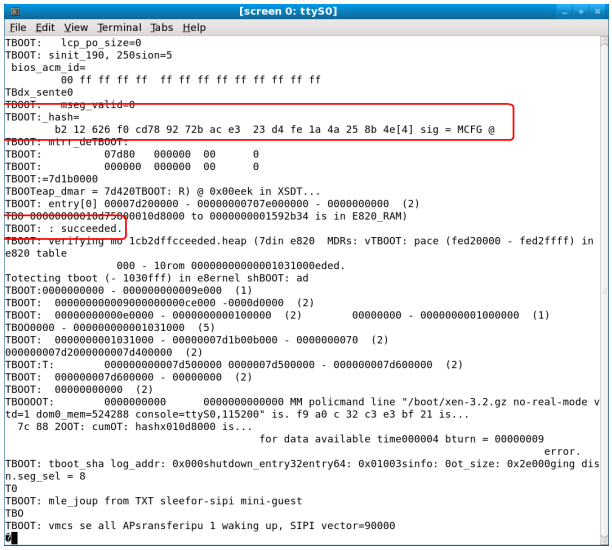

Figure 3.7: VMM measurement success

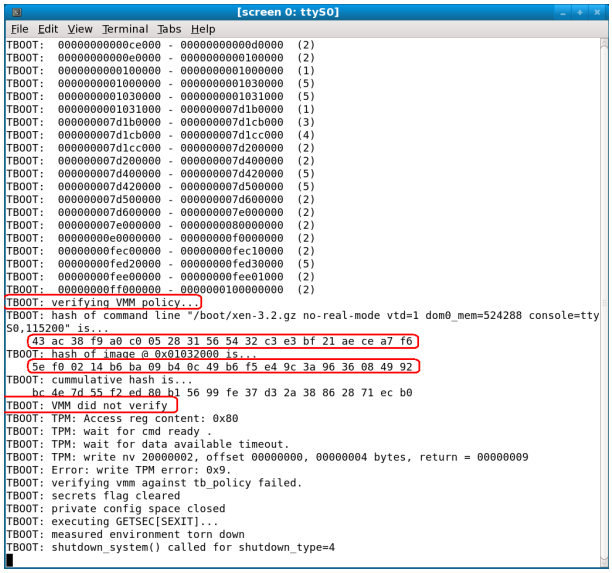

Figure 3.8: VMM measurement failure

be addressed by appropriate mechanisms.

3.4.1 Trusted Network

Another missing piece in the candidate architecture is the lack of "trust" in the network. Simple network traffic labeling has its shortcomings. Namely, a secured host gets the same network label as an under-patched host with an old anti-virus version, running an outdated version of a software if they have the same MLS level.

Furthermore, it is possible to connect mobile devices (e.g. laptops) to the open Ethernet ports or wireless access points, essentially bypassing some of the protection mechanisms. In this case, even if MVMMs behave as expected and VMs' traffics are labeled properly, a

malicious device can potentially ignore these mechanisms by directly connecting to unsecured network access points.

To prevent such attacks, a host's compliance must be verified before it can connect to the network. This means that a machine must be "trusted" before it can send or receive network traffic.

Trusted networks can be used to fill in the gap of trust in the network. They provide user authentication, comprehensive network device admission control, end-device health check (compliance checks), policy based access control and traffic filtration, automatic remediation, and auditing.

A trusted network has the following components:

- Client Machines: optionally have Trusted Platform Module (TPM)
- Network Access Device (NAD): switches, routers, VPN concentrator, or wireless access points
- Authentication, Authorization, and Access Control server (AAA): holds the policy
- Posture Validation Servers (PVS): anti-virus server, patch server, configuration management server, or software verification server
- Posture Remediation Servers (PRS): remedy non-compliant clients
- Directory Server (DS): contains the user account and roles
- Audit Server

Figure 2.2 shows different components of a trusted network. A complete description of the workings of a trusted network is beyond the scope of this work. Here we just review how an MLS trusted network can be constructed. In such a network, the directory server keeps user security levels. Each user's level is determined during the authentication phase. The AAA server holds the MLS policy to be enforced on client's traffic. NADs tag the traffic and enforce the MLS policy fetched from the AAA server. New Intel vPro processors support trusted networks. Chapter 4 studies trusted networks in detail and describes their architecture and requirements in the context of process control systems.

3.4.2 Patch Management

Trusted networks can automatically enforce patch requirements to the machines joining the network, but this raises the major question of how much a patch/update must be tested before it is applied to the system. Since posture validation servers can strictly require a new patch before admission to the network, pre-deployment testing becomes more crucial. If the testing period is long, it is less likely to introduce new vulnerabilities or disrupt the availability of the system while the system has a longer window of vulnerability and vice versa.

Pre-deployment testing is considered as an "always good" practice in the literature and it is usually done in an ad-hoc fashion. Considering the rates of vulnerability discovery and patch development, however, it is possible to find an optimal testing period that corresponds to the minimum number of open vulnerabilities at any time. To the best of our knowledge, this has not been studied in the related work. Chapter 5 studies the optimal pre-deployment testing period.

3.4.3 High Assurance Graphics

Protected graphics in the candidate architecture which uses TXT, ensures that the display actually shows the graphical outputs of the applications. It also prevents applications from accessing the display buffers of each other.

However, this model does not provide "non-interference" guarantees for the graphical data. The graphics engine of the MVMM handles different VMs' data. Graphics engines (such as X Window system) often have many shared data structures available to all applications. As a result, it is possible for an application to *interfere* with other applications' graphical output. Even with TXT graphics protection, applications can potentially read or modify other applications' data.

Note that the malicious applications in this model do not violate the buffer protection offered by TXT. Instead, they misuse the underlying graphics engine to steal data. Note also that measuring the MVMM could not prevent these interferences because the MVMM's graphics engine is the original version and is not tampered with.

In Chapter 6, we explain the security threats of an untrusted graphics subsystem in detail. Then we describe the design and implementation of TrustGraph, a trusted graphics subsystem.

3.5 COTS vs. Non-COTS

Since both COTS and non-COTS technologies have been used in the candidate architecture, it is important to make this distinction for each component.

The high assurance VMM used in the candidate architecture must be custom designed. Although we have implemented a proof of concept system on top of Xen, commercial hypervisors (including Xen) are very large and complex. Hence, it is very difficult to prove strict isolation properties for COTS VMMs.

The trusted execution and trusted boot components of the candidate architecture are available in COTS form under Intel TXT. Moreover, the measurement of the environment and hardware supports are provided by TPM which is also COTS.

Many of the trusted network components (NADs, AAA servers, clients, PVS, and PRS) are available off-the-shelf from different vendors. The trusted network chapter (Chapter 4) discusses special customizations and algorithms to deploy these components in a trusted and high availability system. However, complete support of trusted network functionalities on commodity devices may require software or hardware updates. In some cases, less intelligent devices may require special drivers to connect to a trusted network. The details of the trusted network requirements are described in the next chapter.

TrustGraph is implemented using the DirectFB API which is available off-the-shelf. To implement TrustGraph, the DirectFB API is modified with security labels, policy enforcement, and covert channel mitigation codes. Moreover, several coding flaws in DirectFB have been corrected in TrustGraph using compiler-based techniques. Thus, TrustGraph can be viewed as a large patch to a COTS component.

Finally, the simplified graphics subsystem is mostly designed from scratch; thus it is non-COTS.

4 Trusted Networks for High Assurance Systems

4.1 Introduction

Trusted networks cover the gap of trust in the current secure networks. They provide high assurance posture validation and end-device access control. In this chapter we study the trusted network technologies in more detail and propose a new network architecture for high assurance applications. An important realm of high assurance systems is industrial control networks and critical infrastructure applications. Hence, in order to make the study more practical, we explore trusted networks in the context of industrial control systems.

The increased interconnectivity of industrial control networks and enterprise networks has resulted in the proliferation of standard communication protocols in industrial control systems. Legacy SCADA protocols are often encapsulated in TCP/IP packets for reasons of efficiency and cost, which blurs the network layer distinction between control traffic and enterprise traffic. The interconnection of industrial control networks and enterprise networks using commodity protocols exposes instrumentation and control systems and the critical infrastructure components they operate to a variety of cyber attacks.

Security surveys reveal significant increases in external attacks that target critical infrastructure assets [38]. The percentage of external attacks has increased from 26% (1982-2001) to 60% (2002-2006). The entry points in most of the incidents were corporate WANs, business networks, modems, wireless access points, and the Internet.

Several government agencies and industry associations have proposed standards and security best practices for industrial control systems [39, 40, 41, 42]. However, these efforts are based on older technologies and security architectures that rely on the differentiation and separation of enterprise and control traffic. While the efforts are, no doubt, important, the underlying security philosophy exposes industrial control systems to attacks that ex-

ploit misconfigurations, out-of-band connectivity and blind trust in the identities of traffic sources.

However, new technologies are emerging that provide more pervasive security within networks [43]. These technologies push security from perimeter devices such as firewalls to the networked devices themselves. This chapter reviews technologies that can be applied to designing the next generation of secure industrial control systems [44, 45]. The technologies are discussed along with their security benefits and design trade-offs.

4.1.1 Contributions

The contributions of this chapter are enumerated as follows:

- We identify the security threats and challenges in the existing process control architectures.
- We propose a new network architecture based on trusted networks for critical infrastructure.
- We study and identify specific requirements, customizations and configurations of the new architecture in order to achieve high availability required in critical infrastructure systems.
- A side effect of deploying trusted networks is the distribution of filtering rules across many network devices. We formally study the problem of rule conflicts; specifically, we show that the naive approach of distributing rules across the devices may result in rule conflicts.
- An algorithm is proposed for distributing the filtering rules that does not introduce any new rule conflict to the system.
- Finally, benefits of the new architecture are studied in light of how they address the security challenges. Moreover a qualitative evaluation of the architecture is performed against real attack patterns.

To the best of our knowledge, we are the first to propose and study a trusted process control network architecture.

4.2 Control System Security Recommendations

Industrial control systems (ICSs) are highly distributed networks used for controlling operations in water distribution and treatment plants, electric power systems, oil and gas refineries, manufacturing facilities and chemical plants. Generally, an industrial complex comprises two distinct networks: a process control network (PCN) containing controllers, switches, actuators and low-level control devices, and an enterprise network (EN) incorporating high-level supervisory nodes and corporate computers [46]. PCN includes supervisory control and data acquisition (SCADA) systems and distributed control systems [39]. The main components of a PCN are the control server or master terminal unit (MTU), remote terminal units (RTUs), intelligent electronic devices (IEDs), programmable logic controllers (PLCs), operator consoles or human-machine interfaces (HMIs), and data historians.

The National Institute of Standards and Technology (NIST), Institute of Electrical and Electronics Engineers (IEEE), Instrumentation Systems and Automation (ISA) Society, International Electrotechnical Commission (IEC) and Industrial Automation Open Networking Association (IAONA) have specified guidelines for securing ICSs (see, e.g., [39, 40, 41]). In fact, most security best practices recommend the segregation of PCNs and ENs.

Firewalls are often used to segregate PCNs and ENs [39, 46, 47]. A firewall can be configured to block unnecessary services, protocols and ports, thereby providing a higher degree of segregation between a PCN and EN. A router may be positioned in front of the firewall to perform simple packet filtering, leaving the firewall to perform more sophisticated tasks such as stateful filtering and acting as a proxy.

Using a single firewall between a PCN and EN has a serious drawback. This is because the firewall must allow the data historian to have a wide range of access to the PCN. Essentially, each service needs a "hole" in the firewall to operate correctly. Configuring too many holes in the firewall reduces PCN-EN segregation and opens the PCN to a slew of attacks. This problem is typically addressed by creating a "demilitarized zone" (DMZ) [39, 46, 47].

An architecture deploying a DMZ has three zones: an outside zone containing the EN, an inside zone containing the PCN, and a DMZ containing the data historian. Firewall rules are crafted to make the DMZ historian the sole point of contact between the EN and PCN. The historian can access PCN services that provide it data; in turn, the EN is allowed access to the historian. Firewall rules block access to the PCN by all devices.

Most attacks originating in (or passing through) the EN and targeting the historian will not affect the control systems; at worst, they would corrupt the historian's data (a redundant copy of this data is stored elsewhere).

A PCN architecture deploying paired firewalls separated by a DMZ [18, 19] is shown in Figure 4.1. It simplifies the firewall rules and achieves a clear separation of responsibility as the PCN-side firewall can be managed by the control group and the EN-side firewall by the IT group [39, 47]. This architecture is highly recommended for ICSs, and best practices have been identified for configuring the firewalls (see e.g. [40, 41, 46]).

There are also mechanisms to enhance host security in a control system. An important example of such mechanisms is process-based security (PBS) which is implemented for some control devices [48]. It shifts the access control paradigm from a user-based model to a process-based one. In the user-based model, access control rules are tied to user identities. A rogue process, however, can escalate its privilege and damage the system. In the process-based security model, on the other hand, fixed access vectors are strictly tied to process profiles which cannot be modified in the field. PBS reduces the risk of privilege escalation as a result of an attack.

4.3 Security Challenges

Firewall configuration errors can lead to security vulnerabilities. One problem is that firewalls often have large rule sets which are difficult to verify. According to a study by Wool [49], firewall rule sets may have as many as 2,600 rules with 5,800 objects, and a significant correlation exists between rule set complexity and the number of configuration errors. A second problem is that firewalls are usually the main line of defense. Configuration errors enable attackers to exploit holes in a firewall and target the otherwise defenseless devices

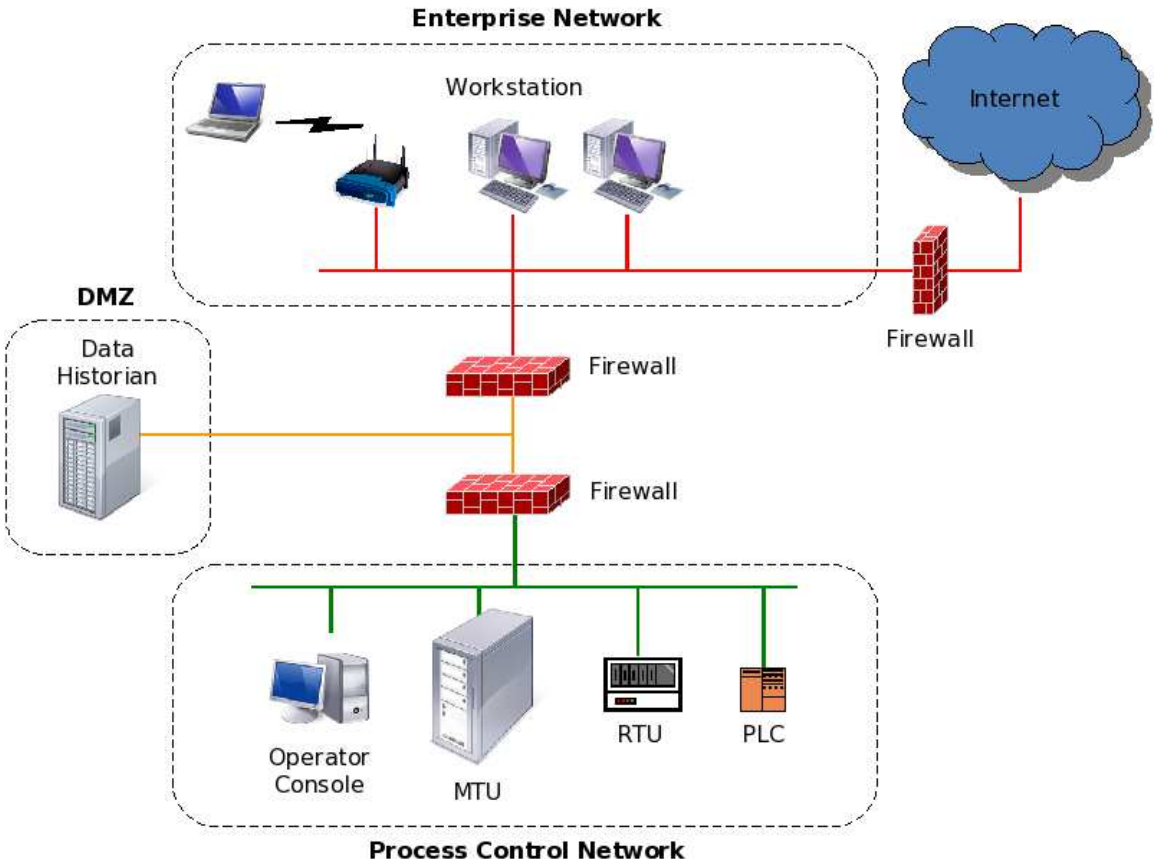

Figure 4.1: Paired firewalls PCN architecture

inside the network.

Wool [49] notes that 80% of rule sets allow "any" service on inbound traffic and insecure access to firewalls. He emphasizes that "the analysis of real configuration data shows that corporate firewalls are often enforcing rule sets that violate well-established security guidelines." The Wool study and others demonstrate that firewall configuration errors pose a real threat to ICS security.

Even properly configured firewalls can be bypassed [50]. This occurs, for example, when a vendor creates a direct (e.g., dial-up) connection to a device for maintenance, or when unsecured wireless access points exist behind a firewall. Firewalls can also be thwarted by tunneling attack traffic using legitimate means (e.g., via a corporate VPN) or by using encryption (firewalls do not inspect encrypted packets). A widely reported firewall breach occurred in January 2003, when the MS SQL Server 2000 worm infected systems at the

Davis-Besse nuclear power plant in Oak Harbor, Ohio [51]. An investigation revealed that a contractor established an unprotected connection to the corporate network which bypassed the power plant firewall and provided a path for the worm.

Vulnerable devices are typically secured by patching their services, updating software or installing the latest version of the devices. However, manual patch/update/version management are difficult and costly tasks, especially when careless users introduce vulnerable (wireless) devices into an industrial control network that establish new entry points for attackers. The mobility of wireless devices makes it difficult for the administrator to manually manage patches by visiting the devices frequently or dedicating a specific time slot for patching the systems. Hence, patch/update/version management should be done automatically and continuously.

Unsecured physical access also exposes ICSs to serious security threats. Open wireless access points and Ethernet ports on office walls enable attackers to enter ICS networks and target critical assets. Nothing in the traditional ICS architecture prevents suspect devices from connecting to the network; thus, serious threats are posed by devices whose hardware, operating systems, executables and/or configurations have been tampered with by attackers.

Many ICS vulnerabilities admit malware such as worms, viruses, Trojan horses and rootkits [51, 52]. ICS security trends reveal that external malware attacks are becoming increasingly common [38]. Finally, rogue users (insiders) are an ever-present threat to ICSs.

4.4 Trusted Process Control Networks

In a traditional network access control model, access is granted to a user without considering the security state of the user's machine. That machine may be running a secure operating system, or it may be a machine that has not been patched for a decade and is riddled with vulnerabilities and malware. Likewise, firewall access control is agnostic about the security status of the device that sends traffic. A port on a machine is opened or not opened to traffic based entirely on the identity of the source.

A trusted network architecture uses information about the hardware and software states of devices in admission and access control decisions. When a device first "joins" the network,

its hardware and software are checked; based on these checks, the appropriate access control rules are applied dynamically to the user, device and traffic. The same principle can be applied to process control architectures. This section discusses technologies that support this concept and their application to ICSs.

4.4.1 Trusted Networks

A trusted network (TN) architecture uses the existing standards, protocols, and hardware devices to extend the concept of "trust" to the network architecture. TNs provide important security services such as user authentication, comprehensive network device admission control, end-device health check, policy-based access control and traffic filtering, automated remediation of non-compliant devices, and auditing.

The Trusted Computing Group (TCG) has promulgated industry standards for TNs [23]. Several commercial TN technologies have been developed, including Cisco TrustSec [24], Cisco CleanAccess [25] (formerly known as Cisco Network Admission Control (NAC) [53, 54, 55]), and Microsoft Network Access Protection (NAP) [56]. Cisco NAC is interoperable with Microsoft NAP; details about their interoperation can be found in [57].

Trusted Network Components

TN component vendors use a variety of names to describe their products. We use generic terms with a bias towards those adopted by Cisco CleanAccess.

A TN has the following components:

- Client device: Every client device must be evaluated prior to admission to a TN.
- Network Access Device (NAD): All connectivity to a TN is implemented via an NAD, which enforces policy. NAD functionality may exist in devices such as switches, routers, VPN concentrators and wireless access points.
- Authentication, Authorization, and Access Control Server (AAA): This server maintains the policy and provides rules to NADs based on the results of authentication and posture validation.

- Posture Validation Servers (PVSs): These servers evaluate the compliance of a client before it can join a TN. A PVS is typically a specialization for one client attribute (e.g., operating system version and patch or virus signature release).

- Posture Remediation Servers: These servers provide remediation options to a client device in case of non-compliance. For example, a server may maintain the latest virus signatures and require a non-compliant client device to load the signatures before joining a TN.

- Directory Server: This server authenticates client devices based on their identities or roles.

- Other Servers: These include trusted versions of Audit, DNS, DHCP and VPN servers [25, 53, 55].

Trusted Network Protocols

TNs leverage existing standards and protocols to implement the required security functionality; this reduces the cost of building TNs.

Protocols used in TNs include IPSec for hardening communications [25, 54], EAP and 802.1x for authentication [24, 54, 55], RADIUS/LDAP/Kerberos for directory services and authentication [25, 54, 55], HCAP for compliance communication [54, 55], and GAME for communication between the AAA and audit servers [54, 58].

4.4.2 TPCN Architecture

A trusted process control network (TPCN) architecture is presented in Figure 4.2. A client device intending to join the network communicates its request to the NAD. The NAD establishes the client device's identity using EAP over the 802.1x protocol and sends the results to the AAA server using the RADIUS protocol. The AAA server returns a list of posture validation requirements and the addresses of the appropriate PVSs.

