SecureBus: Towards Application-Transparent Trusted Computing with Mandatory Access Control

Xinwen Zhang*
George Mason University
Fairfax, Virginia, USA
xzhang6@gmu.edu

Songqing Chen
George Mason University
Fairfax, Virginia, USA
sqchen@cs.gmu.edu

Michael J. Covington
Intel Corporation
Hillsboro, Oregon, USA
Michael.J.Covington@intel.com

Ravi Sandhu
George Mason University
and TriCipher Inc., USA
sandhu@gmu.edu

ABSTRACT

The increasing number of software-based attacks has attracted substantial efforts to prevent applications from malicious interference. For example, Trusted Computing (TC) technologies have been recently proposed to provide strong isolation on application platforms. On the other hand, today pervasively available computing cycles and data resources have enabled various distributed applications that require collaboration among different application processes. These two conflicting trends grow in parallel. While much existing research focuses on one of these two aspects, a few authors have considered simultaneously providing strong isolation as well as collaboration convenience, particularly in the TC environment. However, none of these schemes is transparent. That is, they require modifications either of legacy applications or the underlying Operating System (OS).

In this paper, we propose the SecureBus (SB) architecture, aiming to provide strong isolation and flexible controlled information flow and communication between processes at runtime. Since SB is application and OS transparent, existing applications can run without changes to commodity OS's. Furthermore, SB enables the enforcement of general access control policies, which is required but difficult to achieve for typical legacy applications. To study its feasibility and performance overhead, we have implemented a prototype system based on User-Mode Linux. Our experimental results show that SB can effectively achieve its design goals.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Security, Management

Keywords

SecureBus, Trusted Computing, Mandatory Access Control, Secure Platform

1. INTRODUCTION

Ever-increasing software-based attacks have demonstrated that existing Operating Systems (OS's) cannot provide sufficient isolation for the security demands of legacy applications. Significant research effort has been made to strengthen isolation among processes at runtime. For example, recently the emerging Trusted Computing (TC) technologies have been developed to enforce strong isolation between applications to preserve their integrity at runtime, and provide verifiable trust to remote entities. In TC, the root of trust is generally based on trusted hardware, such as a secure coprocessor [28] or trusted platform module (TPM) [3].

Despite the rapid growth of security threats, widely available commodity computing resources have enabled a large scope of distributed applications, such as Peer-to-Peer (P2P) and grid-based. Such applications often demand collaborations among connected computing nodes, where various processes need to communicate with each other in order to exchange data and share resources on a single platform or across different platforms via networking.

The increase of software-based attacks and the growth of collaborative applications conflict in computing and system requirements. To prevent attacks from outside, processes should be isolated and integrity must be protected at runtime through isolation in memory. For example, an email with an attachment is commonly used for spreading viruses. If the email client and the process launched to view/execute the attachment can be isolated so that the latter cannot read or modify the former's sensitive data, e.g. address book, such attacks would have not succeeded. On the other hand, the frequent and dynamic interactions required between processes in many distributed applications make a running process susceptible to easy compromise by direct interference from other processes, or

*Now at Samsung Information Systems America, San Jose, California, USA.

ASIACCS'07, March 20-22, 2007, Singapore.

by fake/malicious input, or by unexpected data from sources with lower integrity or sensitivity levels.

The majority of existing research focuses on either providing process integrity assurances at runtime via various isolation mechanisms, such as the TC technologies, or providing flexible communications, such as the design of Globus [14]. Only a few authors have considered how to simultaneously provide strong isolation with flexible communication support. For example, in Microsoft's Next Generation Secure Computing Base (NGSCB) [7], a secure kernel, referred to as Nexus, provides separated runtime environments for individual application agents. Communication and access control between agents are mediated by a trusted service provider (TSP) in user space. Proper [20] is a user space application that provides access control services for privileged operations between application-level virtual machines (VMs) on a PlanetLab node. In BIND [27], critical sections of a running process are isolated and there is no runtime communication between the isolated sections and outside.

Although these approaches could be efficient under certain circumstances for achieving both strong isolation and flexible collaborations, a common problem of these schemes is that either applications need to be substantially modified to use the security services provided by the underlying layers, or the underlying OS must be customized or trusted. For example, in NGSCB, the secure kernel and all supported services and applications are located in an area called *Nexus mode* that is separated from the original OS and all applications are developed specifically to run in this Nexus mode. Existing applications can only run in the standard mode, where a legacy OS is running. Thus they cannot use the security services provided by the new design. Proper is dedicated for a specific Linux distribution (Linux VServers) that is deployed on PlatnetLab nodes, and it is difficult for Proper to be applied in other environments. In BIND [27], critical sections of a process code have to be identified and the invocation of the security functions provided by the secure kernel needs to be inserted into these sections. This approach leads to a situation in which most existing applications and OS's cannot benefit from the strong security provided by emerging TC technologies. Furthermore, typically these approaches only support some specific collaboration modes. For example, BIND does not permit input once a critical section of a process has been isolated and it cannot provide information flow control between processes.

In this paper we propose a novel architecture called SecureBus (SB), which can provide simultaneous strong isolation and transparent access control enforcement between processes. Built on TC, trusted hardware is leveraged to provide the root of trust and extend trust to a secure kernel (SK) and SB, in turn. The SB enhances existing TC technologies and has the following features.