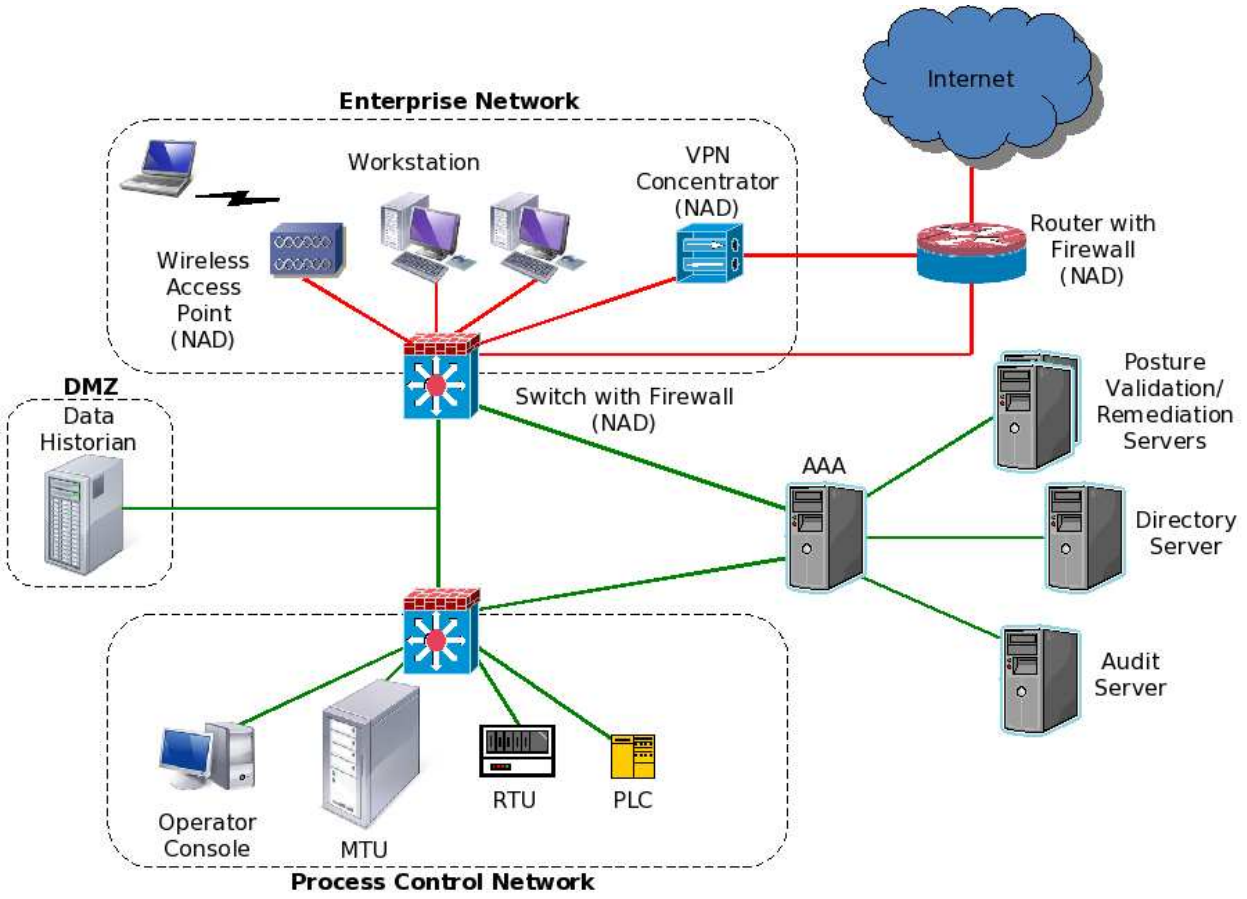

Figure 4.2: Trusted process control network (TPCN)

The client then validates its posture with each of the PVSs. If the client is in compliance, the results are sent to the AAA server using the HCAP protocol. On the other hand, if the client lacks one or more requirements, the appropriate posture remediation servers suggest remediation actions to the client.

The directory server determines the client's group or role. Given all the results from the PVSs and the directory server, the AAA server determines the set of rules that apply to the client's access and traffic and sends them to the NAD for enforcement. From this point on, the client is permitted to communicate via the NAD and all its activities are monitored for policy compliance. Interested readers are referred to [25, 53, 55] for additional details.

The policy held by the AAA server is in the form of an authentication requirement and a list of posture validation requirements. For example, token based authentication may be

required and postures must be validated with the anti-virus server, patch management server and driver validation server. When a client device joins the network, a NAD communicates with an AAA server on behalf of the device. The AAA server authenticates the device and provides rules based on the device's security postures to the NAD. From this point on, the NAD enforces the policy on all ingress and egress traffic to/from the device. For example, an RTU with valid firmware is allowed to communicate with the historian; all other traffic is blocked. The two examples below further clarify the workings of a TPCN.

Example 1: Consider a scenario where an analyst on a workstation intends to connect wirelessly to the PCN to access historical data about plant operations. The workstation connects to a wireless access point (AP) in the enterprise network with NAD functionality. The AP applies the default policy, which is to block all traffic except what is needed to establish trust. The workstation then authenticates with the AP using EAP over the 802.1x protocol to send a stored certificate. The AP uses RADIUS to send the workstation's identity to the AAA server. The AAA server then sends the user's identity to the directory server, which knows the user's role ("analyst"). The AAA server uses RADIUS to send the workstation a list of posture requirements (anti-virus version number and OS patch history). The workstation uses a trusted platform monitor (TPM) chip to sign in and send the posture values to the relevant PVSs, which proceed to validate these values. The patch management PVS discovers that the workstation OS has a missing patch and coordinates with the remediation server to have the appropriate patch sent to the workstation. The PVSs transmit the results back to the AAA server using the HCAP protocol. If the workstation is compliant, the AAA sends a rule set to the AP for enforcement. Since the user role is "analyst," the rule set allows TCP connections to the historian but blocks access to all other devices.

Example 2: Consider a scenario where an RTU intends to join the PCN. The RTU connects to a switch on the factory floor via a network cable; the switch has NAD functionality. The protocols used are the same as in Example 1, so we avoid repetition. The switch authenticates the RTU using the RTU's stored token. The AAA server requires the RTU to validate its configuration with a configuration management server. The RTU sends its configuration to the configuration management server, which returns the successful result to

the AAA server. The AAA server, in turn, sends the appropriate rule set for the compliant RTU to the switch for enforcement. The RTU may now communicate with other RTUs, the MTU and the historian; the switch blocks all other traffic. In the next section, we show how to perform authentication and posture validation without losing availability for important devices. Essentially, the AAA server holds two sets of roles and two sets of policies. The devices can still connect to the network through the backup policy before we ensure that configuration validation does not interrupt service.

4.5 TPCN Requirements and Availability

In order to transform an existing PCN into a TPCN, there are specific changes necessary ranging from a simple patching of the existing software to new hardware and devices. This section discusses the requirements of a TPCN and the cost/security tradeoff. More importantly, the issue of availability is the centerpiece in any process control system architecture. We discuss how to customize a TN to achieve high availability required for a TPCN.

4.5.1 TPCN Requirements

For added security and separation of duty, a TPCN requires at least two NADs (switches with firewalls) and a AAA server (Figure 4.2). An enterprise can add as many PVSs as required, e.g., an anti-virus validation server to ensure that devices have up-to-date virus protection, a patch management server to check that devices have the correct patches and a software validation server to verify the authenticity of embedded device firmware. Various PVSs are available off-the-shelf and often it is only necessary to configure them with the appropriate requirements for the control system. Incorporating multiple PVSs adds to the cost of a TPCN, but enhances security.

All NADs (switches, routers, wireless access points, etc.) must support trusted network functionality. Many vendors offer products with trusted network functionality. Therefore, if an enterprise is already using new equipment, implementing a TPCN may be very cost-effective. Older systems would likely involve significant upgrades, which can be costly. Note

that in a TPCN architecture the firewall functionality is integrated in NADs.

Client devices may need software and firmware upgrades to support trusted network functionality. A trusted network client is required for authentication with the AAA server and for sending posture values. For secure applications, TPM chips can be used to verify configurations and obtain posture signatures. Devices such as RTUs and PLCs do not usually have TPMs; however, as some RTUs already come with built-in web servers, adding TPM to these devices is feasible, especially if government regulations mandate the implementation of trusted ICS architectures.

A client device must be intelligent and configurable to be evaluated by PVSs. At a minimum, the device must be able to run a small piece of software that sends its configuration to a PVS. If the end device is not intelligent (e.g., a simple mechanical relay), it is not necessary to check its configuration; rather, PVSs check the postures of the more intelligent device that controls it (e.g., the RTU that controls the relay). The trusted network client software is available for different platforms off-the-shelf from various vendors, but for special or small devices it may be necessary to develop the piece of code that sends the configuration to PVSs.

In principle, TN can be implemented on top of any physical and link layer (e.g. wireless, LAN, or serial). However, to the best of our knowledge, the current TN technologies only support LAN and wireless media. There are two options to establish trust with serial control devices using the existing TN technology. The first option is to put the trusted network client software in the main device that is connected to the LAN and controls the serial device. In this case, the client software in the main device is responsible for acquiring the posture of the serial device and performing authentication with the AAA server. The client software on the controlling device blocks any communication of the serially connected device before the authentication is successful and the posture is validated. This option has the benefit of using the existing hardware. The other option for connecting serial devices to a TPCN is to use serial-to-Ethernet converters [59, 60] and directly connect the devices to the NAD. These converters translate serial (RS-232/422/485) to TCP or UDP packets and transmit them over the Ethernet. The converter can be configured to use a specific IP address. Serial-to-Ethernet converters can be used in the compact form [59] to connect one serial device or

as a rack [60] to connect multiple serial devices to the Ethernet. Using serial-to-Ethernet converter has the benefit of making serial control devices stand-alone; on the other hand, it has the drawback of extra hardware cost and extra latency due to the network delays. The choice of which option to use when connecting serial devices to a TPCN depends on the acceptable cost and the latency requirements for the specific control system.

4.5.2 TPCN Availability

To apply updates to the system, the administrator puts the new requirements in the AAA or posture validation servers. After that point, the AAA server informs end-devices of the new policy. If they have the update, they establish its existence with a posture validation server and continue working in the network. However, if they do not have the new requirements, the server provides them with appropriate patches (or installs the patches automatically on them). Now we discuss the use of backup roles and policies to prevent the new requirements from interrupting the service.

TPCNs have the same availability issues as traditional PCNs—applying patches can cause components to crash. Therefore, every patch or update must be tested thoroughly before being placed on the AAA server. Exact replicas of TPCN components should be used for testing. If concerns exist after testing, a backup device may be placed in the TPCN. In such a situation, the AAA server holds two different policies for the device. One policy is associated with the actual role and the other policy with the backup role. The backup policy does not enforce the new requirement on the backup device until the actual device is verified to function correctly with the patch. It is only then that the administrator applies the requirement to the backup device as well. Note that if the actual device is affected by the patch, the backup device can function correctly since it is not required by its policy to have the patch in order to connect to the network. TPCNs do not positively or negatively affect system availability; they merely enforce the requirements. It is the testing phase, before the specification of a requirement, that determines whether or not system availability is affected.

To enhance the availability of a TPCN, redundant servers must be used in the architecture. If important servers of a TPCN fail, devices cannot join the network, endangering

the availability of the control system. Commercial TN servers support a mode called "high-availability" (HA-mode). In this mode, important TN servers (such as AAA or PVSs) have a standby replica available. A "heartbeat" signal is exchanged frequently between the primary and the standby server over a dedicated serial connection or using UDP packets over the Ethernet. If a failover occurs and the heartbeat signal stops (e.g., as a result of a crash or a restart), the standby server immediately takes the role of the failed server, preventing an interruption in the service. Since availability has a high priority in the context of process control, it is recommended that HA-mode is used for TPCNs.

It is important to differentiate between backup devices and standby servers. Backup devices and roles prevent failure of the control devices (e.g., MTUs or RTUs) as a result of applying new patches or updates. The use of backup policy ensures that the backup device can provide service if the main device fails after an update. Standby servers, on the other hand, increase the availability of the TPCN infrastructure, ensuring that a control device can access the network at any time even when the primary server has crashed (using automatic failover mechanism). These two customizations address the high availability needs of a TPCN, minimizing the risk of service interruption.

4.6 NAD Rule Conflicts

Conflict in the firewall rules (more generally referred to as filter conflicts) always endangers the security or availability of instrumentation and control systems. Such conflicts may result in unwanted traffic being allowed to the process control network, exposing it to attacks or legitimate traffic being denied, endangering the availability of the system.

In this section, we study the nature of conflicts in firewall rules and show that the total number of effective conflicts in a distributed filtration environment is no greater than that of a traditional network.

4.6.1 Filter Conflicts

Filter conflicts happen when there is an ambiguity in classifying packets using a set of filtration rules. Although we study filter conflicts in the context of firewall rules, they can exist in any network access device such as a router, VPN, or QoS device that classifies packets based on a ruleset.

Each filter R is an n-tuple (R[1], R[2],..., R[n]) where R[i] defines a subset of values acceptable for that field. Each field is usually defined as a prefix. For instance, the prefix 10.* for source IP refers to the subset of all IP quartets that have 10 in their first entry. A rule is a filter R followed by an action *Act(R)* (typically *allow* or *deny*). Although there can be many different fields in each filter depending on the device, we focus on the five most common fields in firewall rules: source IP, source port, destination IP, destination port, and protocol. Note, however, that the analysis presented here is general and can be applied to any n-tuple.

We use the definition by Hari et al. [61] for rule conflicts.

Definition 1: Rule A is said to be the prefix of rule B if for every field i, A[i] is the prefix of B[i] and A[i] is a strict prefix of B[i] for at least one i.

Two rules conflict with each other if and only if their filters intersect (i.e. there exists traffic to which both rules apply), one is not the prefix of the other, and their actions are not equal. More formally, we have the following.

Definition 2: Rules A and B conflict with each other if and only if all of the following hold:

1. For all i, A[i] and B[i] are not disjoint.
2. A is not a prefix of B.
3. B is not a prefix of A.
4. Act(A) is not equal to Act(B).

Definition 3: A ruleset is an *ordered* set of rules.

For example, consider the following rules in an internal firewall inside a PCN:

```
R1: <PCN.MTU.*   any   PCN.RTU_farm1.*   any   TCP> allow
R2: <PCN.MTU.*   any   PCN.RTU_farm2.*   any   TCP> allow
R3: <PCN.*   any   PCN.RTU_farm1.*   any   TCP> deny
R4: <PCN.*   any   PCN.RTU_farm2.*   any   TCP> deny
R5: <PCN.RTU_farm1.*   any   PCN.*   any   TCP> allow
R6: <PCN.RTU_farm2.*   any   PCN.*   any   TCP> allow
```

The first two rules ensure that any connection from MTU IP addresses to RTU farms is allowed. These two rules do not conflict with any other rule because either the fields are disjoint or they are prefixes of another rule. For instance, R1 is disjoint from R4, R5, and R6 while it is a prefix of R3.

However, there are conflicts between R3 and R6, and R4 and R5. R3 and R6 both apply to traffic from any address in RTU_farm2 to any address in RTU_farm1, yet R3 denies the traffic while R6 allows it. There is a similar conflict between R4 and R5.

In many firewall implementations, the first encountered rule is given higher priority. However, in a situation like this, there can be no ordering which provides the desired behavior. This happens when rule ordering results in a cycle as shown by Hari et al. [61]. The solution in such a situation is to "add" rules that cover the intersection of the conflicting rules to the ruleset. Such rules are called "conflict resolution" rules. For instance, R7 is a conflict resolution rule for R4 and R5.

```
R7: <PCN.RTU_farm1.*  any  PCN.RTU_farm2.*  any  TCP> deny
```

4.6.2 Conflicts in Distributed Firewalls

In this section, we show that the total number of conflicts in a distributed firewall environment (such as TPCNs) is no greater than the number of conflicts in traditional architecture. The total number of conflicts in a TPCN is the number of conflicts in all of the network access devices' (NAD) rulesets. Note that the rules are distributed among NADs and each NAD potentially has a much smaller ruleset which makes it less complex and more manageable. Now we show that the total number of conflicts is still bounded by the number of

conflicts in the single ruleset model.

To prove this property, first consider the simple "ordered subset" operation on the ruleset. We say that ruleset $\Re_A$ is an ordered subset of $\Re_B$ if and only if:

1. Every rule in $\Re_A$ is in $\Re_B$.

2. For each pair of rules R_i and R_j in $\Re_A$, if R_i precedes R_j in $\Re_B$, then R_i precedes R_j in $\Re_A$ too.

The ordered subset operation is simply taking a number of rules from a ruleset while preserving the order. We cannot state anything about the number of conflicts in $\Re_A$. It can even be more than the number of conflicts in $\Re_B$. To observe this, consider three rules: R1, R2, and R3 in which R1 and R2 conflict and R3 is the intersection of the two rules that resolves the conflict. If $\Re_B$ contains R1-R3 and $\Re_A$ only contains R1 and R2, then $\Re_B$ is conflict-free, but $\Re_A$ has one conflict. The increase in the number of conflicts arises from the fact that the ordered subset operation may eliminate conflict-resolution rules.

Now consider the following method for distributing a single firewall ruleset among different NADs.

1. *Every* rule in the firewall ruleset for which the source or destination is in the subnet controlled by a NAD is added to that NAD's ruleset.

2. For each pair of rules R_i and R_j in the NAD ruleset, if R_i precedes R_j in the firewall, then it precedes in the NAD ruleset too.

This method puts every rule related to a subnet in the subnet's NAD ruleset while preserving the order. Note that there can be bad rules in the firewall (as a result of configuration mistakes) that do not apply to any subnet controlled by that firewall. These rules are not included in any NAD ruleset. Hence, here we consider "effective conflicts" which involve flows that actually reach the firewall.

Theorem 1: In a distributed firewall system in which each NAD ruleset is constructed using the above method, the total number of effective conflicts is no greater than that of the single firewall.

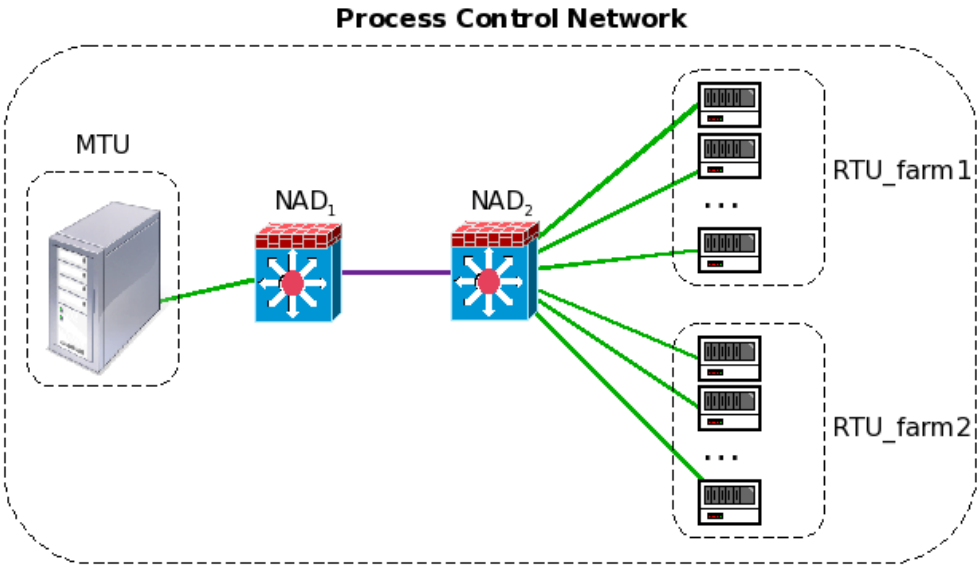

Figure 4.3: NADs and subnets inside a TPCN

Proof: Suppose not. Let R1 and R2 be the rules that conflict under partition but not in the firewall. Note that R1 and R2 are in the same NAD without a conflict resolution rule, but there must be a rule R3 in the firewall which covers the conflict, yet is not present in the NAD ruleset. R3 is the intersection of R1 and R2, and thus is more restrictive than either of them. But the definition of the distribution method implies that R3 must be included in the NAD ruleset; hence, the presumed conflict between R1 and R2 does not exist. This is a contradiction.

As an example, consider the architecture in Figure 4.3 and the following ruleset.

```
R1: <PCN.*   any   PCN.RTU_farm1.*   any   TCP> deny
R2: <PCN.RTU_farm2.*   any   PCN.*   any   TCP> allow
R3: <PCN.*   any   PCN.MTU.*   any   TCP> deny
R4: <PCN.RTU_farm2.*  any  PCN.RTU_farm1.*  any  TCP> deny
```

NAD_2 ruleset contains rules R1, R2, and R4 while NAD_1 contains R3. Note that NAD_2 ruleset includes R4, which resolves the conflict between R1 and R2.

Theorem 1 shows that although in the distributed firewall architecture each NAD potentially contains a smaller ruleset which makes it more manageable and less error-prone, the total number of effective conflicts is no greater than that of the traditional architecture.

Firewall rule conflicts can endanger the security of the network by allowing attack traffic to enter the network. More importantly, they can endanger the availability of the network by blocking legitimate traffic. Considering the importance of availability in PCNs, it is crucial to address this problem in the context of TPCNs. In this section, we provided a recipe for distributing PCN firewall rules between TPCN NADs to avoid introducing new rule conflicts. Moreover, we prove that if the rules are divided using this method, the number of total conflicts in the NADs is no greater than the lumped ruleset model.

4.7 TPCN Evaluation

The benefits of a TPCN are best seen in light of how it addresses the security issues that impact traditional networks. A TPCN addresses the following security issues either partially or completely.

- **Firewall Configuration Errors (partial):** A TPCN breaks the set of firewall rules into smaller rule sets associated with each access control group or role. These rule sets are sent by the AAA server to the NADs for enforcement upon completion of the authentication phase. According to Wool [49], the number of configuration errors decreases logarithmically as the rule set complexity decreases. Because a TPCN has smaller rule sets, the potential for firewall configuration errors is correspondingly lower. Moreover, access rules in a TPCN are defined based on groups or roles, not just IP addresses; this helps reduce confusion and, consequently, configuration errors. Note that configuration errors will never be completely eliminated; therefore, TPCN only provides a partial solution to the problem.

- **Bypassing Firewalls (Complete):** TPCNs explicitly address this issue by securing all NADs and requiring them to establish trust relationships with client devices before forwarding traffic (including wireless traffic and VPN traffic). Furthermore, the access control and traffic rules are applied at every access point. It is not possible to bypass the rules by hooking a line behind a firewall; this is because the line's switch (access point) enforces the rules.

- **Vulnerable Devices (Partial):** In a traditional network architecture, patch/update/version/configuration management is performed manually by the network administrator. This is an extremely difficult task for remote and mobile devices. As a result, it may be done less frequently than recommended or it may be simply ignored. In a TPCN, the state of a device is checked automatically before it can join the network. Moreover, its behavior is continuously monitored upon entry and status checks can be performed at the desired frequency. Consequently, a TPCN is less vulnerable to known attacks. Note, however, that a TPCN is still vulnerable to zero-day attacks.

- **Unsecured Physical Access (Complete):** TPCNs again address this problem by enforcing security policies on NAD ports. This is sometimes referred to as "port-based access control." Thus, a malicious or careless user cannot hook a device to an open Ethernet port and gain entry into the network. Note also that ports on TPCN switches and wireless access points do not forward traffic until trust relationships are established with the communicating entities.

- **Malware (Partial):** The compliance rules enforced on devices before and after joining a TPCN reduce the likelihood of infections by malware. A SCADA security study [38] notes that "the majority of worm events occurred months or years after the worm was widely known in the IT world and patches were available." This implies that the majority of incidents can be prevented by enforcing compliance rules before a node joins a network. Since nearly 78% of the (external) SCADA security incidents are caused by malware [38], TPCN incidents are reduced dramatically. Nevertheless, a TPCN remains vulnerable to zero-day attacks.

- **Untrusted Devices (Complete):** TPCNs address this problem explicitly by verifying the signatures of the critical components of a device using the TPM chip and also checking the device status. Note that if the TPM chip is trusted, the device can attest to its identity.

- **Untrusted Users (Partial):** By using stronger authentication methods and clearly defining user roles, TPCNs prevent attacks such as password cracking/stealing, access

violations and impersonation. Also, by blocking all unnecessary accesses, TPCNs partially prevent accidents caused by careless insiders that account for more than 30% of all security incidents [38].

We employed the Common Attack Pattern Enumeration and Classification (CAPEC) database [62] to further compare the TPCN architecture with traditional PCN designs. CAPEC contains twelve attack categories along with their descriptions, prerequisites, methods, consequences and mitigation strategies. We consider nine attack categories (with 31 attack patterns), which we believe are meaningful in the ICS context and showcase the differences between TPCNs and traditional PCNs. For example, while buffer overflow attacks are effective against software applications, they are not relevant when evaluating network designs.

Table 4.1 presents the results of the comparison. The descriptor H (high) means that an attack is performed with little effort and cost; M (medium) implies that an attack is still possible but requires expert knowledge and is costly; L (low) indicates that an attack is highly unlikely or involves enormous effort, time and/or cost. The last column shows the security controls provided by a TPCN to address the attack (if any).

Considering the 31 total attack patterns, a PCN is vulnerable to nineteen (61.3%) high, nine (29%) medium, and three (9.7%) low feasibility attacks. On the other hand, a TPCN is vulnerable to only two (6.5%) high feasibility attacks along with nine (29%) medium and twenty (64.5%) low feasibility attacks. Note that this is a qualitative comparison of the two architectures; the quantitative assessment of network architectures based on security metrics is an open research problem and is beyond the scope of this work.

Note that all of the intelligent and configurable devices on a trusted network must be authenticated and posture validated. If the legacy devices are allowed to join the network without such authentication and validation, it can potentially eliminate all the benefits of a TPCN. We discussed earlier how to incorporate client functionality in legacy and small devices.

We plan to implement a prototype TPCN on top of a cyber-security testbed that we have developed [63]. The testbed is developed to perform assessments and study attack/defense

scenarios in a large scale power infrastructure. It consists of a number of real and emulated devices and two different simulators. Real RTUs, control station systems, and historians have been used in the testbed. We have also emulated a number of IEDs. The simulation of power generation and distribution is done using PowerWorld, a power grid simulator which is connected to the real devices [64]. It provides the grid parameters to the devices and issues commands in some cases. All network communications are passed through RINSE [65], a network simulator which simulates the communication infrastructure.

For the TPCN prototype, we plan to use Trusted Network Connect (TNC) [23] open source health check protocols (IF-MAP). Legacy control devices can be augmented with simple open source clients to authenticate and send posture information. TNC's open source projects can also be used for simple PVS and AAA servers. Many of the existing access points (Ethernet switches and wireless access points) in our testbed already support NAD functionality.

Trusted network technology can help address the challenges involved in securing industrial control systems that are vital to operating critical infrastructure assets. Adding trust to industrial control networks eliminates security problems posed by inadequate controls, non-compliant devices and malicious users. It dramatically reduces vulnerabilities to malware attacks that constitute the majority of external attacks. The likelihood of internal attacks is also reduced via compliance verification, port-based access control, device and user authentication, and role-based access control.

Table 4.1: Feasibility of attack patterns

Category	**Attack Pattern**	**PCN**	**TPCN**	**TPCN SC**
Abuse of Functionality	Inducing Account Lockout	H	L	Strong Authentication
	Exploiting Password Recovery	H	L	Strong Authentication
	Trying All Common Application Switches and Options	H	L	Configuration Verification
	Exploiting Incorrectly Configured SSL Security Levels	H	L	Configuration Verification
Spoofing	Faking the Source of Data	M	L	Message Authentication

Table 4.1: Feasibility of attack patterns (continued)

Category	Attack Pattern	PCN	TPCN	TPCN SC
	Spoofing Principal	H	L	Strong Authentication
	Man-in-the-Middle Attack	H	L	Device Authentication
	Creating a Malicious Client	M	L	Accounting
	External Entity Attack	H	L	VPN Access Control
Probabilistic Techniques	Brute Forcing Password	L	L	Strong Authentication
	Brute Forcing Encryption	L	L	N/A
	Rainbow Table password cracking	L	L	Strong Authentication
	Manipulating Opaque Client-based Data Tokens	M	M	N/A
Exploiting Authentication	Bypassing Authentication	H	L	Port-based Access Control
	Reflection Attack in Authentication Protocol	H	H	N/A
	Exploiting of Session Variables, Resource IDs and other Trusted Credentials	M	M	Software Verification
Resource Depletion	Denying Service via Resource Depletion	H	M	Compliance Verification
	Depleting Resource via Flooding	H	M	Traffic Filtration
Exploitation of Privilege or Trust	Manipulating Writeable Configuration Files	H	L	Configuration Verification
	Lifting credential(s)/key material embedded in client distributions	M	L	Software Verification
	Lifting cached, sensitive data embedded in client distributions	M	L	Software Verification
	Accessing Functionality Not Properly Constrained by ACLs	H	M	Small Rule Sets
	Exploiting Incorrectly Configured Access Control Security Levels	H	M	Role-based Access Control
Injection	Manipulating User-Controlled Variables	H	L	Configuration Verification
	Manipulating Audit Log	H	L	Audit Verification
	Poisoning DNS Cache	H	L	Trusted DNS
	LDAP Injection	H	H	N/A
	Sniffing Information Sent Over Public Networks	M	M	IPSec
Protocol Manipulation	Manipulating Inter-component Protocol	M	M	N/A

Table 4.1: Feasibility of attack patterns (continued)

Category	Attack Pattern	PCN	TPCN	TPCN SC
	Manipulating Data Interchange Protocol	M	M	N/A
Time & State	Manipulating User State	H	L	Configuration Verification

5 Evaluation of Patch Management Strategies

5.1 Introduction

Successful patch management is crucial to the security of large organizations. New vulnerabilities are discovered in software applications almost every day. These vulnerabilities open a gateway for attackers to penetrate secure systems and exploit their resources. In response to discovery of vulnerabilities, software vendors develop patches to fix them. However, secure and mission critical organizations cannot apply new patches without testing them. This is because a new patch can break the system resulting in loss of functionality, or more importantly, it can potentially open a new vulnerability in the system, thus resulting in loss of security. As a result, in a secure patch management system, there are two processes after a vulnerability is discovered and before the patch is applied to the software: one is the process to develop a patch by the vendor and the other is the pre-deployment testing process. Both of these processes take some time, during which the system is susceptible to attacks using that known vulnerability.

Pre-deployment testing is usually done in an ad-hoc fashion, inspecting the patch for known problems or applying it to replicate machines to see whether it breaks the system. This method, although effective under many circumstances, does not consider the tradeoff between the time spent for testing a patch and the window of exposure. Given that a faulty patch can introduce new vulnerabilities to the system, the tradeoff is non-trivial.

In this chapter, we study real-world vulnerability discovery for three popular web browsers: Mozilla Firefox 2, Microsoft Internet Explorer 7, and Apple Safari 2 [66]. First we study an analytical model for the trade-off between pre-deployment testing and the total number of open vulnerabilities in a system. This model uses the exponential vulnerability discovery model (AML model) [67] which is shown to fit the real data well [68]. We fit the model to

the vulnerability discovery data for the three web browsers.

To evaluate the trade-off, we then develop a stochastic model for the patch management system and solve it using a simulation tool. From both the simulation and the analytical model, an optimal pre-deployment testing time is obtained. The optimal time ensures that pre-deployment testing is neither so short that patching introduces new vulnerability to the system, nor so long that the system has a long window of exposure.

Finally, we validate the results by showing that the simulation and analytical models fit the real-world vulnerability data for the web browsers with small errors and that simulation results match the analytical model.

5.1.1 Contributions

Although many models have been proposed for vulnerabilities or faults in software or hardware systems, to the best of our knowledge, we are the first to quantitatively study the trade-off of pre-deployment testing and find the optimal testing time. The related work on vulnerability lifecycle and vulnerability discovery models is presented in Sections 5.2.1 and 5.2.2. The contributions of this chapter are as follows:

- We study patch management and especially the problem of pre-deployment testing.
- We develop an analytical model for vulnerability discovery, patch development, and patch testing and analytically find the optimal testing time.
- Also a stochastic model is developed for vulnerability discovery and patch testing. The model is solved using simulation.
- We use the analytical and stochastic models on real vulnerability information from the National Vulnerability Database (NVD). It is found that the error in our model is less than 12% for real vulnerabilities.

5.2 Patching Process

This section describes the process of patching a piece of software as well as probabilistic models built to describe different events involved in such a process.

5.2.1 Life Cycle of Vulnerabilities

A vulnerability has an eight-phase life cycle (some of the phases are taken from the literature [69]):

Introduction : In this phase, the vulnerability is released as a part of software. This can happen during the development or maintenance of the software.

Discovery : This is the time when the vulnerability is discovered.

Private exploitation : During this period a small group of attackers use the vulnerability without the general public knowing.

Disclosure : The time when the vulnerability is published.

Public exploitation : The phase during which the general communities of hackers use the vulnerability.

Patch release : When the vendor fixes the vulnerability with a patch.

Patch testing : Time during which the organization tests the patch for problems or new vulnerabilities.

Patch deployment : When patch is applied to the machine(s).

Vulnerabilities can be avoided during design and implementation phases of software development using testing, verification, and (semi)formal method techniques. However, little can be done for existing vulnerabilities before they are disclosed. So, we focus on the life cycle of the vulnerability after its disclosure. In fact some studies [70, 71] show that the majority of attacks exploit publicly known vulnerabilities. It is important for an organization to find vulnerabilities as quickly as possible and find or develop appropriate patches to

fix them. Vulnerability Discovery Models (VDMs) and Vulnerability Exploitation Models (VEMs) try to capture dynamics involved in this element of patch management.

Many references emphasize the importance of pre-deployment testing for successful application of patches [72]. However, there is a strong tradeoff between pre-deployment testing and the vulnerability of a system. On one hand, more time spent on testing means fewer new vulnerabilities introduced to the system and a more successful patching process. On the other hand, it opens a bigger window of opportunity for attackers to exploit that specific vulnerability. None of the previous works consider this tradeoff and they mainly refer to testing as an "always-good" strategy. We show that there exists an optimal amount of testing which results in the minimum vulnerability. To the best of our knowledge, this tradeoff has not been studied before.

5.2.2 Vulnerability Discovery Models (VDM)

In this section, different models proposed for the vulnerability discovery process are studied. There have been several attempts to model vulnerability discovery. Some of these models use similar models from other fields of science (e.g. thermodynamics) and apply them to vulnerability discovery. Others formulate the underlying process using differential equations and/or fit models to real vulnerability data.

Anderson Thermodynamic Model (AT)

Anderson [73] proposes this model for vulnerability discovery. He argues that the number of vulnerabilities discovered at each instant is inversely proportional to time. Denote the number of new vulnerabilities at each time by $w(t)$ and the total cumulative number of vulnerabilities by $\Omega(t)$. The AT model is described by Equation 5.1 in which a and k are application specific constants. Hence, $\Omega(t)$ has a logarithmic form.

$$w(t) = \frac{k}{a \times t} \tag{5.1}$$

Rescorla Exponential Model (RE)

Rescorla [69] builds this model to fit real data. In the RE model, $w(t)$ decays exponentially with time. Thus, $\Omega(t)$ exponentially approaches a fixed value which is the total number of vulnerabilities in the system. N and a are the application specific constants in the RE model.

$$\Omega(t) = N \times (1 - e^{-at}) \tag{5.2}$$

Logarithmic Poisson Model (LP)

The LP model [74] expresses the total number of vulnerabilities using Equation 5.3.

$$\Omega(t) = a \times ln(1 + b \times t) \tag{5.3}$$

In the LP model, a and b are the constants which should be found by fitting the model to vulnerabilities of a specific application.

Alhazmi-Malaiya Logistic Model (AML)

The AML model [67] is based on capturing the underlying process of vulnerability discovery. It is observed that the attention given to a piece of software increases after its introduction, resulting in discovery of many vulnerabilities. It peaks at some time and after a while it gradually declines because of the fact that new versions of the software are introduced and fewer users use the older version.

Based on the above assumption, the AML model expresses $\Omega(t)$ using Equation 5.4.

$$\frac{d\Omega(t)}{dt} = A \times \Omega \times (B - \Omega) \tag{5.4}$$

The time domain solution to this model is given by Equation 5.5.

$$\Omega(t) = \frac{B}{B \times C \times e^{-ABt} + 1} \tag{5.5}$$

In Equations 5.4 and 5.5, A, B, and C are the application specific constants.

Alhazmi et al. [68] evaluate different models explained above against real data from vulnerability databases. They perform various statistical tests on the models and evaluate their deviation from real-world data. It is found that the best model for vulnerability discovery is the AML model. It fits real vulnerability data for all different softwares and has the least error while others fail to fit some real experiments.

In this work, without loss of generality, we use the AML model fitted to real-world data as our vulnerability discovery model. Nevertheless, the analysis and simulation are general and they can use any vulnerability discovery model.

5.2.3 Patch Development

Patch development is another random process involved in patching and it refers to the process of making a fix for a known vulnerability by the vendor. Although speeding up, patch development is still a slow and time consuming process. Mean patch development time depends strongly on the vendor of the software. Symantec [75] reports mean patch development time for different web browsers. This time for Firefox and Internet Explorer is five days while for Safari it is three days.

We model vendor dependency and consider different mean patch development times.

5.2.4 Patch Testing and Deployment

Testing and deployment are the last phases of the patching process. Pre-deployment testing is an important part of this process. Although it is recommended by previous studies, we argue that it is a double-edged sword. It can help correctly apply new patches and prevent introduction of new vulnerabilities. At the same time, it provides attackers a window of time to attack the system before testing is done. Deployment is usually fast compared to patch development and testing, taking on the order of a few minutes.

5.3 Analytical Model

In this section, we describe an analytical model to show the tradeoff between pre-deployment testing and the vulnerability of the system.

Although some studies describe vulnerability exploitation models (VEMs) [76], they are mostly limited to the specific software application under study and the specific attacker model. For instance, the exploitation model of a worm is different from that of a hacker targeting a specific organization. To keep the model general and avoid such specificities, we do not use an exploitation model in this work. We simply study vulnerabilities in a system and use the total number of open vulnerabilities at each time as a measure of the susceptibility of the system.

Assume that the number of new vulnerabilities found in a piece of software at time t is given by $w(t)$. Note that $w(t)$ is discrete in time; however, it can be approximated with little error with a continuous time function. Further assume that the cumulative number of vulnerabilities in a system is given by $\Omega(t)$, which approaches a final value as time passes. This final value is the total number of vulnerabilities that will ever be discovered for that software. This function does not account for hidden vulnerabilities that are never discovered. Nonetheless if a vulnerability is forever undiscovered, it does not pose a threat and it is of little interest.

If we denote the patch development time by T_P and pre-deployment testing time by T_T, the total number of open vulnerabilities at time t for a perfect patching system (in which patches introduce no new vulnerability) is given by Equation 5.6.

$$V_{perfect}(t) = \int_{t-T_P-T_T}^{t} w(\tau)d\tau \tag{5.6}$$

Equation 5.6 is obtained using the fact that every vulnerability discovered from time 0 to $t - T_P - T_T$ is patched by time t. The only open vulnerabilities are those for which no patch has been developed yet or those under test.

We know that in reality patching is not perfect. In fact, each patch can introduce a new vulnerability to the system. Assume that on average, each untested patch introduces f new patches to the system. If f is greater than 1 (each patch introduces more than one new

vulnerability on average), the system is unstable and the number of vulnerabilities grows indefinitely with time, so we study the system for $0 \leq f < 1$. If no testing is done, the number of new vulnerabilities by previous patches at time t is given by Equation 5.7.

$$V_{faulty}(t) = f \times \int_0^{t-T_P-T_T} w(\tau)d\tau \tag{5.7}$$

Pre-deployment testing, however, can find some of these new vulnerabilities before applying the patch to the software. Unfortunately, to the best of our knowledge, no real data is available for the number of new vulnerabilities discovered when testing a patch. However, these vulnerabilities are related to the same software version and the same vendor. As a result, we assume that they can be modeled using the vulnerability discovery trend for that software. Since the vulnerability discovery trend heavily depends on the software and its version, but it is accurate for a specific version of a software application, we believe that this assumption is valid.

Consequently, if each patch is tested for T_T before it is deployed, the number of new vulnerabilities introduced during testing is given by Equation 5.8.

$$V_{tested-faulty}(t) = [1-\Omega(T_T)-\Omega(0)] \times V_{faulty}(t) = f \times [1-\Omega(T_T)-\Omega(0)] \times \int_0^{t-T_P-T_T} w(\tau)d\tau \tag{5.8}$$

Equation 5.8 is obtained from the fact that during testing $\Omega(T_T)$ vulnerabilities are discovered in the patch.

Some of the VDMs have a small non-zero value at time 0 as an artifact. The term $\Omega(0)$ is added to the equation to compensate for that. If no such artifact exist, the multiplier can be simplified to $(1 - \Omega(T_T))$.

Note that Equation 5.8 is correct for $\Omega(T_T) - \Omega(0) < 1$. This is because a patch on average has at most one new vulnerability. The total number of open vulnerabilities at time t follows Equation 5.9.

$$
\begin{aligned}
V_{Total}(t) =& V_{perfect}(t) + V_{tested-faulty}(t) \\
=& \int_{t-T_P-T_T}^{t} w(\tau)d\tau + f \times [1 - \Omega(T_T) - \Omega(0)] \times \int_{0}^{t-T_P-T_T} w(\tau)d\tau \\
=& \Omega(t) - \Omega(t - T_T - T_P) + \alpha(f, T_T) \times [\Omega(t - T_T - T_P) - \Omega(0)]
\end{aligned} \tag{5.9}
$$

In Equation 5.9, for simplicity, the term $f \times [1 - \Omega(T_T) - \Omega(0)]$ is denoted by $\alpha(f, T_T)$. The tradeoff can be observed from Equation 5.9. By increasing the testing time (T_T), the first term grows because the window of exposure (integral limits) widens. On the other hand, the second term shrinks because more testing results in more faults discovered.

The optimal pre-deployment testing at each time t is the amount of T_T that minimizes Equation 5.9, the total number of open vulnerabilities at time t. If it is desired to have one optimal T_T for all times and remove the time dependency, the optimal T_T is given by Equation 5.10.

$$
Optimal\ T_T = T_T \mid min \int_{0}^{L} V_{Total}(\tau)d\tau \tag{5.10}
$$

In Equation 5.10, L is the lifetime of the software. The optimal testing time minimizes the number of open vulnerabilities at all times. The upper bound of the integral in real-world applications is about two or three years. After this time, there are usually very few new vulnerabilities discovered for that software.

The analysis clearly shows that pre-deployment testing is not always good. There is a certain amount of testing which minimizes the number of vulnerabilities in the system. More testing increases the window of exposure for little gain and is detrimental to the security of the system.

5.4 Stochastic Model

This section describes the stochastic model of patch management. We simulate the stochastic model to find the optimal testing time using simulation as well. The model includes

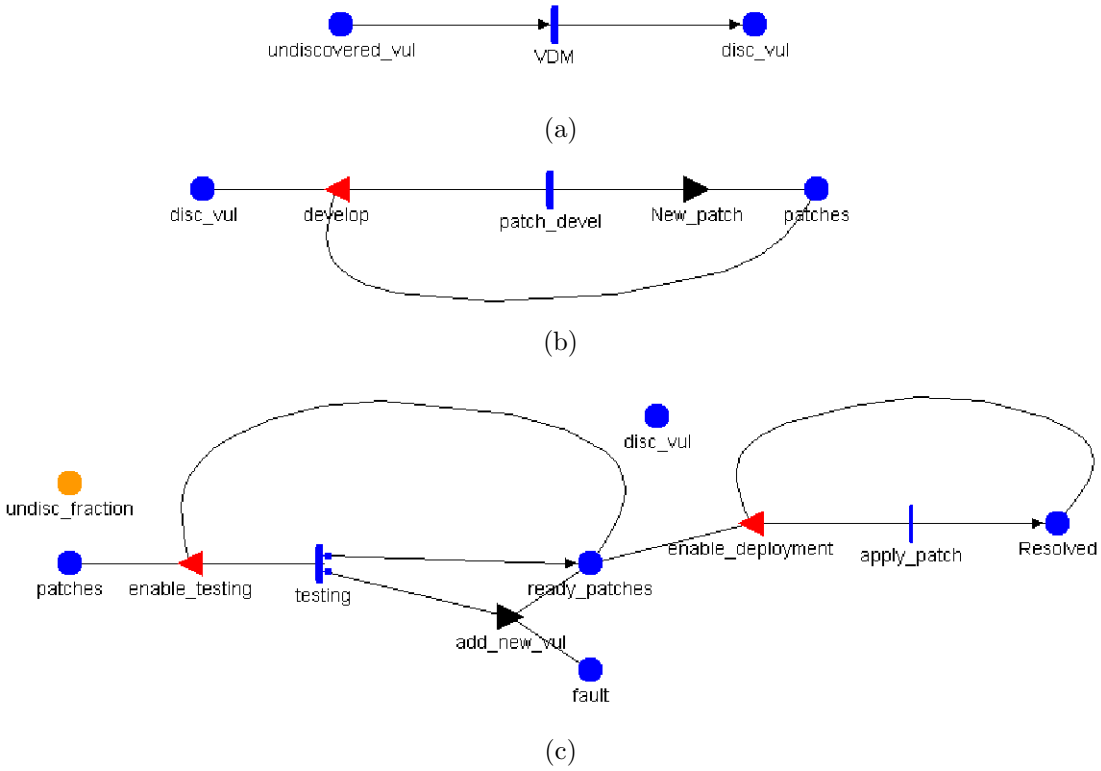

Figure 5.1: The stochastic model of the patching system

vulnerability discovery as well as patch development, testing, and deployment processes.

Patching is modeled in this work using Stochastic Activity Networks (SANs) [77]. SANs are generalized forms of Stochastic Petri Nets (SPNs.) In addition to the components of SPNs, SANs also include input gates (shown by triangles pointing left), output gates (shown by triangles pointing right), probabilistic cases (shown by small circles on activities), and instantaneous activities (shown by thin vertical lines). Input gates specify general enabling conditions, output gates define general completion functions, cases represent a probabilistic placement of tokens in the outputs with a specific probability for each case, and instantaneous activities denote activities that are completed in zero time. Mobius [78] is the tool used for simulation of the SAN model. All times in the model are expressed in months. Here, the details of the SAN model are explained and different choices of the parameters are discussed.

The model consists of three sub-models (Atomic models) illustrated in Figure 5.1. They model vulnerability discovery, patch development, and patch testing/deployment processes.

The three sub-models are joined into a composite patch_system model using "Join" representation. This means that places with the same name in different sub-models refer to the same place in the composite model.

The first Atomic model, "vulnerability," expresses the vulnerability discovery model. In this model, two places contain undiscovered and discovered vulnerabilities and an activity (VDM) takes tokens from the former and puts them in the latter (Figure 5.1a). The initial marking of the undiscovered vulnerabilities is equal to the total number of vulnerabilities. We set the initial marking of the discovered vulnerabilities as one (and not zero) because making this marking zero also sets the discovery rate to zero and no vulnerability will ever be discovered.

For our simulation, we use the AML vulnerability discovery model, although the stochastic model is general and any VDM can be used for it. The choice of AML model is because it fits the real vulnerability discovery data from the three web browsers with small error.

To use the AML model, the rate of VDM activity is set to $A \times \Omega \times (B - \Omega)$ in which Ω is the marking of the discovered vulnerability and A and B are global variables. Note that VDM is a variable rate activity in which the rate is a function of the vulnerabilities discovered so far. In the stochastic model, we do not use the time domain solution for $\Omega(t)$; rather, the underlying differential equation describing Ω (i.e. $\dot{\Omega} = A \times \Omega \times (B - \Omega)$) is used to model vulnerability discovery. This is one of the differences between the analytical and stochastic models.

In our study, we assign real fitted values to A and B so that the model represents vulnerability discovery for Firefox, Internet Explorer, and Safari. The second Atomic model describes the patch development process (Figure 5.1b). Patch development time is software and vendor dependent. For the web browsers under study, the patch development times are set to those listed in Table 5.1. These times are reported for different browsers in the Symantec report [71].

Patch development continues until the total number of patches equals the number of discovered vulnerabilities. This is done using the input gate "develop." The next element models the deployment phase of the patching process (see Figure 5.1c). If pre-deployment testing is being done on patches, a timed activity moves new patches to tested patches

Table 5.1: Patch development time for different browsers

Web Browser	Patch Development Time
IE 7	5 days
Mozilla 2	5 days
Safari 2	3 days

ready for deployment. We change the testing time to find the optimal testing period. The distribution of testing activity is deterministic.

For a given testing period and faulty fraction (f), the fraction of patches that introduce a new vulnerability after testing is computed as $(f \times [1 - (\Omega(T_T) - \Omega(0))])$. This fraction is assigned to the second case of the "testing" activity which adds a new fault to the system. The first case of the activity is when the vulnerability is discovered and it puts a new token in the "ready_patches" place (Figure 5.1c).

Applying a ready patch takes a small amount of time compared to development of the patch and pre-deployment testing, so it is modeled using an instantaneous activity named "apply_patch."

5.5 Simulation Results and Discussion

This section presents the results from the analytical model and different experiments on the stochastic model. To study the effect of testing on the vulnerability of the system, we first need reliable data for the vulnerability discovery trend. For this, we use the National Vulnerability Database (NVD) and collect the vulnerabilities discovered in the three popular web browsers as well as the date of public disclosure.

For our study, we use Mozilla Firefox 2, Microsoft Internet Explorer 7, and Apple Safari 2 browsers. The reason for choosing these particular versions is that they are not old; at the same time, they are not the most recent versions (i.e. Firefox3 or Safari 3) for which very few vulnerabilities are discovered to this day. The vulnerabilities can be in the core of the browser or in any of its extensions or add-ons.

Then, we fit the AML model to these vulnerabilities by minimizing the mean-squared

error. The cumulative numbers of vulnerabilities (Ω) for the three browsers from the real data as well as the fitted models and the parameter values are shown in Figure 5.2.

We have simulated the stochastic model in Mobius [78] using the parameters from the three web browsers.

Figure 5.3 shows the cumulative number of vulnerabilities discovered obtained by simulating the stochastic model. The quantity plotted in this figure is the marking of (i.e. the number of tokens in) the place holding the discovered vulnerabilities (named "disc_vul" in the model).

The effects of different parameters such as faulty fraction (f), testing time (T_T), and browser dependency are studied in different experiments. In each experiment, the parameter under study is changed and other parameters are set to some fixed value. This does not suggest that those fixed values are typical in any way. It is done to keep other parameters invariant so that we can observe the effect of the parameter under study. Of course, in a real system, the behavior is a function of all these parameters.

To study the effect of faulty patches, the number of open vulnerabilities is plotted versus time for different f values in Figure 5.4. The number of open vulnerabilities is the number of unpatched discovered vulnerabilities plus the number of faults introduced during previous patch deployments. For this experiment, no pre-deployment testing is done and the results are simulated for Firefox 2. Notice three facts from Figure 5.4. First, the number of patches lags behind the number of discovered vulnerabilities. This is due to the time it takes for the vendor to develop new patches. Second, when there is no faulty patch ($f = 0$), the number of open vulnerabilities goes to zero eventually. The non-zero value in the middle is the difference between the number of vulnerabilities and patches. Third, for faulty patches, there are always residual faults that remain in the system.

The next experiment studies the effect of testing on the number of open vulnerabilities. This experiment is done for two f values (0.2 and 0.5) and three testing periods (two weeks, one week, and one day). Internet Explorer 7 is used as the model. The results are shown in Figure 5.5. Observe that for a given time, a specific testing period minimizes the area under the curve from zero to that time representing the total number of open vulnerabilities over that time. This amount of pre-deployment testing is the optimal testing.

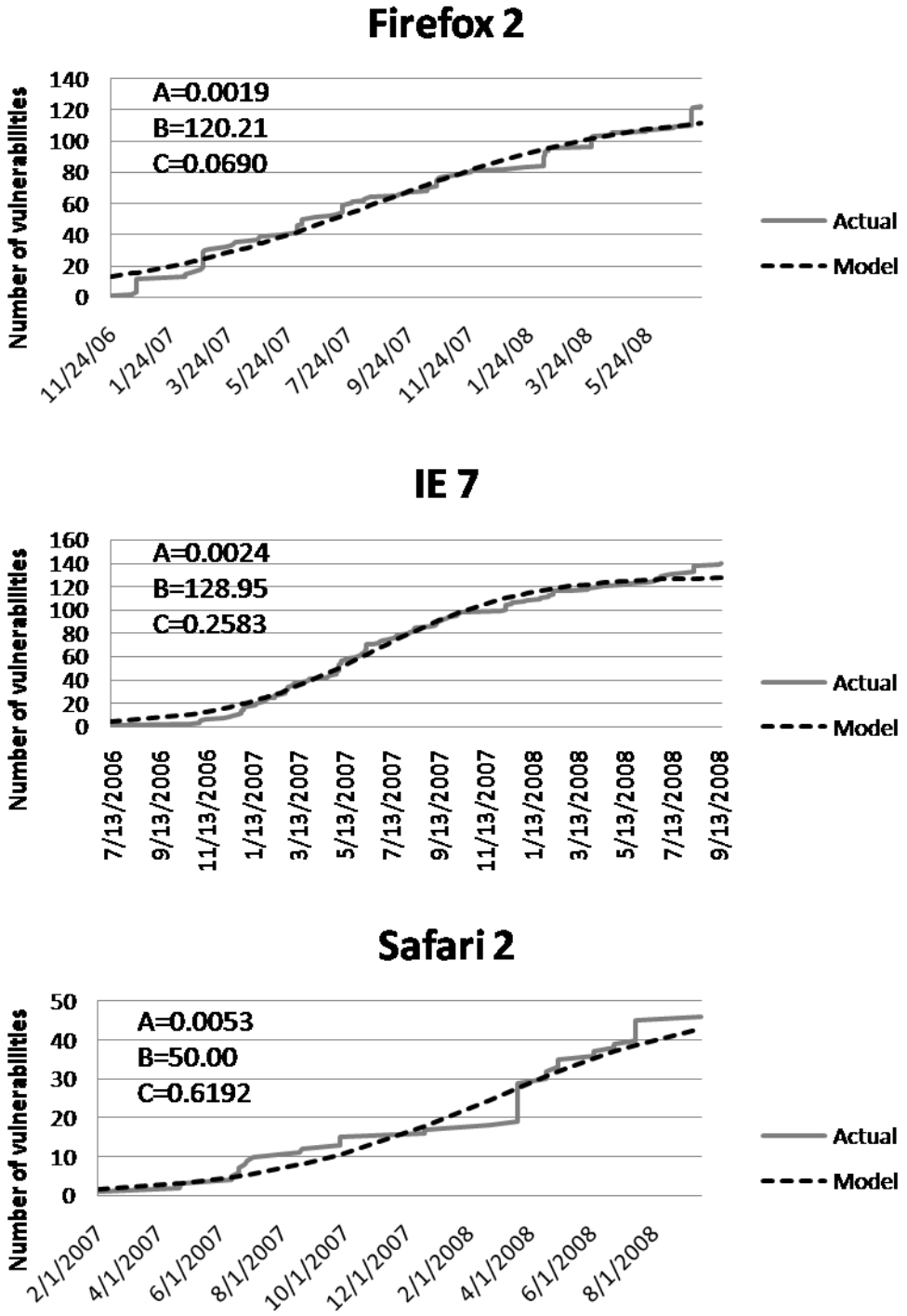

Figure 5.2: Vulnerability discovery for different browsers

The parameters used in the experiment are arbitrary values chosen for demonstration of the effects only. The exact optimal testing period for a given faulty fraction can be obtained

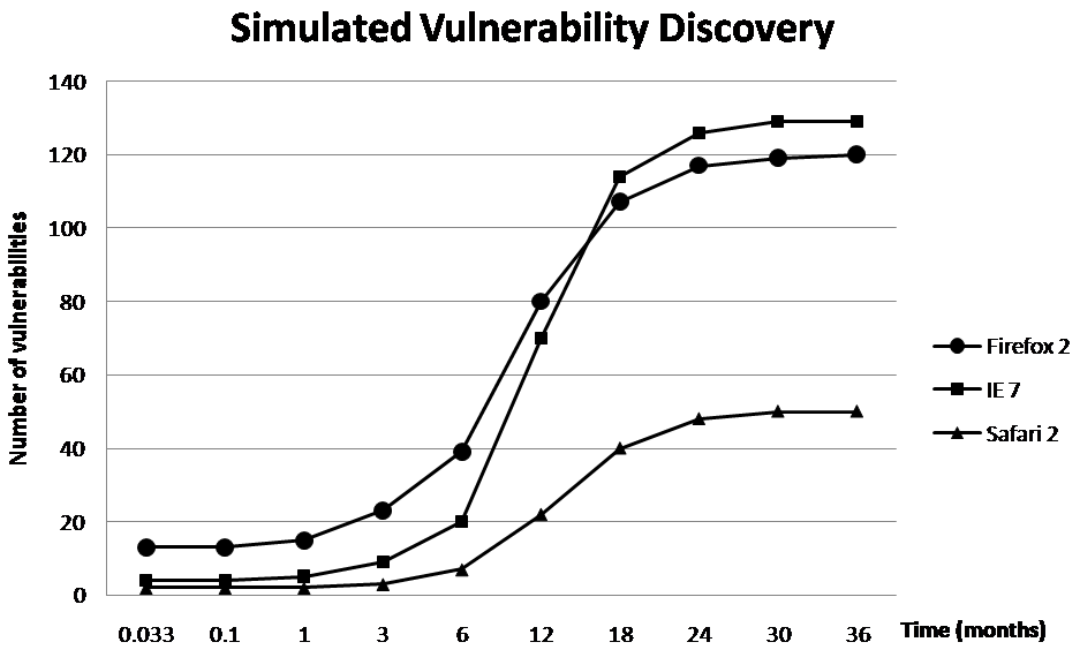

Figure 5.3: Vulnerability discovery from the stochastic model

from Equation 5.10. In the next section we show that the results from the analytical model match those obtained here from simulation.

Intuitively, more testing or smaller faulty fraction would result in fewer residual vulnerabilities in the system. However, interestingly, two weeks of testing with 50% fault result in the same residual vulnerability as one day of testing with 20% fault. This means that in order to compensate for faultier patches, a lot more testing has to be done. We verify this result in the next section.

Finally, the last experiment studies the software dependency of pre-deployment testing. For this experiment, we plot the number of open vulnerabilities at each time for the three web browsers given a faulty fraction of 0.3 and testing periods of two weeks and one day. The results are shown in Figure 5.6. The patch development times for this experiment are five days for Firefox and IE and three days for Safari. These times are taken from the Symantec report [71].

There are two observations to be made for this experiment. First, the evolution of open

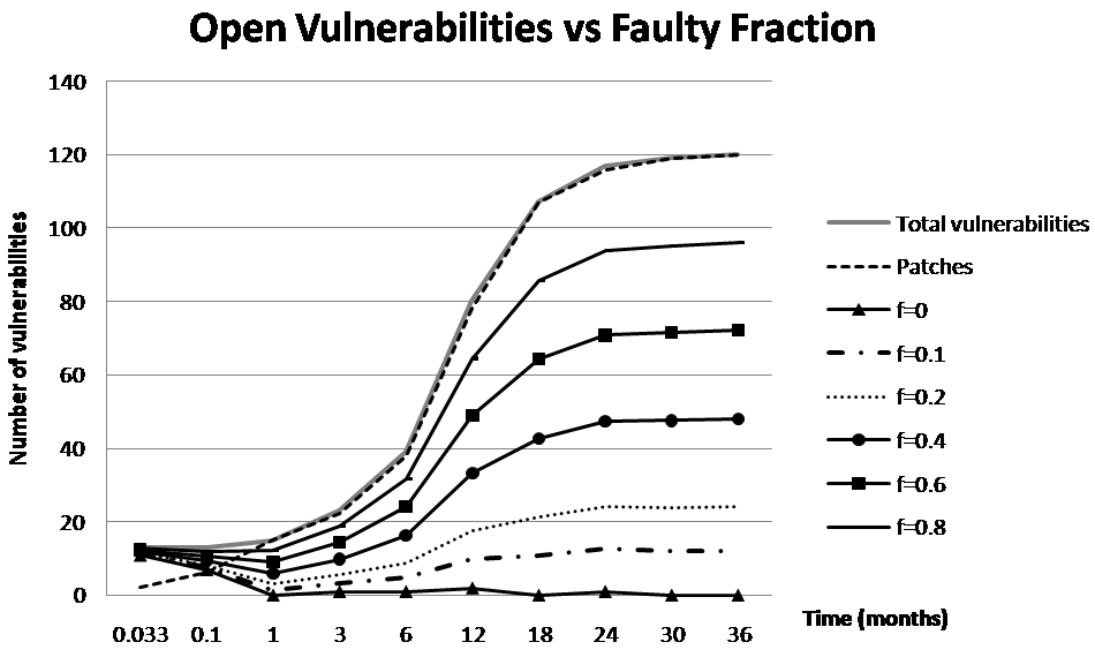

Figure 5.4: The number of open vulnerabilities for various faulty fractions

vulnerabilities over time is highly dependent on the software (in this case the web browser). Second, there is a common trend between different web browsers and testing times. The number of open vulnerabilities is high at the beginning before the vendor gets a chance to develop patches for the initial problems. It shrinks to a minimum between one and six months after the initial phase is passed. It then grows rapidly or peaks at around one year into the process. This sharp growth is because users become more familiar with the software and the rate of vulnerability discovery is very high at this point. Finally, it approaches the terminal after about three years.

5.6 Verification and Validation

In this section, we validate the sub-models used in the analytical and stochastic models in this work. Furthermore, we verify the results obtained by simulating the stochastic model using the analytical model.

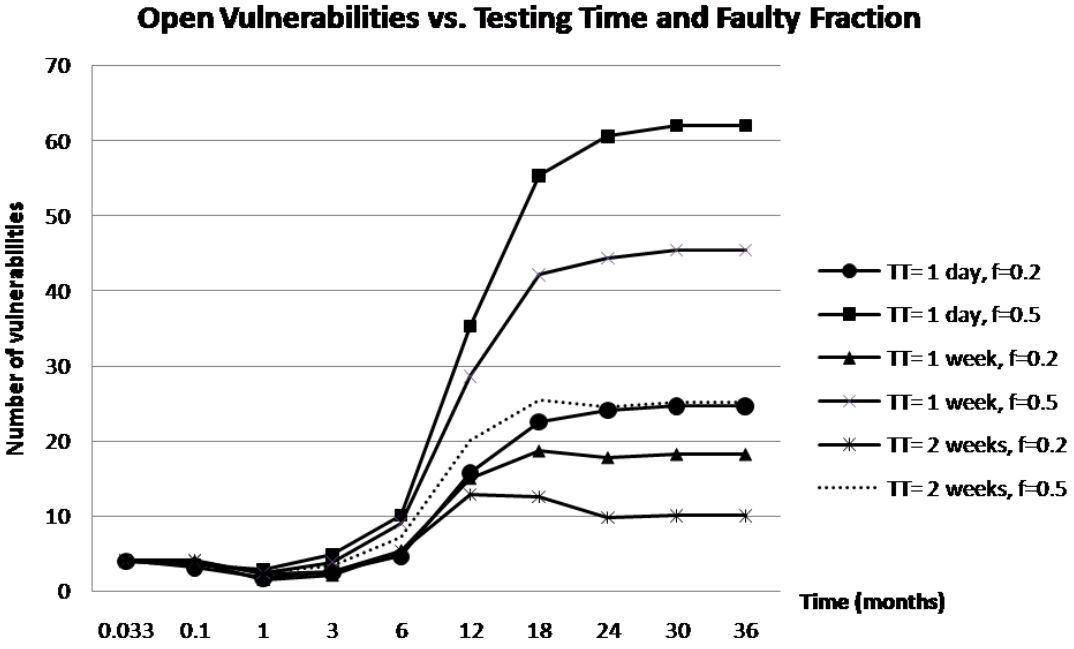

Figure 5.5: The number of open vulnerabilities for various testing times

First, to validate the vulnerability discovery model, we calculate the mean squared error (MSE) of the fitted AML model compared to the actual data from the vulnerability database (Figure 5.2). The MSEs for Firefox, IE, and Mozilla are 27.5, 22.7, and 11.7 respectively. Also the percentages of error for these browsers are 6.94%, 5.50%, and 11.74% respectively. The errors for IE and Firefox show that the model is a good fit for the actual data. For Safari, the error is a little larger because of fewer samples and cumulative public disclosure of vulnerabilities which results in a staircase-like trend.

Next, we validate the stochastic vulnerability discovery model (Figure 5.3) by calculating its error from the actual data. The percentages of error for stochastic models of Firefox, IE, and Safari are 12.0%, 12.8%, and 12.5%. Most of the error is because of the initial error in the model for the first two samples (i.e. one day and three days).

To verify the simulation results, we compare them to those obtained from the analytical model. Equation 5.10 gives the optimal pre-deployment testing time for any period of lifetime. It is important to notice that the optimal time depends on the goal one wants to

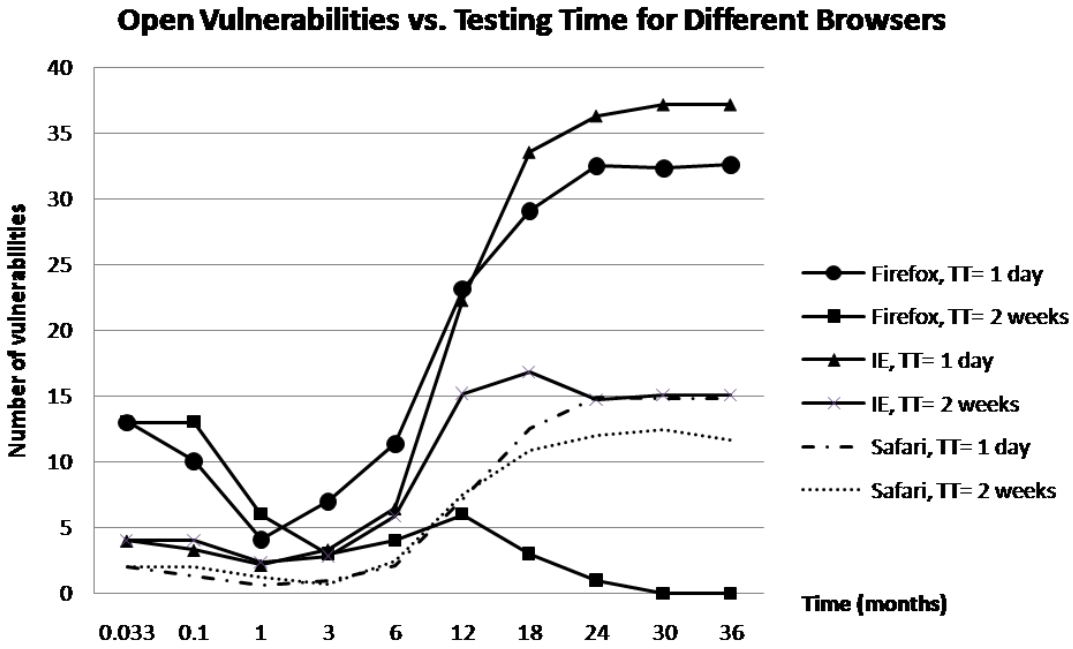

Figure 5.6: The number of open vulnerabilities for the different browsers

achieve. For instance, the goal of minimizing the number of open vulnerabilities over the two-year lifetime of the software results in a different optimal time than that of minimizing the number of open vulnerabilities at a specific time. In the analysis provided here, without loss of generality, assume that the goal is to minimize the number of open vulnerabilities over the two-year lifetime of the software. Since a new version of the browser is introduced after two-years, this can be a meaningful goal.

First, we solve the analytical model for the second simulation experiment. The model is solved for $f = 0.2$ and the three testing times (one day, one week, and two weeks). By solving Equation 5.9, the total numbers of open vulnerabilities over two years for the three experiments are 317.3, 262.2, and 195.9 respectively. These values do not refer to real vulnerabilities; rather, they are the summation of all possible windows of exposure. The values from the analytical model and the stochastic model agree in what they suggest. That is, two weeks of testing is better than one week or one day. The reason becomes apparent when we explicitly solve the analytical model for IE 7 to find the optimal testing time. By

solving Equation 5.10, the optimal testing time is 0.7845 months or about 23 days (total open vulnerabilities over two years=116.8). That is why two weeks of testing achieves better results than one week or one day in the simulation. In fact, the analytical model gives insight into the results obtained in the simulation section.

In addition, we verify the result found in the second experiment that 20% fault with one day of testing has the same number of residual vulnerabilities as 50% fault with two weeks of testing. From the analytical model, and by substituting the actual values for IE 7, the value of $f \times [1 - (\Omega(T_T) - \Omega(0))]$ is the same for f=0.2 and T_T=0.033 (months) as f=0.5 and T_T=0.5 (months). This quantity is what we called α and shows the residual vulnerability. The analytical value obtained for α in both cases is 24.2 which again is the same value from the simulation in Figure 5.5.

Finally, we obtain the optimal testing time for all of the web browsers for the last simulation experiment. The optimal T_T for IE, Firefox, and Safari (f=0.3) are 0.784 month (=23 days), 0.353 month (=10 days), and 1.928 month (=57 days) respectively. This verifies why the residual vulnerability after two weeks of testing for Firefox is zero, but for IE and Safari it is not (Figure 5.6). Two weeks of testing is greater than the optimal testing time of Firefox, but smaller than those of IE or Safari. Hence, α is zero for Firefox while it is non-zero for IE and Safari.

It is important to realize what the verifications in this section suggest. They do not prove that the analytical model and the models used in the simulation are correct. On the other hand, they show that these models, capturing medium-level details of the system, indicate the same tradeoff and they match in what they suggest. It is possible to come up with more sophisticated models to capture low level details of a patching system. However, the fundamental tradeoff always exists in the system: pre-deployment testing results in more exposure and fewer new vulnerabilities.

By comparing the results with the actual real-world vulnerability data for different browsers, we have shown that these medium level models can capture the reality with a margin of about 12% error.

6 TrustGraph: Trusted Graphics Subsystem

6.1 Introduction

This chapter describes the design and implementation of TrustGraph [79], a trusted graphics subsystem for high assurance systems. We first provide 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 knowledge of graphics systems to understand the rest of the chapter.

6.1.1 Contributions

The contributions of this chapter are as follows:

- We enumerate and describe different classes of attacks possible using the API of a graphics subsystem.
- We describe the 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.
- Finally, we perform static analysis on the actual implementation of TrustGraph using compiler-based techniques, and we identify and correct several coding flaws.

6.2 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 hereafter refer to simply as the *interface*. 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 graphics subsystems include the X Window System [80] for Linux-like operating systems, DirectFB for Linux and embedded systems, and Quartz [81] 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 [7].

Window: 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 for an application window in the middle, and yet another for a video playing on top.

Surface: 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: A buffer that holds all of the input events for a window (e.g. keystrokes 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

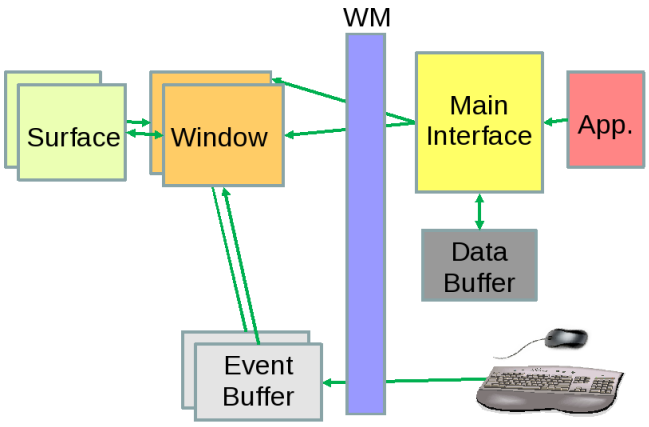

Figure 6.1: Architecture of a graphics subsystem

methods. The window manager is also responsible for redirecting the input events to the appropriate window.

In this chapter, 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 6.1 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 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 6.4.3. 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 operates 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 ($f : S \rightarrow S$). For 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. $GS = \langle R \mid M \mid OP \mid D \mid WM \rangle$.

6.3 Threat Model

The threat model used in this work 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; however, 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 [82] 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. Global operations such as copying/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 make graphics systems such as X inherently bad choices for trusted graphics. X was developed at the time when computer graphics had low color depth and there was no hardware acceleration [83]. 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 these resources can be used as communication channels not regulated by the security policy. In short, X is too large and complex for secure graphics.

6.4 Design

6.4.1 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 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 mediation is either not done or not completed.

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.

6.4.2 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.

6.4.3 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).

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 follows:

- 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 Data Buffer: This method is called from the main interface to create a data buffer. It propagates the security tag of the main interface to the data buffer.

- Create Window: It is called from the specific layer interface of the main interface. It creates a window and propagates the main interface security tag to it.

- Create Surface: This method is called from the main interface to create a surface. It propagates the main interface label to the surface.

- Create Event Buffer: It is called from the window or main interface. It propagates the corresponding security tag to the event buffer.

- Get Window: This method is called from the specific layer interface of the main interface to acquire an already existing window. If the window has a higher security level than the main interface, it can result in information leakage from a higher security level into a lower one. Consequently, in such a situation the access is denied. A main interface can only read from a window with lower or equal security level. Hence, in such a condition, the window acquisition is granted to the main interface while the security level of the window is elevated to that of the main interface. This is done because an application writes its graphical data which has the same security level as the application to its window. The elevation of security level prevents leakages in future window acquisitions.

- Get Surface: The method is called from a window interface to acquire an already existing surface. Similar to getting a window, if the window has a lower security level than the surface, the access is denied. Otherwise, the window is granted access and the surface security level is elevated to that of the window.

- Get Event Buffer: Similar to getting a surface, this method is called from the window interface to acquire an existing event buffer. If the event buffer has a higher security

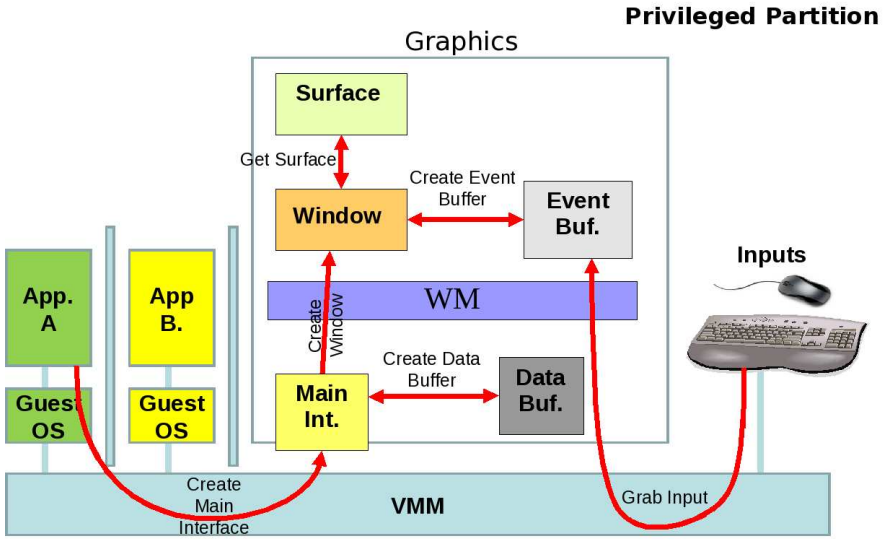

Figure 6.2: Label flow in TrustGraph

level than the window, the access is denied. Otherwise, the window is granted access and the event buffer security level is elevated to that of the window.

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 [82], a label dominates another one if it has a higher level and its set of categories is a super-set 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 6.2. 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 tag from the VMM (or the operating system in a non-virtualized model). This is done through a small piece of code called "membrane." Membrane is responsible for receiving the application security tag from the VMM and delivering it to TrustGraph when the main interface is being created.

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 windows 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 a 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 follows. "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 a 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.

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. Respectively, these methods redirect all keyboard events, specific key events, mouse/joystick events, and unselected key events to the application.

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 keystroke 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.

6.4.4 Secure Operations

The global operations are implemented in a graphics subsystem to facilitate the usage and augment the system with additional functionalities. However, they 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 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 window that does not have a dominated label in

the screenshot. 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.

6.4.5 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 6.3, 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. 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 covert channel attacks that exploit graphical methods in Section 6.6.4.

6.5 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. Table 6.1 lists the label propagation and checking for the methods in TrustGraph. 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.

Table 6.1: Label-propagating methods in TrustGraph

CreateMainInterface	`Main interface PL = Application label;`
CreateDataBuffer	`Data buffer PL = Main interface PL;`
CreateWindow	`Window PL = Window DL = Main interface PL;`
CreateSurface	`Surface PL = Surface DL = Main interface PL;`
CreateEventBuffer	`Event buffer PL = Window PL;` `Event buffer DL = Window DL;`

GetWindow	`If(Main interface PL DOM Window PL)`
	`  Window PL = Window DL = Main interface PL;`
	`else`
	`  deny;`

GetSurface	`If(Window DL DOM Surface PL)`
	`  Surface PL = Window PL;`
	`  Surface DL = Window DL;`
	`else`
	`  deny;`

GetEventBuffer	`If(Window DL DOM Event buffer PL)`
	`  Event buffer PL = Window PL;`
	`  Event buffer DL = Window DL;`
	`else`
	`  deny;`

SetWindowLevel	`If(Main interface PL DOM L)`
	`  Window  DL=L;`
	`else`
	`  deny;`

6.5.1 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 [84].

6.5.2 End-to-End Implementation

As a proof of concept, we have implemented an end-to-end virtualized architecture using TrustGraph. We have used Xen [5] and sHype [85] 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 is sent to dom0 via hyper calls (see Figure 6.2). 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.

6.6 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 to 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.

6.6.1 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.

6.6.2 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 6.3. In fact, the reader can easily design and implement similar attacks 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.

6.6.3 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 check the correctness of those parts.

ACL2

ACL2 (A Computational Logic for Applicative Common Lisp) [86] 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 [87].

Both the ACL2 language and its implementation are built on the side-effect free version of Common Lisp [88].

In common Lisp everything including the code and the data is a list. Lists hold data such as integers, lists, fractions, or characters as a list. 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, Table 6.2 shows some ACL2 scripts with their equivalent pseudo code meaning.

Table 6.2: ACL2 scripts and their pseudo code meanings

ACL2 Scripts	Psedo Code
`(* 3 4)`	`3 * 4`
`(* (+ n 1) n)`	`n*(n+1)`
`(if  x  y  z)`	`if x then y else z`
`(>=  5  6)`	`5 >= 6 ?`
`(cons x  y)`	`(x y)`
`(car (x y z))`	`first element of (x y z) = x`
`(cdr (x y z))`	`the rest of (x y z) = (y z)`
`(defun minus_one (n) (- n 1))`	`function minus_one (n) {return n-1}`

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 the ACL2 tool tries to prove about that operation or model. To clarify the point, consider the following example from the ACL2 book [86]. Assume that we want to prove that the length of the concatenation of two lists is equal to the sum of their lengths. First, we define the concatenation function. The function is defined recursively.

```
(defun my-app (x y)
  (if (atom x)
      y
      (cons (car x) (my-app (cdr x) y))))
```

Then we express the desired theorem in ACL2 syntax:

```
(defthm my-app-length
  (equal (len (my-app x y))
         (+ (len x) (len y))))
```

The ACL2 theorem prover tries to prove this theorem about the concatenation operation. It uses some basic axioms that it has in its libraries and proves the theorem 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. The 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.

As another example, the following theorem shows that the summation of integers from 1 to n equals n*(n+1)/2:

```
(defun sum(n)
   (if (zp n)
       0
       (+  n sum(-n  1))  ))

(defthm algebra1
   (implies (natp n)
            (equal  (sum n)
                    (* n (+ n 1) 1/2) )))
```

ACL2 has been used for a variety of theorem-proving and model-checking problems from the correctness of the write-invalidate cache system to the Dijkstra's shortest path algorithm. For a list of problems solved using ACL2, refer to its official web site [89].

Detailed description of how ACL2 works or how to prove theorems or check models with ACL2 is beyond the scope of this report. The reader may refer to the ACL2 book [86] for more information.

Formal Verification of TrustGraph

To check the correctness of the logic implemented in the TrustGraph window manager and the label flows, we have performed formal verification 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 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 window level. Table 6.3 shows the ACL2 scripts for these four methods. Each method behaves as described in Table 6.1. 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 are propagated to the resource.

To prove the correctness of label flow, we define four ACL2 theorems as shown in Table 6.4. 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 considers 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. Table 6.5 shows the scripts that model window insertion, order checking, and the theorem to prove that inserting windows preserves the correct ordering. Similar scripts are used for restacking windows.

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

Table 6.3: TrustGraph label flow scripts

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

```
(defun get_surface (surface win)
   (if (dom (car (cdr win)) (car surface))
       (cons (car win) (cdr win))
        nil))
```

```
(defun attach_event_buffer (event_buffer win)
   (if (dom (car (cdr win)) (car event_buffer))
       (cons (car win) (cdr win))
        nil))
```

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

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

```
(defun subset (lista listb)
   (if (consp lista)
       (AND (member (car lista) listb)
            (subset (cdr lista) listb))
        T))
```

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 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

Table 6.4: Label flow security theorems

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

```
(defthm no_leak3
   (implies
      (AND
        (consp win)
        (consp event_buffer)
        (NOT (dom (car (cdr (set_win_label win int_PL label))) (car event_buffer))))
      (= (attach_event_buffer event_buffer (set_win_label win int_PL label))
         nil)))
```

```
(defthm no_leak4
   (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 (get_win win int_PL) int_PL label))
         nil)))
```

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. For 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 compartment 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

Table 6.5: Window ordering model

```
(defun insert_win (x y)
   (if (or (endp y)
           (<= x (car y)))
       (cons x y)
       (cons (car y) (insert_win x (cdr y)))))
```

```
(defun my_inorder (y)
(if (AND (consp y)
         (not (equal (cadr y) nil)))
    (AND (<= (car y) (cadr y))
         (my_inorder (cdr y)) )
    t))
```

```
(defthm my_thrm
   (implies
      (my_inorder y)
      (my_inorder (insert_win x y)) ))
```

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 Tables 6.3, 6.4, and 6.5 are the corrected versions.

6.6.4 Covert Channel Analysis

Covert channel attacks [90] can pose a threat to the security of high-assurance systems by using the shared resources in a way not intended in their design, in order 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 [91] and its successor the Common Criteria [92] 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 10 MB file over the channel. The total transfer time was about 10 min and 30 sec, 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 [93]. 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 [94]. 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.

6.6.5 Compiler-Based Verification

The model verified using the formal method techniques is different from the actual implementation of TrustGraph. The model represents the high level workings of TrustGraph using ACL2 while the actual implementation is much more detailed and is done in C. There have been some attempts [86] to directly compile a formal model described in ACL2 to a C implementation for small modules. However, for an implementation as complex as that of TrustGraph, soon the system becomes too complex and large to directly build by compiling the model.

In order to narrow down the gap between the formal model and the actual implementation, we use a more feasible option than direct compilation. We check the C implementation by performing compiler-based static analysis on the code. If the code passes the checks or the security violations are corrected, the code can become much closer to achieve control flow integrity (CFI). Having control flow integrity means that the formal model correctly represents the system (at least for the abstractions captured by the model) and one cannot change the behavior of the code by exploiting vulnerabilities.

ROSE Compiler Infrastructure

To perform compiler-based verification we use the ROSE compiler infrastructure [95]. ROSE is a tool for performing source-to-source translation. ROSE can be used to build tools for analyzing source codes from large scale applications (millions of lines). Such tools can be useful for many purposes:

- automated analysis and/or modification of source code
- instrumentation
- data extraction
- building domain-specific tools

The intermediate representation (IR) used by ROSE is called SAGE III which is detailed enough to build an abstract syntax tree (AST) that is well suited to source-to-source trans-

lation; i.e., ROSE does not lose any information about the structure of the original source code. The mid-end contains an evolving set of analyses and optimizations. The Edison Design Group (EDG) front-end is used to parse C and C++ applications. ROSE converts the intermediate representations (IRs) produced by the front-ends into abstract syntax trees.

Compass Tool

To check for security or coding violations, we use a tool named Compass [96]. It is currently distributed as part of ROSE, and represents one of many tools that can be built using the ROSE open compiler infrastructure. Compass runs on top of ROSE and it uses ROSE to convert a source code to an IR. It then analyzes the code for various security or coding violations using the features provided by ROSE which include Abstract Syntax Tree (AST), control flow graph, system dependence graph, call graph, class hierarchy graph, etc.

Each security or coding violation is formulated as a checker in Compass. Users can select the checkers to run on a piece of code or they can define their own checkers.

Compass checks for more than 100 different violation types which include the following major categories:

- Input Validation: forbidden functions, const string, buffer overflow, etc.
- Memory Bugs: double free, no free, NULL dereference, delete, etc.
- Complexity Violations: deep nesting, computations, cyclomatic complexity (CC), etc.
- Race Conditions: same handler for many signals, etc.

The complete list of Compass checkers is given in Appendix A. Some of the violations can directly result in vulnerability of the code (e.g. memory bugs or input validation) while others are violations of coding best practices and may or may not result in a vulnerability (e.g. too much complexity).

After checking TrustGraph, Compass found a long list of violations in its code. Table 6.6 lists three samples of the violations. The first line shows a bitwise operation without an explicit bit mask. This can result in signed/unsigned confusion or overflow of the value,

which may result in control flow violation. The second example shows the use of ternary operator (condition ? expression : expression). The ternary operator is not appropriate for high-integrity C programming [97]. The reason is that if the expression involves an object, the ternary operator behaves differently than an explicit test. Finally, the third example illustrates a piece of memory that is allocated but is not "freed" in this scope.

The complete list of code violations found in TrustGraph can be found in Appendix B. By correcting all these violations, the gap between the low level implementation of TrustGraph and its formal model narrows substantially.

Table 6.6: Sample violations in TrustGraph code found using static analysis

Source	Violating Code
palette.c:172.47-61	`palette->entries[i].r = lookup3to8[ (i & 0xE0) >> 5 ];`

wm.c:901.45-80	`return wm_local->funcs->RestackWindow(`
	`  window, wm_local->data, window->window_data, relative,`
	`  relative ? relative->window_data : NULL, relation );`

core.c:828.6-49	`CoreCleanup *`
	`dfb_core_cleanup_add( CoreDFB *core, CoreCleanupFunc  func, void *data, bool emergency)`
	`{`
	`  CoreCleanup *cleanup;`
	`  cleanup = D_CALLOC( 1, sizeof(CoreCleanup) );`

6.7 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 [98] and VMM-based security kernel for VAX architecture [99] represent two of the earlier attempts.

NSA's NetTop [100] and MILS [101] and IBM's sHype [85] are three of the more recent VMM-based high assurance systems. Karger [102] studies the requirements of the MLS systems and discusses their implications for the design of VMMs. Terra [103] and Secvisor [104] 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. [105] use formal techniques to verify the Linux security

kernel.

Karger and Safford [3] describe the I/O virtualization complexities and study the performance and security tradeoffs of different I/O models. AMD [106] and Intel [107] support the Input Output Memory Management Unit (IOMMU) approach for assigning I/O devices to virtual machines. However, this approach is not suitable for a graphics system which is inherently shared between the virtual machines.

Woodward [108] 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 [109] study the security problems of X.

There have been different efforts to build trusted X. Epstein [110, 111, 112] and Woodward [108] describe different trusted X implementations. Picciotto [113] presents two approaches for implementing trusted cut and paste operation in X. Another work by Picciotto and Epstein [114] surveys the architectures and security policies implemented by the trusted X implementations. Finally, the work by Feske and Helmuth [115] is another recent effort at 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 [116]. However, simply dividing clients into "trusted" and "untrusted" is too coarse-grained. A more flexible policy model is implemented by extending the SE-Linux policy [117] to X using XACE hooks [118]. Nevertheless, this implementation only mediates known channels under X and does not provide any type of assurance [6]. 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 [119] describes how design flaws in the Win32 API can be exploited to escalate privilege.

7 Simplified Graphics Subsystem

7.1 Introduction

This chapter describes a simplified design of a graphics subsystem. We simplify a graphics subsystem by keeping only the necessary functionalities and eliminating all the unnecessary complexities. The simplified design allows us to perform a more complete verification of the system which can potentially lead to fewer vulnerabilities. In fact, the simplified design enables us to build a complete formal model of the system and verify it using formal methods.

7.1.1 Contributions

The contributions of this chapter are as follows:

- We precisely describe and design a simplified graphics subsystem.
- An analysis is performed on the necessary functionalities in the core of the simplified design and the features that can be eliminated. A solution is proposed to support the eliminated functionalities.
- The implementations of the graphical resources are presented.
- The simplified graphics subsystem is then formally modeled and a number of security properties are proved about the system. We present the steps involved in formal modeling of the system and theorem development. The steps described can be used to model similar systems at the abstraction level presented.

The related works for the simplified graphics subsystem are the same as those presented for TrustGraph (Section 6.7.) To avoid repetition, we do not present them in this chapter.

7.2 Graphics Resources, Methods, and Operations

The simplified design has four main graphics resources. They have the same definitions as in TrustGraph, but for the sake of completeness we review their functionalities here.

Window: A window (W) is the visual area that displays the graphical information of an application. A window is always assigned a surface.

Surface: A surface (S) is a pixel buffer that holds the pixel data for a window. The drawing functions of a graphics subsystem operate directly on a surface in order to draw an object.

Event Buffer: An event buffer (EB) holds all of the input events for a window (e.g. keystrokes or mouse events).

Data Buffer: A data buffer (DB) holds non-vector graphical data (images or videos) to be displayed in a window.

All windows, surfaces, event buffers, and data buffers are referred to as the graphical resources (R). As in TrustGraph, each of the graphical resources is tagged with two security labels. A label is a standard MLS label with a sensitivity level and a set of categories. Each graphical resource has a permanent label and a declassification label. All of the interactions between an application and the graphics subsystem are done through the main interface (I). The main interface automatically gets the same security label as the application through the VMM. Moreover, the main interface only has a permanent label.

Graphical resources are created or dynamically acquired using the graphical methods (M). A graphical method can be viewed as a function either from the main interface to the graphical resources ($M : I \rightarrow R$) or from the graphical resources to the graphical resources ($M : R \rightarrow R$). In general, the graphical methods are $M : \{R, I\} \rightarrow R$. A complete list of graphical methods in the simplified graphic subsystem and their respective functions is given in Table 7.1.

An important part of a graphics subsystem is the drawing (blitting) functions (D). These functions are used by an application to draw on an already acquired (or created) surface.

Table 7.1: Simplified graphical methods

Create Window	$f : I \rightarrow W$
Get Window	$f : I \rightarrow W$
Create Surface	$f : W \rightarrow S$
Get Surface	$f : W \rightarrow S$
Create Event Buffer	$f : W \rightarrow EB$
Attach Event Buffer	$f : W \rightarrow EB$
Create Data Buffer	$f : W \rightarrow DB$
Get Data Buffer	$f : W \rightarrow DB$

Aside from the capacity of the memory, the drawing functions are among the main differences of a high-end and a low-end video card. A high-end video card supports sophisticated drawing functions in the hardware (e.g. 3D shapes and shading) while with a low-end video card only simple drawing functions (e.g. lines or triangles) are handled in the hardware and more sophisticated functions must be emulated in software.

There are two major types of drawing: geometric (vector) and image/video. Geometric drawing is done by modifying the pixel values of a surface using an algorithm to add an object (e.g. line, circle, rectangle, etc.) to the window. It is a function that maps a surface to another surface $D : S \rightarrow S$ (with different pixel values). Image/video drawing is done by loading a data buffer to a surface $D : DB \rightarrow S$. Hence, drawing functions are mappings from surfaces or data buffers to surfaces $D : \{S, DB\} \rightarrow S$.

A graphics subsystem also includes a set of global operations (OP) that work with a set of resources. The most common operations include taking screenshots (usually from many windows), copy-pasting (between two windows), and drag-and-dropping (between two windows). The simplified graphics subsystem supports copy-pasting and screenshots. They are secured as described in Section 6.4.4 for TrustGraph.

The relationships among graphical resources, drawing functions, graphical methods, and graphical operations are illustrated in Figure 7.1.

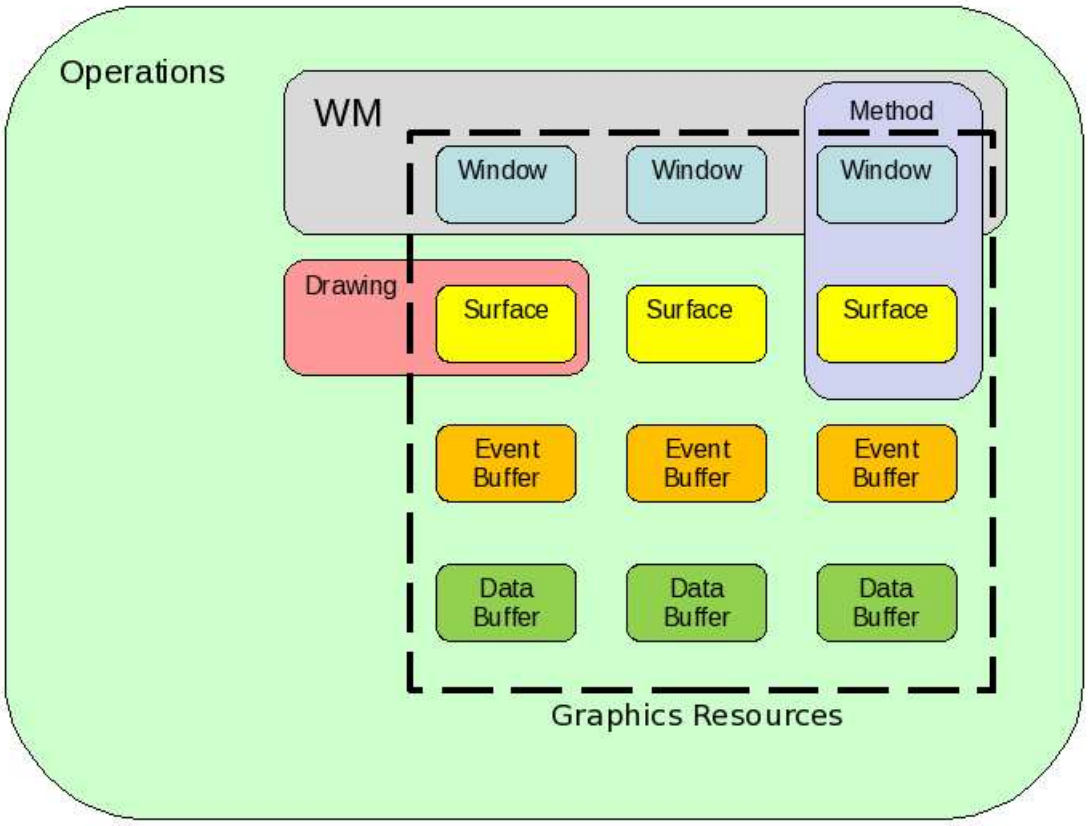

Figure 7.1: A simplified graphics subsystem

7.3 Necessities

When designing a simplified graphics subsystem, we have to include only the necessary functionalities to keep the design as simple and small as possible. Hence, we should understand the necessities of the features implemented, especially those that have strong security implications. Drawing functions manipulate already acquired surfaces, so they are restricted to the security compartment of their applications. Here we study the other graphics features that work across different security compartments.

7.3.1 Dynamic Graphical Methods

The dynamic graphical methods in a graphics subsystem have strong security implications. For instance, if an application cannot dynamically acquire the surface of another application

(using the GetSurface method), there would be no need to label the surfaces and check the security policy on each surface method. The same is true of windows, event buffers, and data buffers. Thus, it is important to justify the necessity of dynamic graphical methods.

There are two main reasons to have dynamic methods in a graphics subsystem:

1. Applications often create windows on the fly. When a browser opens a new pop-up or when an application opens a new dialog, new windows have to be created dynamically. Removing dynamic methods makes the graphics subsystem unusable.

2. Image/video manipulation programs use dynamic methods to work with graphical data. For example, when an effect is added to an image/video or when parts of one image are added to another image, the program dynamically acquires the surface of a window. Removing dynamic methods breaks these programs.

7.3.2 Enumerating Windows

The graphics subsystems allow applications to enumerate the windows on the display. This is the reason that an application can acquire the resources created by another application. Being able to enumerate windows is necessary for three main reasons.

1. Task managers use windows enumeration to find out what windows are open on the display.

2. Facilities such as "Alt-Tab" (also called Task Switcher or Flip) sometimes use windows enumeration to switch between different windows.

3. Some crash-recovery applications use windows enumeration to find the windows associated with non-responsive applications.

Disabling windows enumeration breaks the above mentioned applications.

7.3.3 Graphical Operations

The main reason to have graphical operations in a system is ease of use. Copy-paste, drag-and-drop, and screenshots add flexibility to a graphics subsystem if done securely.

Since securing the graphical operations is relatively straightforward, as described in the previous chapter, we keep them in the simplified design to preserve the system's usability and flexibility.

7.4 Eliminated Features and Compatibility

Some of the features that exist in TrustGraph and DirectFB have been eliminated in the simplified graphics subsystem to keep the system as simple as possible. We first provide the complete list of features eliminated from TrustGraph and then explain how the functionalities that they provide can be supported.

7.4.1 Eliminated Features

The first feature eliminated from the simplified design is the font interface. This interface allows an application to define a custom font face and size, and write text directly to a surface. The interface defines a number of functions to manipulate a font, set or retrieve a string of text, and define the encoding of the text. The interface gets these settings from the application, determines the appropriate pixel data (for the specific text and font), and puts the pixel data on a surface.

The second feature eliminated is the palette interface. Under TrustGraph, the palette interface allows an application to define many custom colors at a time and retrieve them using an index. This feature facilitates drawing functions, especially if the application contains a large number of them. Note that without a palette, an application can still define a single color (by RGB values) as used by the drawing functions.

Finally, the last feature removed from TrustGraph is the image/video provider interface. The interface allows an application to load/access different image or video formats into a data buffer. Removing this feature essentially means that all image and video codecs can also be removed for the simplified design.

7.4.2 Compatibility

In order to preserve the functionalities that the eliminated features provide, they must be pushed into a library on top of the simplified graphics subsystem. The library performs the rendering required for supporting fonts, palette, and different image/video formats and loading the pixel values directly into the surface of the application. Note that since the surface is already acquired by the application, the library cannot violate the security policy of the graphics subsystem. Also, pushing these functionalities to an additional library further simplifies the core of the graphics subsystem, which facilitates formal evaluation of the system.

Furthermore, applications often use additional libraries (e.g. GDK or GTK+) to work with widgets. The widget libraries provide the functionalities of the eliminated features (fonts, palettes, and image/video providers). To support X applications on top of TrustGraph, we use the XDrirectFB library. This library also supports the eliminated features.

7.5 Implementation

We describe the implementation of the main graphical entities in this section. The implementation is done in C and it directly corresponds to the formal model developed in the next section.

7.5.1 Window

A window has five major components: surface, bound, alpha, permanent label, and declassification label. The "surface" is a pointer to the surface holding the pixel values of the window. The "bound" identifies the position and size of the window. A window in general can have any shape (e.g. a polygon). In the simplified graphics subsystem, a window can only be rectangular. The third property, alpha, specifies the transparency value of the window. Finally, the fourth and fifth properties specify the permanent and declassification labels of a window. Table 7.2 illustrates the implementation of a window.

Table 7.2: Implementation of a window

```
struct Window {
     WindowID           id;
     Surface            *surface;
     Rectangle          bound;
     int                alpha;
     MLSLabel           permanent;
     MLSLabel           declassification;
 };
```

7.5.2 Surface

A surface has four major components: pixel buffer, configuration, permanent label, and declassification label. The "pixel buffer" is a pointer to a two-dimensional array holding the pixel values. The "configuration" holds the size of the surface and the pixel format used (the number of bits and YUV versus RGB representations). The security labels are similar to those defined for a window. Table 7.3 illustrates the implementation of a surface. The drawing functions directly modify the pixel buffer of a surface to draw shapes.

Table 7.3: Implementation of a surface

```
struct Surface {
     SurfaceID          id;
     PixelBuffer        *buffer;
     SurfaceConfig      config;
     MLSLabel           permanent;
     MLSLabel           declassification;
 };
```

7.5.3 Event Buffer and Data Buffer

A event buffer has three major components: event list, permanent label, and declassification label. The event list is a pointer to a linked list containing the input events. A data buffer has four major components: buffer, length, permanent label, and declassification label. The buffer holds the image/video data loaded into the data buffer. The length specifies the size of the buffer. The security labels are similar to those defined for a window. Table 7.4 illustrates the implementation of a data buffer and an event buffer.

Table 7.4: Implementation of a data buffer and an event buffer

```
struct EventBuffer {
     EventBufferID       id;
     EventList           *events;
     MLSLabel            permanent;
     MLSLabel            declassification;
 };
```

```
struct DataBuffer {
     DataBufferID        id;
     const void          *buffer;
     unsigned int        length;
     MLSLabel            permanent;
     MLSLabel            declassification;
 };
```

We have implemented the main features of the simplified graphics subsystem including the resources and the graphical methods. Additional features such as complex drawing functions and global operations can be implemented by reusing and simplifying the implementation of similar features in DirectFB. Also, DirectFB includes a small default window manager that can be reused in the simplified system.

7.6 Formal Modeling Steps

Having a simplified design allows us to formally verify the entire "model" of the graphics system. We now describe how the simplified graphics system is formally modeled and verified step by step. The process is described in three main steps: modeling the resources (the entities), modeling the methods (the functions on the resources), and writing the theorems (the desired properties to be proved about the system). Using these steps, one can model not only a graphics subsystem, but also similar systems at the given abstraction level.

7.6.1 Modeling Resources

Each resource in the simplified graphics subsystem (e.g. window, surface, etc.) has a number of properties. These properties are described in the data structure of the entity as different variables. For example, a window has an ID, a pointer to its surface, a rectangular boundary,

an alpha value, and two security labels. The window properties are described as variables in a C structure that describes a window. In order to formally model a window, all properties of a window must be described in the formal language. Here, the formal model is developed using ACL2. In ACL2, all codes and data are expressed using lists (a.k.a. cons). Hence, our models are also described in lists. Each resource is modeled as a list with the properties of that resource constituting the elements of the list. For instance, a window is a list with the following elements: ID, surface, boundary, etc. Since ACL2 does not restrict the type of the elements, they can have appropriate types as required by the model. In our window example, the ID element of the window is an integer, the surface is a list describing the surface, the boundary is a list of four coordinates, etc. The window implementation and model are illustrated side-by-side in Table 7.5. The model states that the first element of a window list is its ID, the second element is its surface and so on.

Table 7.5: Modeling a window

Implementation	ACL2 Model

`struct Window {`
`  WindowID  id;`
`  Surface  *surface;`
`  Rectangle  bound;`
`  int  alpha;`
`  MLSLabel  permanent;`
`  MLSLabel  declassification;`
`};`
	`(`
	`  id`
	`  surface`
	`  bound`
	`  alpha`
	`  permanent_label`
	`  declassification_label`
	`)`

A similar process is repeated to model a surface in Table 7.6. The complete model of the graphical resources is described in the next section.

Table 7.6: Modeling a surface

Implementation	ACL2 Model

`struct Surface {`
`  SurfaceID  id;`
`  PixelBuffer  *buffer;`
`  SurfaceConfig  config;`
`  MLSLabel  permanent;`
`  MLSLabel  declassification;`
`};`
	`(`
	`  id`
	`  buffer`
	`  config`
	`  permanent_label`
	`  declassification_label`
	`)`

The security labels of the graphical resources are implemented as C structures too. Each

label consists of an MLS level and a set of, at maximum, five categories. A security label is again modeled as a list with the first element being the level and the next five elements being the categories (Table 7.7). Note that each security label in the formal model of the resources (Tables 7.5 and 7.6) is in fact a list as illustrated in Table 7.7.

Table 7.7: Modeling a security label

Implementation	ACL2 Model

`struct MLSLabel {`
`  unsigned char level;`
`  unsigned char cat1;`
`  unsigned char cat2;`
`  unsigned char cat3;`
`  unsigned char cat4;`
`  unsigned char cat5;`
`};`
	`(`
	`  level`
	`  cat1`
	`  cat2`
	`  cat3`
	`  cat4`
	`  cat5`
	`)`

7.6.2 Modeling Methods

The next step is to model the graphical resources that operate on the resources. In each case we model the logic of the method using the formal syntax. The methods that create a resource initialize that resource's data structure. Some properties of a resource are initialized to a default value (e.g. alpha value of a window or pixel data of a surface). These properties are later modified by other graphical methods. The security labels of the resource, however, must be set to those of the source. For example, Table 7.8 illustrates the model of the "CreateSurface" method. It returns a surface list with the first element (pixel_data) initialized to zero, the second and third elements (permanent label and declassification label) initialized to those of the creator window. Note that "win_return_pl" and "win_return_dl" are just helper functions used to return the fourth and fifth elements of a window respectively.

Table 7.8: Modeling CreateSurface

```
(defun create_surface (win)
    (list 0 (win_return_pl win) (win_return_dl win)))
```

A resource acquisition method is more involved. It first checks whether the acquirer's declassification label dominates the permanent label of the resource being accessed. If so, the access is granted; otherwise, it is denied. Table 7.9 shows a resource acquisition method and its model. The model matches the implementation line by line. If the label domination property is satisfied, the surface is assigned to the window. In the implementation, this is done by assigning the surface pointer to the surface element of the window structure. In the model, the assignment is done by returning the surface. Note that when a surface is acquired, the pixel data of the surface remain the same, so the model sets the pixel data (first element) to that of the surface being accessed. Other resource acquisition methods are modeled similarly.

Table 7.9: Modeling GetSurface

Implementation	ACL2 Model

`GetSurface (Window *win, Surface **surface){`
`  if(dominate(win->declassification,`
`  (*surface)->permanent)){`
`  win->surface = *surface;`
`  return STATUS_OK;`
`  }`
`  else{`
`  win->surface=NULL;`
`  return STATUS_BAD_ACCESS;`
`  }`
`}`
	`(defun get_surface (win surface)`
	`  (if (dom (win_return_dl win)`
	`  (sur_return_pl surface))`
	`  (list (sur_return_pixel_data surface)`
	`  (win_return_pl win) (win_return_dl win)`
	`  )`
	`  nil`
	`  )`
	`)`

The drawing functions of a graphics subsystem modify the pixel data of a surface to draw objects on a window. The complexities of drawing different geometric objects are irrelevant to our analysis of a graphics subsystem's behavior. As a result, a drawing function (`draw`) is modeled as a function that simply modifies the pixel data of a surface. Here, the modification is modeled by incrementing the pixel data element by one (Table 7.10).

7.6.3 Writing Theorems

In order to prove properties about a system, one has to describe the desired behavior in the form of theorems. We prove a number of security properties about the simplified subsystem by first writing them in English and then translating them to theorems.

Table 7.10: Modeling drawing functions

```
(defun draw (surface)
   (if surface
       (list (modify_pixels (sur_return_pixel_data surface)) (sur_return_pl surface)
             (sur_return_dl surface))
        nil))
```

Each theorem in ACL2 is defined by the `defthm` construct. A theorem usually consists of a number of assumptions and a statement to be proved. Hence, one way of describing a theorem in ACL2 is using the `(implies P Q)` construct. The first argument of this construct is a list of assumptions and the second argument is the statement. To clarify how the security theorems for the simplified graphics system are developed, we describe two in detail. The complete list of the theorems is presented in the next section.

One desired property is to show that no unauthorized window can access a surface. In other words, if the declassification label of a window does not dominate the permanent label of a surface, the GetSurface function must return NULL. To prove this property we have to assume that the security labels of the window and the surface are not empty. Table 7.11 illustrates the security property and the formal theorem. The property matches the theorem line by line.

Table 7.11: A security theorem on GetSurface

Desired Property	ACL2 Theorem
If the security labels of window and surface are not NULL and the declassification label of the window does not dominate the permanent label of the surface, then GetSurface is denied.	(defthm no_leak_get_surface (implies (AND (consp win) (consp surface) (NOT (dom (win_return_dl win) (sur_return_pl surface)))) (= (get_surface surface win) nil)))

As another example, consider the desired property which states that two unauthorized windows cannot communicate using a data buffer as the communication channel. More precisely, assuming that the security labels are not empty, if the first window's declassification label does not dominate the second window's permanent label, the first window's

GetDataBuffer call on a buffer created by the second window must be denied. This property and its formal theorem are illustrated in Table 7.12.

Table 7.12: A security theorem on GetDataBuffer

Desired Property	ACL2 Theorem
Assuming that the security labels are not empty, if win1's declassification label does not dominate win2's permanent label, a GetDataBuffer call by win1 on a buffer created by win2 is denied.	(defthm no_leak_win_win_data_buffer (implies (AND (consp win1) (consp win2) (NOT (dom (win_return_dl win1) (win_return_pl win2)))) (= (get_data_buffer (create_data_buffer win2) win1) nil)))

The other desired properties of the simplified graphics subsystem are converted into formal theorems in a similar way.

7.7 Formal Verification

Table 7.13 illustrates the complete list of graphical resources models.

Table 7.13: Formal model of the graphical resources

Window	(surface bound alpha permanent_label declassification_label)
Surface	(pixel_data permanent_label declassification_label)
EventBuffer	(queue permanent_label declassification_label)
DataBuffer	(data permanent_label declassification_label)
Label	(level category1 category2 category3 ...)

The formal model also contains a number of helper functions which facilitate accessing different components of the graphical resources. For example, "win_return_surface" returns the surface component of a window, "win_return_bound" returns the bound of a window, etc. The helper functions are listed in Table 7.14

We then model the simplified graphics subsystem by formally describing its main features. Each model represents the behavior and the functionality of a feature. Note that the formal model is much more comprehensive than the one developed for TrustGraph. Table 7.15 lists the complete model.

Table 7.14: Helper functions of the formal model

```
(defun win_return_surface (win)
   (car win))
```

```
(defun win_return_bound (win)
   (car
       (cdr win)))
```

```
(defun win_return_alpha (win)
   (car
       (cdr (cdr win))))
```

```
(defun win_return_pl (win)
   (car (cdr
        (cdr (cdr win)))))
```

```
(defun win_return_dl (win)
   (car (cdr
        (cdr (cdr (cdr win))))))
```

```
(defun sur_return_pixel_data (surface)
   (car surface))
```

```
(defun sur_return_pl (surface)
   (car
       (cdr surface)))
```

```
(defun sur_return_dl (surface)
   (car (cdr
           (cdr surface))))
```

```
(defun buff_return_pl (buff)
   (car
       (cdr buff)))
```

```
(defun buff_return_dl (buff)
   (car (cdr
           (cdr buff))))
```

```
(defun subset (lista listb)
   (if (consp lista)
       (AND
           (member (car lista) listb)
           (subset (cdr lista) listb))
        T))
```

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

When a resource is created (e.g. create_win, create_surface, create_event_buffer, and create_data_buffer), its security labels are initialized with those of the creator. When a resource is acquired (e.g. get_win, get_surface, attach_event_buffer, and get_data_buffer), if the declassification label of the acquirer "dominates" the permanent label of the resource, it

Table 7.15: Formal model of the simplified graphics subsystem

```
(defun create_win (int_PL)
   (list 0 1 2 int_PL int_PL))
```

```
(defun create_surface (win)
   (list 0 (win_return_pl win) (win_return_dl win)))
```

```
(defun create_event_buffer (win)
   (list 0 (win_return_pl win) (win_return_dl win)))
```

```
(defun create_data_buffer (win)
   (list 0 (win_return_pl win) (win_return_dl win)))
```

```
(defun get_win (win int_PL)
   (if (dom int_PL (win_return_pl win))
       (list (win_return_surface win) (win_return_bound win) (win_return_alpha win) int_PL int_PL)
        nil))
```

```
(defun get_surface (surface win)
   (if (dom (win_return_dl win) (sur_return_pl surface))
       (list (sur_return_pixel_data surface) (win_return_pl win) (win_return_dl win))
        nil))
```

```
(defun attach_event_buffer (event_buffer win)
   (if (dom (win_return_dl win) (buff_return_pl event_buffer))
       (list (car event_buffer) (win_return_pl win) (win_return_dl win))
        nil))
```

```
(defun get_data_buffer (data_buffer win)
   (if (dom (win_return_dl win) (buff_return_pl data_buffer))
       (list (car data_buffer) (win_return_pl win) (win_return_dl win))
        nil))
```

```
(defun set_win_label (win int_PL label)
   (if (dom int_PL label)
       (list (win_return_surface win) (win_return_bound win) (win_return_alpha win)
             (win_return_pl win) label)
        win))
```

```
(defun modify_pixels (pixel_data)
   (if pixel_data
       (+ pixel_data 1)
        nil))
```

```
(defun draw (surface)
   (if surface
       (list (modify_pixels (sur_return_pixel_data surface)) (sur_return_pl surface)
             (sur_return_dl surface))
        nil))
```

```
(defun change_alpha (win)
   (if win
       (list (win_return_surface win) (win_return_bound win) (+ (win_return_alpha win) 1)
             (win_return_pl win) (win_return_dl win))
        nil))
```

is granted access to the resource. Otherwise, the method returns NULL. An application can declassify a window through its main interface (using set_win_label) only if the new label is dominated by the application's label. "Draw" models any drawing function in the graphics

subsystem. It simply modifies the pixel values of a surface. Finally, the "change_alpha" function models changing the transparency of a window.

The complete list of theorems about the graphics subsystem is shown in Table 7.16. The theorems are developed from the desired properties as described in the previous section.

Table 7.16: Security theorems of the simplified graphics subsystem

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

```
(defthm no_leak_get_surface
   (implies
       (AND (consp win) (consp surface)
            (NOT (dom (win_return_dl (set_win_label win int_PL label)) (sur_return_pl surface))))
       (= (get_surface surface (set_win_label win int_PL label)) nil)))
```

```
(defthm no_leak_attach_event_buffer
   (implies
       (AND (consp win) (consp event_buffer)
            (NOT (dom (win_return_dl (set_win_label win int_PL label)) (buff_return_pl event_buffer))))
       (= (attach_event_buffer event_buffer (set_win_label win int_PL label)) nil)))
```

```
(defthm no_leak_get_data_buffer
   (implies
       (AND (consp win) (consp data_buffer)
            (NOT (dom (win_return_dl (set_win_label win int_PL label)) (buff_return_pl data_buffer))))
       (= (get_data_buffer data_buffer (set_win_label win int_PL label)) nil)))
```

```
(defthm no_leak_overall
   (implies
       (AND (consp win) (consp surface)
            (NOT (dom int_PL (sur_return_pl surface))) (dom int_PL (win_return_pl win)))
       (= (get_surface surface (get_win win int_PL)) nil)))
```

```
(defthm no_leak_win_surface
   (implies
       (AND (consp (create_win int_PL)) (consp surface)
            (NOT (dom (win_return_dl (set_win_label (create_win int_PL) int_PL label))
                 (sur_return_pl surface))))
       (= (get_surface (draw surface) (set_win_label (create_win int_PL) int_PL label)) nil)))
```

Table 7.16: Security theorems of the simplified graphics subsystem (continued)

```
(defthm no_leak_win_event_buffer
   (implies
       (AND (consp (create_win int_PL)) (consp event_buffer)
            (NOT (dom (win_return_dl (set_win_label (create_win int_PL) int_PL label))
                      (buff_return_pl event_buffer))))
       (= (attach_event_buffer event_buffer (set_win_label (create_win int_PL) int_PL label)) nil)))
```

```
(defthm no_leak_win_data_buffer
   (implies
       (AND (consp (create_win int_PL)) (consp data_buffer)
            (NOT (dom (win_return_dl (set_win_label (create_win int_PL) int_PL label))
                 (buff_return_pl data_buffer))))
   (= (get_data_buffer data_buffer (set_win_label (create_win int_PL) int_PL label)) nil)))
```

```
(defthm no_leak_win_win_surface
   (implies
       (AND (consp win1) (consp win2)
            (NOT (dom (win_return_dl win1) (win_return_pl win2))))
       (= (get_surface (create_surface win2) win1) nil)))
```

```
(defthm no_leak_win_win_event_buffer
   (implies
       (AND (consp win1) (consp win2)
            (NOT (dom (win_return_dl win1) (win_return_pl win2))))
       (= (attach_event_buffer (create_event_buffer win2) win1) nil)))
```

```
(defthm no_leak_win_win_data_buffer
   (implies
       (AND (consp win1) (consp win2)
            (NOT (dom (win_return_dl win1) (win_return_pl win2))))
       (= (get_data_buffer (create_data_buffer win2) win1) nil)))
```

```
(defthm no_leak_int_int_win
   (implies
       (AND (consp int1) (consp int2) (NOT (dom int1 int2)))
       (= (get_win (create_win int2) int1) nil)))
```

```
(defthm no_leak_draw
   (implies
       (AND (consp win) (consp surface)
            (NOT (dom (win_return_dl (set_win_label win int_PL label)) (sur_return_pl surface))))
       (= (get_surface (draw surface) (set_win_label win int_PL label)) nil)))
```

Table 7.16: Security theorems of the simplified graphics subsystem (continued)

```
(defthm no_leak_alpha
   (implies
       (AND (consp win) (consp surface)
            (NOT (dom (win_return_dl win) (sur_return_pl surface))))
       (= (get_surface surface (change_alpha win)) nil)))
```

The first four theorems of Table 7.16 verify that no leakage can happen when a resource acquires another resource (e.g. window, surface, event buffer, and data buffer). The fifth theorem considers the overall effect of all resource acquisition methods. The next three theorems (no_leak_win_surface, no_leak_win_event_buffer, and no_leak_win_data_buffer) verify that an application cannot access a resource through a new window if it normally does not have access to that resource. The next four theorems (no_leak_win_win_surface, no_leak_win_win_event_buffer, no_leak_win_win_data_buffer, no_leak_int_int_win) verify that two resource cannot communicate using another resource as the medium. Finally, the last two theorems (no_leak_alpha and no_leak_draw) prove that drawing and changing the alpha value do not leak information.

ACL2 was able to prove the theorems using 216 axioms and rules and by breaking them into 63 sub-goals. Note that the axioms are all internal to ACL2. They are mathematical properties (e.g. commutativity of integer addition) automatically selected by the ACL2 tool to prove the theorems. The correctness of the axioms is proven by ACL2 in advance.

8 Conclusion and Future Work

This work investigated trusted and high assurance systems in detail. First, we studied different commercial technologies that can be used to build trusted, MILS, and MLS systems. The study covers tamper resistant hardware devices, processors, operating systems, network architectures, databases, and storage technologies. We then selected a candidate architecture based on virtualization, trusted boot and execution, and protected I/O. We have implemented the candidate architecture and identified several security gaps in it. The gaps are the lack of trust in the network, the question of pre-deployment testing for patches/updates applied in a trusted network, and the lack of trust in the I/O subsystems, especially in graphics. We have addressed each security gap in a chapter. A trusted network architecture has been proposed and its security properties and high availability customization are studied in the context of process control systems. We then focused on the problem of updates in a trusted network. More specifically, using stochastic and analytical modeling, we have evaluated the optimal pre-deployment testing period for software patches. Finally, we have designed, implemented, and evaluated a trusted graphics subsystem to cover the gap of trust in the I/O systems. The evaluation is done through different levels of verification which include functionality testing, attack evaluation, formal verification, covert channel analysis, and compiler-based verification.

Future work can take different directions. For trusted networks, we plan to implement a prototype TPCN on top of a cyber-security testbed that we have developed [63]. The testbed is developed to perform assessments and study attack/defense scenarios in a large-scale power infrastructure. With an implementation of a trusted network on top of the testbed, one can evaluate the security services against real attack scenarios.

For patch/update management, more detailed models and empirical distributions can be used to describe the vulnerability discovery process. In addition, a more sophisticated

model for patch development time can result in error reduction. Empirical data can be used to express the probability distribution of this delay. Finally, real experiments on testing patches can result in more accurate models for the number of faults discovered during the testing period.

Future work can also focus on studying trusted graphics in a service-oriented architecture. The challenge is to define the trust requirements of an end-to-end service-oriented graphics system that deals with cross-domain security policies, implement a prototype graphics server, and develop a methodology for an integrated end-to-end analysis. For an end-to-end analysis of trust, the intermediate goal is to develop security techniques for graphics in high assurance MLS systems, in the context of supporting collaboration among applications that operate under different security policies.

Another direction for future work is to formally verify some of the critical components of the VMM. Verification of the VMM involves formal proof of the properties such as "memory non-interference" and "state-less switch." This along with the TXT trusted boot and protected I/O mechanisms provides a comprehensive high assurance MLS system. Using the trustworthy VMM as the platform, a scheme can be developed to establish and verify the properties of the VMs. *Property-based attestation* enables the VMM to validate the configurations or postures of the VMs similar to the way clients prove their postures in a trusted network. The challenge here is that the information available at the operating system level is usually lost at the VMM level because the hypervisor works at a lower abstraction level. For instance, files, memory regions, and network sessions are viewed as disk blocks, pages, and packets in the VMM. To recover the lost abstractions at the VMM level, one can use the idea of virtual machine introspection (VMI). Using property-based attestation, the VMM can validate the posture of the VMs, provide some level of assurance about those properties via the TPM chip, and enforce a flexible and dynamic security policy on the interactions of the VMs.

Finally, we plan to extend the formal verification technique to more general security policies. Security policy models such as type-enforcement, role-based access control, and attribute-based access control provide more expressiveness and flexibility. Formal verification of the security kernel which enforces a more general policy model ensures that even if a

number of components in the system are breached, the security policy is enforced correctly and the system as a whole remains secure.

References

[1] "Commercial Internet Protocol Security Option (CIPSO)," Internet Engineering Task Force (IETF), Internet-Draft, 1992.

[2] "Security Options for the Internet Protocol," Internet Engineering Task Force (IETF), RFC 1108, 1991.

[3] P. A. Karger and D. R. Safford, "I/o for virtual machine monitors: Security and performance issues," *IEEE Security and Privacy*, vol. 6, no. 5, pp. 16–23, 2008.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Ottawa Linux Symposium*, July 2007, pp. 225–230.

[5] 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 *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 164–177.

[6] D. Kilpatrick, W. Salamon, and C. Vance, "Securing the X window system with SELinux," National Security Agency, White Paper, March 2003. [Online]. Available: http://www.nsa.gov/research/_files/publications/securing_xwindow.pdf

[7] "DirectFB," 2009. [Online]. Available: http://www.directfb.org/

[8] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 secure coprocessor," *Computer*, vol. 34, no. 10, pp. 57–66, 2001.

[9] "Trusted platform modules strengthen user and platform authenticity," Trusted Computing Group, White Paper, 2005.

[10] D. Grawrock, *Dynamics of a Trusted Platform: A Building Block Approach*, 1st ed. Hillsboro, OR: Intel Press, 2009.

[11] "Intel trusted execution technology architectural overview," Intel, White Paper, 2003.

[12] D. Grawrock, *The Intel Safer Computing Initiative*, 1st ed. Hillsboro, OR: Intel Press, 2006.

[13] "Intel virtualization technology for directed I/O," Intel, White Paper, 2007.

[14] "Secure virtual machine architecture reference manual," Advanced Micro Devices, White Paper, 2005.

[15] "Intel Centrino Pro and Intel vPro processor technology," Intel, White Paper, 2007.

[16] C. Hanson, "SELinux and MLS: Putting the pieces together," Trusted Computer Solutions, Inc., Beaverton, OR, Tech. Rep. NAI-02-007, 2006.

[17] G. Faden, "Solaris trusted extensions," Sun Microsystems, White Paper, 2006.

[18] "Trusted Solaris 8 operating environment," Sun Microsystems, White Paper, 2000.

[19] R. Watson, "TrustedBSD: Trusted operating system features for BSD," McAfee Security, White Paper, 2004.

[20] "Multilevel security (MLS) by trusted IRIX," SGI, White Paper, 2002.

[21] D. Safford and M. Zohar, "A trusted Linux client (TLC)," IBM TJ Watson Research Center, White Paper, 2004.

[22] M. Peinado, Y. Chen, P. Engl, and J. Manferdelli, "NGSCB: A trusted open system," in *Proceedings of 9th Australasian Conference on Information Security and Privacy (ACISP)*, 2004, pp. 86–97.

[23] "Trusted network connect to ensure endpoint integrity," Trusted Computing Group, White Paper, 2005.

[24] "Cisco TrustSec: Enabling switch security services," Cisco Systems, Inc., White Paper, 2007.

[25] "Cisco NAC Appliance - Clean Access manager installation and configuration guide," Cisco Systems, Inc., Release 4.1(2), 2007.

[26] "Network Admission Control (NAC)," Cisco Systems, Inc., Technical Overview, 2005.

[27] "Network access protection platform architecture," Microsoft Corporation, White Paper, 2007.

[28] "802.1x IEEE standard for local and metropolitan area networks port-based network access control," IEEE Computer Society, Technical Information Bulletin 05-2, 2004.

[29] C. Rayns, "Securing DB2 and implementing MLS on z/OS," IBM, White Paper, 2007.

[30] "Oracle label security - Best practices for government and defense applications," Oracle, White Paper, 2007.

[31] "IBM Tivoli security administrator for RACF," IBM, White Paper, 2006.

[32] J. C. Robinson and J. Alves-Foss, "A high assurance mls file server," *SIGOPS Operating System Review*, vol. 41, no. 1, pp. 45–53, 2007.

[33] "The Trusted Computing Group (TCG) storage specification: Securing storage and information lifecycle management," Trusted Computing Group, White Paper, 2007.

[34] J. Marchesini, S. W. Smith, O. Wild, and R. MacDonald, "Experimenting with TCPA/TCG hardware, or: How I learned to stop worrying and love the bear," Dartmouth College, Hanover, NH, Computer Science Tech. Rep. TR2003-476, 2003.

[35] D. Grawrock, *Dynamics of a Trusted Platform: A Building Block Approach*, 1st ed. Hillsboro, OR: Intel Press, 2009.

[36] "Trusted Boot," 2008. [Online]. Available: http://sourceforge.net/projects/tboot/

[37] "IBM TrouSerS," 2008. [Online]. Available: http://trousers.sourceforge.net/

[38] E. Byres, D. Leversage, and N. Kube, "Security incidents and trends in SCADA and process industries: A statistical review of the Industrial Security Incident Database (ISID)," Symantec Corporation, White Paper, 2007.

[39] K. Stouffer, J. Falco, and K. Scarfone, "NIST guide to Industrial Control Systems (ICS) security," National Institute of Standards and Technology, NIST Special Publication 800-82, 2007.

[40] "The IAONA handbook for network security," Industrial Automation Open Networking Alliance, Version 1.3, 2005.

[41] "Integrating electronic security into the manufacturing and control systems environment," ISA, Research Triangle Park, NC, ISA Tech. Rep. TR99.00.02, 2004.

[42] R. Ross, S. Katzke, A. Johnson, M. Swanson, G. R. G. Stoneburner, and A. Lee, "Recommended security controls for federal information systems," National Institute of Standards and Technology, Gaithersburg, MD, NIST Special Publication 800-53, 2005.

[43] M. Franz and D. Miller, "Industrial ethernet security: Threats & countermeasures," Critical Infrastructure Assurance Group, Cisco Systems, NIST Special Publication 800-53, 2003.

[44] H. Okhravi and D. Nicol, "Applying trusted network technology to process control systems," in *Critical Infrastructure Protection II*, 2nd ed., E. Goetz and S. Shenoi, Eds. Boston, MA: Springer, 2008, pp. 57–70.

[45] H. Okhravi and D. Nicol, "Application of trusted network technology to industrial control networks," *Elsevier International Journal of Critical Infrastructure Protection (IJCIP)*, vol. 2, no. 3, pp. 84–94, 2009.

[46] E. J. Byres, B. Chauvin, J. Karsch, D. Hoffman, and N. Kube, "The special needs of SCADA/PCN firewalls: Architectures and test results," in *10th IEEE Conference on Emerging Technologies and Factory Automation*, 2005, pp. 892–902.

[47] M. Sopko and K. Winegardner, "Process control network security concerns and remedies," in *IEEE Cement Industry Technical Conference Record*, 2007, pp. 26–37.

[48] "Process-based security," SAGE, Inc., White Paper, 2003.

[49] A. Wool, "A quantitative study of firewall configuration errors," *Computer*, vol. 37, no. 6, pp. 62–67, 2004.

[50] E. J. Byres and J. Lowe, "The myths and facts behind cyber security risks for industrial control systems," ISA, White Paper, 2004.

[51] "Potential of plant computer network to worm infection," U.S. Nuclear Regulatory Commission, Information Notice 2003-14, 2003.

[52] S. Kuvshinkova, "SQL Slammer worm lessons learned for consideration by the electricity sector," North American Electric Reliability Council, White Paper, 2003.

[53] "Getting started with Cisco NAC network modules in Cisco access routers," Cisco Systems, Inc., Technical Manual, 2007.

[54] "Network Admission Control (NAC)," Cisco Systems, Inc., Technical Overview, 2005.

[55] "Implementing network admission control phase one configuration and deployment," Cisco Systems, Inc., Tech. Rep. OL-7079-01, 2005.

[56] "Network access protection platform architecture," Microsoft, Technical Manual, 2007.

[57] "Cisco network admission control and microsoft network access protection interoperability architecture," Cisco Systems and Microsoft Corporation, White Paper, 2005.

[58] D. D. Capite, "Self-defending networks: The next generation of network security," Cisco Press, White Paper, 2006.

[59] "IOLAN DS1 device server data sheet," 2009. [Online]. Available: http://www.perle.com/Datasheets/IOLAN-DS1.pdf

[60] "IOLAN STS rackmount data sheet," 2009. [Online]. Available: http://www.perle.com/Datasheets/IOLAN-STS.pdf

[61] A. Hari, S. Suri, and G. Parulkar, "Detecting and resolving packet filter conflicts," in *Proceedings of IEEE INFOCOM*, 2000, pp. 1203–1212.

[62] "CAPEC: Common Attack Pattern Enumeration and Classification," 2008. [Online]. Available: http://capec.mitre.org/

[63] C. M. Davis, J. E. Tate, H. Okhravi, C. Grier, T. J. Overbye, and D. Nicol, "SCADA cyber security testbed development," in *Proceedings of the 38th North American Power Symposium (NAPS 2006)*, 2006, pp. 483–488.

[64] T. Overbye, "Power system simulation: Understanding small- and large-system operations," *IEEE Power and Energy Magazine*, vol. 2, no. 1, pp. 20–30, 2004.

[65] M. Liljenstama, J. Liu, D. Nicol, Y. Yuan, G. Yan, and C. Grier, "RINSE: The real-time immersive network simulation environment for network security exercises," in *Workshop on Principles of Advanced and Distributed Simulation*, 2005, pp. 119–128.

[66] H. Okhravi and D. Nicol, "Evaluation of patch management strategies," *International Journal of Computational Intelligence: Theory and Practice*, vol. 3, no. 2, pp. 109–117, 2008.

[67] O. H. Alhazmi and Y. K. Malaiya, "Prediction capabilities of vulnerability discovery models," in *RAMS '06: Proceedings of Annual Reliability and Maintainability Symposium*, 2006, pp. 86–91.

[68] O. H. Alhazmi and Y. K. Malaiya, "Modeling the vulnerability discovery process," in *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, 2005, pp. 129–138.

[69] E. Rescorla, "Is finding security holes a good idea?" *IEEE Security and Privacy*, vol. 3, no. 1, pp. 14–19, 2005.

[70] E. Byres, D. Leversage, and N. Kube, "Security incidents and trends in SCADA and process industries: A statistical review of the Industrial Security Incident Database (ISID)," Symantec Corporation, White Paper, 2007.

[71] "Symantec internet security threat report," White Paper, 2007, trends for January–June '07.

[72] T. Gerace and H. Cavusoglu, "The critical elements of patch management," in *SIGUCCS '05: Proceedings of the 33rd Annual ACM SIGUCCS Conference on User Services*, 2005, pp. 98–101.

[73] R. J. Anderson, "Security in open versus closed systems—The dance of Boltzmann, Coase and Moore," in *Proceedings of the Conference on Open Source Software Economics*, 2002, p. 5.

[74] J. D. Musa and K. Okumoto, "A logarithmic poisson execution time model for software reliability measurement," in *ICSE '84: Proceedings of the 7th International Conference on Software Engineering*, 1984, pp. 230–238.

[75] "Symantec internet security threat report," White Paper, Sep. 2006, trends for January–June '06.

[76] H. K. Browne, W. A. Arbaugh, J. McHugh, and W. L. Fithen, "A trend analysis of exploitations," in *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001, p. 214.

[77] W. H. Sanders and J. F. Meyer, "Stochastic activity networks: Formal definitions and concepts," *Lectures on formal methods and performance analysis*, vol. 2090/2001, pp. 315–343, 2002.

[78] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius framework and its implementation," *IEEE Transaction on Software Engineering*, vol. 28, no. 10, pp. 956–969, Oct. 2002.

[79] H. Okhravi and D. Nicol, "Trustgraph: Trusted graphics subsystem for high assurance systems," in *ACSAC '09: Proceedings of IEEE Annual Computer Security Applications Conference*, Dec. 2009, pp. 254–265.

[80] R. W. Scheifler and J. Gettys, "The X window system," *ACM Transaction on Graphics*, vol. 5, no. 2, pp. 79–109, Apr. 1986. [Online]. Available: http://dx.doi.org/10.1145/22949.24053

[81] R. S. Thompson, *Quartz 2D Graphics for Mac OS X(R) Developers.* Boston, MA: Addison-Wesley Professional, 2006.

[82] D. Bell and L. LaPadula, "Secure computer systems: Mathematical foundations," MITRE Corp., Tech. Rep. MTR-2547, 1973.

[83] M. Manely, "The x window system must die," White Paper, 2000. [Online]. Available: www.linux.com

[84] "Mozilla DirectFB porting," 2009. [Online]. Available: https://wiki.mozilla.org/Mobile/DFBPorting

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

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

[87] M. Kaufmann and R. S. Boyer, "The boyer-moore theorem prover and its interactive enhancement," *Computers and Mathematics with Applications*, vol. 29, no. 2, pp. 27–62, 1995.

[88] G. Steele, *Common LISP: The Language (LISP Series).* Woburn, MA: Digital Press, June 1984.

[89] "ACL2," 2009. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/

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

[91] "Trusted computer system evaluation criteria," National Security Institute, White Paper, 1985, 5200.28-STD.

[92] "Common criteria security assurance requirements," Common Criteria Recognition Arrangement, White Paper, 2007, cCPART3V3.

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

[94] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 3, pp. 3–30, 1998.

[95] M. Schordan and D. Quinlan, "A source-to-source architecture for user-defined optimizations," in *Joint Modular Languages Conference (JMLC'03)*, 2003, pp. 214–223.

[96] "Compass manual," Lawrence Livermore National Laboratory, Manual, 2009. [Online]. Available: http://www.rosecompiler.org/compass.pdf

[97] "High-integrity c++ coding standard manual," The Programming Research Group, Manual, May 2004. [Online]. Available: http://www.literateprogramming.com/CppHighIntegrity.pdf

[98] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk, "Multiple operating systems on one processor complex," *IBM System Journal*, vol. 28, no. 1, pp. 104–123, 1989.

[99] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, "A retrospective on the vax vmm security kernel," *IEEE Transaction Software Engineering*, vol. 17, no. 11, pp. 1147–1165, 1991.

[100] R. Meushaw and D. Simard, "NetTop: Commercial technology in high assurance applications," *National Security Agency Tech Trend Notes*, vol. 9, no. 4, pp. 3–10, Fall 2000.

[101] J. Alves-Foss, C. Taylor, and P. Oman, "A multi-layered approach to security in high assurance systems," in *HICSS '04: Proceedings of the 37th Annual Hawaii International Conference on System Sciences - Track 9*, 2004, p. 90302.2.

[102] P. A. Karger, "Multi-level security requirements for hypervisors," in *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, 2005, pp. 267–275.

[103] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003, pp. 193–206.

[104] 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*, 2007, pp. 335–350.

[105] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the ucla unix security kernel," *Communications of ACM*, vol. 23, no. 2, pp. 118–131, Feb. 1980. [Online]. Available: http://dx.doi.org/10.1145/358818.358825

[106] "AMD I/O virtualization technology (IOMMU) specification," Advanced Micro Devices, White Paper, 2006, publication no. 34434.

[107] J. Humphreys and T. Grieser, "Mainstreaming server virtualization: The intel approach," IDC, White Paper, 2006.

[108] J. P. L. Woodward, "Security requirements for system high and compartmented mode workstations," MITRE Corporation, Tech. Rep. MTR 9992, Nov. 1987.

[109] 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*, Oct. 1991.

[110] 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," in *Proceedings of the Seventh Annual Computer Security Applications Conference*, Dec. 1991, pp. 44–55.

[111] 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*, 2006, pp. 301–320.

[112] J. Epstein, "A high-performance hardware-based high-assurance trusted windowing system," in *NISSC'96: National Information Systems Security Conference*, 1996, pp. 12–22.

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

[114] J. Picciotto and J. Epstein, "A comparison of trusted X security policies, architectures, and interoperability," in *Proceedings of the Eighth Annual Computer Security Applications Conference*, Dec. 1992, pp. 142–152.

[115] 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*, 2005, pp. 85–94.

[116] E. Walsh, "X access control extension specification," X.org Foundation, White Paper, 2006.

[117] S. Smalley, C. Vance, and W. Salamon, "Implementing SELinux as a Linux security module," National Security Agency, White Paper, May 2002.

[118] E. Walsh, "Application of the Flask architecture to the X window system server," in *Proceedings of the 2007 SELinux Symposium*, 2007, pp. 1–8.

[119] C. Paget, "Exploiting design flaws in the Win32 API for privilege escalation," White Paper, Aug. 2002. [Online]. Available: web.archive.org

Appendix A: Compiler-Based Checkers

Table A.1 provides a complete list of Compass checkers used for compiler-based analysis of TrustGraph.

Table A.1: Complete list of Compass checkers

Allocate And Free Memory In The Same Module	Allowed Functions	Assignment Operator Check Self
Avoid Using The Same Handler For Multiple Signals	Bin Print Asm Functions	Assignment Return Const This
Bin Print Asm Instruction	Binary Buffer Overflow	Boolean Is Has
Buffer Overflow Functions	byte-by-byte comparisons between structures	Binary Interrupt Analysis
Computational Functions	Making string literals const-qualified	Char Star For String
Constructor Destructor Calls Virtual Function	Control Variable Test Against Function	Comma Operator
Copy Constructor Const Arg	Cpp Calls Setjmp Longjmp	Cyclomatic Complexity
Cycle Detection	Const Cast	Data Member Access
Deep Nesting	Default Case	Do Not Assign Pointer To Fixed Address
Default Constructor	Discard Assignment	Do Not Call Putenv With Auto Var
Do Not Delete This	Do Not Use C-style Casts	Empty Instead Of Size
Duffs Device	Dynamic Cast	Enum Declaration Namespace Class Scope
Explicit Char Sign	Explicit Copy	Float For Loop Counter
Explicit Test For Non Boolean Value	File Read Only Access	Floating Point Exact Comparison
Fopen Format Parameter	For Loop Construction Control Stmt	Friend Declaration Modifier
For Loop Cpp Index Variable Declaration	Forbidden Functions	Function Call Allocates Multiple Resources
Function Definition Prototype	Function Documentation	Loc Per Function
Induction Variable Update	Internal Data Sharing	Localized Variables
Lower Range Limit	Magic Number	Malloc Return Value Used In If Stmt
Name All Parameters	New Delete	Multiple Public Inheritance
No Asm Stmts Ops	No Exceptions	No Overload Ampersand
No Exit In Mpi Code	No Goto	No Rand
No Second Term Side Effects	Operands to sizeof should not contain side effects	No Template Usage
No Vfork	Non Associative Relational Operators	No Variadic Functions
Non Standard Type Ref Args	Non Standard Type Ref Returns	Non Virtual Redefinition
Null Dereference	Omp Private Lock	Nonmember Function Interface Namespace

Table A.1: Complete list of Compass checkers (continued)

Other Argument	Place Constant On The Lhs	One Line Per Declaration
Pointer Comparison	Prefer Algorithms	Operator Overloading
Prefer fseek() to rewind()	Prefer Setvbuf To Setbuf	Protect Virtual Methods
Right Shift Mask	Set Pointers To Null	Push Back
Single Parameter Constructor Explicit Modifier	Size Of Pointer	Ternary Operator
Use strtol() to convert string token to an integer	Sub Expression Evaluation Order	Time to Direct Manipulation
Unary Minus	Uninitialized Definition	Upper Range Limit
Void Star	Variable Name Equals Database Name	

Appendix B: Code Violations in TrustGraph

Table B.1 lists the security or coding violations in TrustGraph.

Table B.1: The list of security or coding violations in TrustGraph

Source	Violation	Suggestion
clipboard.c :264.11-40	in function: dfb_clipboard_get Every malloc must be followed by a free	Explicitly free the allocated memory.
clipboard.c :186.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
clipboard.c :189.6	Variable new_data does not seem to be used right after its declaration.	Try to move it down in the code.
clipboard.c :244.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
clipboard.c :280.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
clipboard.c :264.19-24	This checker checks if the return value of calling malloc is part of an If-Statement conditional test. call to malloc does not have a corresponding If-statement conditional in this block.	Check the NULL condition using If-statement.
colorhash.c :133.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
colorhash.c :171.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
colorhash.c :186.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
colorhash.c :204.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
colorhash.c :234.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
colorhash.c :297.11-35	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
colorhash.c :313.16-316.16	Place the constant on the left hand side in this comparison!	Place the constant on the LHS.
colorhash.c :309.48-69	Finding a RightShift with no Bit Mask	Define a bit mask to avoid sign confusion.
core.c :828.6-49	in function: dfb_core_cleanup_add Every malloc must be followed by a free.	Explicitly free the allocated memory.
core.c :243.1-244.37	dfb_core_create This function is too complex CC = 35 > 30	Reduce complexity by breaking the function into multiple functions.

Table B.1: The list of security or coding violations in TrustGraph (continued)

Source	Violation	Suggestion
core.c :983.16-68	Result of assignment should be discarded.	Remove the assignment if not needed.
core.c :1015.16-62	Result of assignment should be discarded.	Remove the assignment if not needed.
core.c :905.1-906.50	matching function prototype not available	Explicitly define a function prototype.
core.c :956.1-957.36	matching function prototype not available	Explicitly define a function prototype.
core.c :992.1-993.47	matching function prototype not available	Explicitly define a function prototype.
core.c :1005.1-1006.30	matching function prototype not available	Explicitly define a function prototype.
core.c :436.13-438.66	This checker checks that relational binary operators (==, ! =, <, >, <=, >=) are not treated as if they were non-associative.	Make sure LHS and RHS are associative.
core.c :277.6	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
core.c :344.17-38	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
core.c :437.32-73	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
core.c :999.41-44	Finding examples of using upper range inclusive	Always use inclusive lower limits and exclusive upper limits.
core_parts.c :53.6	Variable ret does not seem to be used right after its declaration.	Try to move it down in the code.
core_parts.c :55.6	Variable ret does not seem to be used right after its declaration.	Try to move it down in the code.
core_parts.c :103.6	Variable ret does not seem to be used right after its declaration.	Try to move it down in the code.
core_parts.c :104.6	Variable ret does not seem to be used right after its declaration.	Try to move it down in the code.
core_parts.c :105.6	Variable ret does not seem to be used right after its declaration.	Try to move it down in the code.
core_parts.c :146.6	Variable ret does not seem to be used right after its declaration.	Try to move it down in the code.
core_parts.c :180.6	Variable ret does not seem to be used right after its declaration.	Try to move it down in the code.
gfxcard.c :210.11-81	In function: dfb_graphics_core_initialize Every malloc must be followed by a free.	Explicitly free the allocated memory.
gfxcard.c :299.11-81	In function: dfb_graphics_core_join Every malloc must be followed by a free.	Explicitly free the allocated memory.
gfxcard.c :990.26-991.36	Scope is nested deeper than allowed.	Use helper functions to decrease the nesting.
gfxcard.c :539.1-540.70	dfb_gfxcard_state_check This function is too complex : CC = 55 > 30	Reduce complexity by breaking the function into multiple functions.
gfxcard.c :941.1-942.82	dfb_gfxcard_fillrectangles This function is too complex : CC = 56 > 30	Reduce complexity by breaking the function into multiple functions.
gfxcard.c :729.1-730.72	matching function prototype not available	Explicitly define a function prototype.

Table B.1: The list of security or coding violations in TrustGraph (continued)

Source	Violation	Suggestion
gfxcard.c :873.1-874.45	matching function prototype not available	Explicitly define a function prototype.
gfxcard.c :1085.1-1089.60	matching function prototype not available	Explicitly define a function prototype.
gfxcard.c :2100.1-2101.52	matching function prototype not available	Explicitly define a function prototype.
gfxcard.c :964.16-20	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
gfxcard.c :1527.11-13	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
gfxcard.c :1150.6-28	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
gfxcard.c :1220.21-36	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
gfxcard.c :1221.21-36	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
gfxcard.c :1024.31	Finding a RightShift with no Bit Mask	Define a bit mask to avoid sign confusion.
gfxcard.c :1055.31	Finding a RightShift with no Bit Mask	Define a bit mask to avoid sign confusion.
gfxcard.c :457.12-459.49	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
gfxcard.c :1024.31:	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
gfxcard.c :1515.34-57	Finding examples of using upper range inclusive	Always use inclusive lower limits and exclusive upper limits.
input.c :1399.1-1400.64	fixup_key_event This function is too complex : CC = 58 > 30	Reduce complexity by breaking the function into multiple functions.
input.c :1785.1-1788.49	id_to_symbol This function is too complex : CC = 55 > 30	Reduce complexity by breaking the function into multiple functions.
input.c :1161.16	Scope is nested deeper than allowed.	Use helper functions to decrease the nesting.
input.c :448.6-491.6	This checker checks that only loop control expressions appear in the for loop constructor block	Put other expressions outside the constructor.
input.c :515.6-519.6	This checker checks that only loop control expressions appear in the for loop constructor block	Put other expressions outside the constructor.
input.c :960.1-961.43	matching function prototype not available	Explicitly define a function prototype.
input.c :1016.1-1017.36	matching function prototype not available	Explicitly define a function prototype.
input.c :1035.1-1036.40	matching function prototype not available	Explicitly define a function prototype.
input.c :1625.26-28	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
input.c :1625.54-56	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
input.c :449.11	Variable driver does not seem to be used right after its declaration.	Try to move it down in the code.

Table B.1: The list of security or coding violations in TrustGraph (continued)

Source	Violation	Suggestion
input.c :435.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
input.c :1311.15-73	This checker checks that relational binary operators (==, ! =, <, >, <=, >=) are not treated as if they were non-associative.	Make sure LHS and RHS are associative.
input.c :1493.10-57	This checker checks that relational binary operators (==, ! =, <, >, <=, >=) are not treated as if they were non-associative.	Make sure LHS and RHS are associative.
input.c :448.6	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
input.c :515.6	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
input.c :1211.25-57	Finding examples of using upper range inclusive	Always use inclusive lower limits and exclusive upper limits.
layer_context.c :640.26-641.36	Scope is nested deeper than allowed.	Use helper functions to decrease the nesting.
layer_context.c :93.1-94.66	matching function prototype not available	Explicitly define a function prototype.
layer_context.c :773.1-776.65	matching function prototype not available	Explicitly define a function prototype.
layer_context.c :568.6:	Variable layer does not seem to be used right after its declaration.	Try to move it down in the code.
layer_context.c :788.11	Variable layer does not seem to be used right after its declaration.	Try to move it down in the code.
layer_context.c :62.1	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
layer_context.c :238.6	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
layer_context.c :434.20-32	goto found	Use loop statements instead.
layer_context.c :238.6	Finding any 'and' or 'or' with a side effect somewhere on the right hand side.	The RHS may never be evaluated. Remove the side effect.
layer_context.c :291.6	Finding any 'and' or 'or' with a side effect somewhere on the right hand side.	The RHS may never be evaluated. Remove the side effect.
layer_context.c :1416.11-27	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
layer_control.c :80.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
layer_control.c :125.6	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
layer_control.c :241.10-29	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
layer_control.c :329.10-29	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
layer_control.c :393.26-36	goto found	Use loop statements instead.
layer_control.c :404.21-31	goto found	Use loop statements instead.

Table B.1: The list of security or coding violations in TrustGraph (continued)

Source	Violation	Suggestion
layer_control.c :385.15-31	Finding examples of using upper range inclusive	Always use inclusive lower limits and exclusive upper limits.
layer_region.c :413.1-416.58	fb_layer_region_flip_update This function is too complex : CC = 50 > 30	Reduce complexity by breaking the function into multiple functions.
layer_region.c :469.21	Scope is nested deeper than allowed.	Use helper functions to decrease the nesting.
layer_region.c :71.1-72.65	matching function prototype not available	Explicitly define a function prototype.
layer_region.c :418.6	Variable ret does not seem to be used right after its declaration.	Try to move it down in the code.
layer_region.c :597.6	Variable ret does not seem to be used right after its declaration.	Try to move it down in the code.
layers.c :428.6-45	in function: dfb_layers_register Every malloc must be followed by a free.	Explicitly free the allocated memory.
layers.c :184.26	Scope is nested deeper than allowed.	Use helper functions to decrease the nesting.
layers.c :125.11	Variable buf does not seem to be used right after its declaration.	Try to move it down in the code.
layers.c :283.11	Variable shared does not seem to be used right after its declaration.	Try to move it down in the code.
layers.c :124.16-31	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
layers.c :176.26-55	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
layers.c :293.16	Finding any 'and' or 'or' with a side effect somewhere on the right hand side.	The RHS may never be evaluated. Remove the side effect.
layers.c :421.6-425.6	Place the constant on the left hand side in this comparison!	Place the constant on the LHS.
layers.c :545.11-546.64	Place the constant on the left hand side in this comparison!	Place the constant on the LHS.
layers.c :325.11	in function: dfb_layer_core_shutdown Set all dynamically allocated pointers to NULL after calling free().	Explicitly set the pointers to NULL.
layers.c :360.11	in function: dfb_layer_core_leave Set all dynamically allocated pointers to NULL after calling free().	Explicitly set the pointers to NULL.
layers.c :384.31-34	Finding examples of using upper range inclusive	Always use inclusive lower limits and exclusive upper limits.
local_surface_pool.c :59.1-65.56	matching function prototype not available	Explicitly define a function prototype.
local_surface_pool.c :76.1-77.19	matching function prototype not available	Explicitly define a function prototype.
local_surface_pool.c :82.1-83.24	matching function prototype not available	Explicitly define a function prototype.
local_surface_pool.c :102.6	Variable local does not seem to be used right after its declaration.	Try to move it down in the code.
local_surface_pool.c :127.6	Variable local does not seem to be used right after its declaration.	Try to move it down in the code.
local_surface_pool.c :67.6	in function: local_surface_pool_call_handler Set all dynamically allocated pointers to NULL after calling free()	Explicitly set the pointers to NULL.

Table B.1: The list of security or coding violations in TrustGraph (continued)

Source	Violation	Suggestion
palette.c :62.1-78	matching function prototype not available	Explicitly define a function prototype.
palette.c :178.11	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
palette.c :205.11	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
palette.c :293.6-296.6	Place the constant on the left hand side in this comparison!	Place the constant on the LHS.
palette.c :172.47-61	Finding a RightShift with no Bit Mask	Define a bit mask to avoid sign confusion.
palette.c :200.47-60	Finding a RightShift with no Bit Mask	Define a bit mask to avoid sign confusion.
palette.c :171.35-49	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
palette.c :198.35-49	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
palette.c :269.10-45	Finding examples of using upper range inclusive	Always use inclusive lower limits and exclusive upper limits.
prealloc_surface_pool.c :47.1-48.28	matching function prototype not available	Explicitly define a function prototype.
prealloc_surface_pool.c :74.1-79.53	matching function prototype not available	Explicitly define a function prototype.
prealloc_surface_pool.c :117.55	Finding a RightShift with no Bit Mask	Define a bit mask to avoid sign confusion.
prealloc_surface_pool.c :126.26	Finding a RightShift with no Bit Mask	Define a bit mask to avoid sign confusion.
prealloc_surface_pool.c :84.13-76	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
screen.c :488.21-48	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
screen.c :510.21-48	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
screens.c :449.6-47	in function: dfb_screens_register Every malloc must be followed by a free.	Explicitly free the allocated memory.
screens.c :188.21-23	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
screens.c :210.21-23	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
screens.c :232.21-23	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
screens.c :105.16-28	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
screens.c :188.26-54	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
screens.c :442.6-446.6	Place the constant on the left hand side in this comparison!	Place the constant on the LHS.
screens.c :558.11-559.30	Place the constant on the left hand side in this comparison!	Place the constant on the LHS.
screens.c :348.11	in function: dfb_screen_core_shutdown Set all dynamically allocated pointers to NULL after calling free().	Explicitly set the pointers to NULL.
screens.c :381.11	in function: dfb_screen_core_leave Set all dynamically allocated pointers to NULL after calling free().	Explicitly set the pointers to NULL.

Table B.1: The list of security or coding violations in TrustGraph (continued)

Source	Violation	Suggestion
screens.c :321.28-31	Finding examples of using upper range inclusive	Always use inclusive lower limits and exclusive upper limits.
shared_surface_pool.c :61.1-62.20	matching function prototype not available	Explicitly define a function prototype.
shared_surface_pool.c :67.1-68.25	matching function prototype not available	Explicitly define a function prototype.
state.c :54.1-58.32	matching function prototype not available	Explicitly define a function prototype.
state.c :395.10-18	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
state.c :153.16	in function: dfb_state_destroy Set all dynamically allocated pointers to NULL after calling free().	Explicitly set the pointers to NULL.
state.c :155.11	in function: dfb_state_destroy Set all dynamically allocated pointers to NULL after calling free().	Explicitly set the pointers to NULL.
state.c :161.11	in function: dfb_state_destroy Set all dynamically allocated pointers to NULL after calling free().	Explicitly set the pointers to NULL.
surface.c :53.1-54.66	matching function prototype not available	Explicitly define a function prototype.
surface.c :76.16-18	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
surface.c :217.16-24	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
surface.c :213.16-26	goto found	Use loop statements instead.
surface.c :223.16-26	goto found	Use loop statements instead.
surface.c :460.16-26	goto found	Use loop statements instead.
surface.c :297.37	Finding a RightShift with no Bit Mask	Define a bit mask to avoid sign confusion.
surface_buffer.c :573.1-576.52	dfb_surface_buffer_dump This function is too complex : CC = 108 > 30	Reduce complexity by breaking the function into multiple functions.
surface_buffer.c :1120.1-1122.51	update_allocation This function is too complex : CC = 52 > 30	Reduce complexity by breaking the function into multiple functions.
surface_buffer.c :941.26-942.79	Scope is nested deeper than allowed.	Use helper functions to decrease the nesting.
surface_buffer.c :944.26-945.79	Scope is nested deeper than allowed.	Use helper functions to decrease the nesting.
surface_buffer.c :844.26-857.26	Finding switch statements with no default	Add a default.
surface_buffer.c :592.6	Pointer gz_ext is initialized to a fixed address.	A pointer may not be necessary.
surface_buffer.c :784.21-790.21	This checker checks that only loop control expressions appear in the for loop constructor block	Put other expressions outside the constructor.
surface_buffer.c :796.21-802.21	This checker checks that only loop control expressions appear in the for loop constructor block	Put other expressions outside the constructor.
surface_buffer.c :1052.1-1057.46	matching function prototype not available	Explicitly define a function prototype.
surface_buffer.c :1201.21-23	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.

Table B.1: The list of security or coding violations in TrustGraph (continued)

Source	Violation	Suggestion
surface_buffer.c :1214.21-23	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
surface_buffer.c :186.6	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
surface_buffer.c :396.21-28	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
surface_buffer.c :136.6	Finding any 'and' or 'or' with a side effect somewhere on the right hand side.	The RHS may never be evaluated. Remove the side effect.
surface_buffer.c :186.6	Finding any 'and' or 'or' with a side effect somewhere on the right hand side.	The RHS may never be evaluated. Remove the side effect.
surface_buffer.c :358.6-32	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
surface_buffer.c :389.15	Finding a RightShift with no Bit Mask	Define a bit mask to avoid sign confusion.
surface_buffer.c :583.36-68	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
surface_pool.c :58.1-59.40	matching function prototype not available	Explicitly define a function prototype.
surface_pool.c :71.1-72.40	matching function prototype not available	Explicitly define a function prototype.
surface_pool.c :242.11-22	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
surface_pool.c :287.11-22	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
surface_pool.c :182.10-39	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
surface_pool.c :436.16-26	goto found	Use loop statements instead.
surface_pool.c :444.11-21	goto found	Use loop statements instead.
surface_pool.c :171.10-172.82	Finding any 'and' or 'or' with a side effect somewhere on the right hand side.	The RHS may never be evaluated. Remove the side effect.
surface_pool.c :626.6	Finding any 'and' or 'or' with a side effect somewhere on the right hand side.	The RHS may never be evaluated. Remove the side effect.
surface_pool.c :378.13-54	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
system.c :435.6-22	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
system.c :453.6-42	This checker checks that only one variable declaration occurs per line.	Break the variable definition to multiple lines.
windows.c :1303.28-58	Casting the const away form a type is not allowed.	Remove the casting.
windows.c :202.21	Scope is nested deeper than allowed.	Use helper functions to decrease the nesting.
windows.c :533.6-539.6	This checker checks that only loop control expressions appear in the for loop constructor block	Put other expressions outside the constructor.
windows.c :100.1-101.65	matching function prototype not available	Explicitly define a function prototype.

Table B.1: The list of security or coding violations in TrustGraph (continued)

Source	Violation	Suggestion
windows.c :137.1-144.53	matching function prototype not available	Explicitly define a function prototype.
windows.c :195.11-80	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
windows.c :353.24-85	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
windows.c :374.26-76	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
windowstack.c :174.11-18	Induction variables should not be updated inside of its loop body.	Use extra loops to avoid updating the induction variable.
windowstack.c :573.10-14	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
windowstack.c :796.16-19	Finding examples of using lower range exclusive	Always use inclusive lower limits and exclusive upper limits.
windowstack.c :792.11-22	goto found	Use loop statements instead.
windowstack.c :802.16-27	goto found	Use loop statements instead.
windowstack.c :797.15-62	This checker checks that relational binary operators (==,! =, <, >, <=, >=) are not treated as if they were non-associative.	Make sure LHS and RHS are associative.
windowstack.c :458.35-69	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.
wm.c :165.11-70	in function: dfb_wm_core_initialize Every malloc must be followed by a free.	Explicitly free the allocated memory.
wm.c :250.11-70	in function: dfb_wm_core_join Every malloc must be followed by a free.	Explicitly free the allocated memory.
wm.c :130.11-21	goto found	Use loop statements instead.
wm.c :149.11-21	goto found	Use loop statements instead.
wm.c :157.16-26	goto found	Use loop statements instead.
wm.c :168.16-26	goto found	Use loop statements instead.
wm.c :186.11	in function: dfb_wm_core_initialize Set all dynamically allocated pointers to NULL after calling free().	Explicitly set the pointers to NULL.
wm.c :271.11	in function: dfb_wm_core_join Set all dynamically allocated pointers to NULL after calling free().	Explicitly set the pointers to NULL.
wm.c :901.45-80	Do not use the ternary operator(?:) in expressions	Use explicit conditional statements.