- Built on trusted hardware, SB provides strong runtime process isolation by allocating and maintaining separated runtime memory space for each process. The authenticity of a process's running code and its input and output data is achieved by using digital signatures with the corresponding usage contexts. Specifically, a process's code is hashed by SB before being loaded. The output data of the process is signed by SB and concatenated with the hash values of the process code and the input.
- By leveraging the trust chain from the trusted root of a platform to SK and to applications, SB supports process-based attestation to enable secure communications and collaborations between processes. On a single platform, SB acts as a trusted proxy of an isolated process to communicate with other processes. In a distributed environment, SB on a platform builds a *trusted* channel with its counterpart on another platform through remote attestation.
- SB provides a reference monitor that implements flexible access control between isolated processes. Various access control and integrity policies can be enforced, such as role-based, history-based, and mandatory access control policies, as well as application- and organization-specific policies.

Compared with existing approaches, SB is transparent to upper-layer applications and the underlying OS. At the application level, SB provides the same system call interface as the OS does and enforces authenticity verification and access control policies transparently. Meanwhile, since SB invokes system calls on behalf of the protected applications, our architecture does not require modifications of the underlying OS. That is, existing applications can run on top of SB without change, and existing OS can use SB to provide strong trusted computing and security services, such as isolated runtime space and controlled accesses between processes, for applications.

As SB is transparent but has trust relationship with applications, common security functions such as authentication and authorization can be implemented by SB for the purpose of confidentiality and integrity. Thus, our architecture enables the separation of functionalities and security demands of applications, which reduces the burden to application developers and provides flexibility for security configurations.

To verify the feasibility of our approach and study the performance overhead of the SB architecture, we have implemented a prototype based on User-Mode Linux [13]. As a proof-of-concept, we also implemented the lattice-based Chinese Wall policy [26] to control information flow in the prototype. Experiments performed based on the prototype system show that it is effective with reasonable performance overhead.

The remainder of this paper is organized as follows. Section 2 presents our problem statement and an overview of the design space. In Section 3, our proposed architecture and the primitive functions of SB are introduced. Section 4 and Section 5 describe how to achieve integrity and authenticity verification, in addition to the access control policy and its enforcement using SB. We present a prototype implementation and some experimental results in Section 6. Related work is reviewed in Section 7 and we make concluding remarks in Section 8.

2. PROBLEM STATEMENT AND DESIGN SPACE

Modern operating systems support process abstraction with isolation, flexible sharing (e.g., of file systems, OS resources, and hardware), and inter-process communication (IPC). With an increasing amount of loss and damage caused by various software-based attacks, it has been realized that a commodity OS alone cannot provide a high-assurance environment for applications. Understanding and using trust at the application layer has been studied for a long time, such as in the Database Interpretation of the Orange

Book [12]. Recently a new approach called Trusted Computing (TC) has been developed by leading industry vendors for this purpose. A distinguishing aspect of TC is that it combines cryptographic mechanisms with access control. Keys providing the root of cryptographic trust are not only protected in a separate hardware component but their use is also limited to approved software.

The Trusted Computing Group (TCG) has defined a set of specifications aiming at providing a hardware-based root of trust and a set of primitive functions that allow trust to propagate to application software. The root of trust in the TCG approach is a hardware component on the platform called Trusted Platform Module (TPM). Application-level trust requires strong integrity checks of binary code for running processes and a mechanism that allows other entities (applications or platforms) to verify the integrity. A TPM has the capabilities to measure and report runtime configurations of the platform, from BIOS to OS. TPM and TC-enhanced hardware technologies such as Intel's LaGrande Technology (LT) [1] and AMD's Secure Execution Mode (SEM) [6] generally allocate isolated memory partitions to different application processes to prevent software-based attacks at runtime.

Although isolation based on TC ensures the *binary code's integrity* of an application at runtime, it creates hurdles if the process needs to communicate or share information with other isolated processes on the same platform. For example, when processes are strongly isolated in memory space at runtime, traditional IPC mechanisms, such as shared memory, are no longer viable.

Moreover, even with strong isolation for runtime integrity warranty, such isolation cannot guarantee the authenticity of the communicating party and the data flow (e.g., input and output) between applications, and cannot provide flexible access control mechanisms between applications. These two aspects are critical to preserve the overall integrity of a system, which is not solely dependent on its running code's integrity. In the former case, a (receiving) process needs to ensure that the data it is receiving is trusted, i.e., the source process is genuine. Furthermore, the trust should be verifiable. For the latter, sometimes even if a data-sending process can be trusted, it may have lower integrity or higher confidentiality than that of the receiving process. Receiving such data may compromise the integrity of its own data or confidentiality of the sender's data. Therefore, access control should be enforced between processes to satisfy certain security requirements defined by the system administrator. Currently such interactions between applications heavily depend on the underlying OS, which can easily cause illegal information flow and compromise the overall integrity and confidentiality of the system. For example, even if an application runs in an isolated memory space and no other process (e.g., virus) can modify it, spyware can invoke its functions by providing some fake/junk input, thus compromise the integrity of the process's data. If a process takes input from the network, the input can be eavesdropped or even modified by malicious processes at the OS layer.

Several approaches have been proposed to enhance the security at the OS level, such as security-enhanced Linux (SELinux) [19] and TrustedBSD [5]. In these systems, the OS kernel is extended to include authorization modules which enforce access control policies. With an increasing number of attacks launched at the OS kernel level, such as malicious device drivers and rootkits [8, 18], these systems cannot provide high assurance for trusted computing services.

Similar to enforcing isolation between processes with a secure kernel (e.g., with Nexus in NGSCB), an intuitive approach to address the above problems is to use a secure kernel as an intermediary for inter-process communications, i.e., it forwards messages between processes. While this is applicable for some simple cases, it would be very complex for the kernel to implement all kinds of communication mechanisms, since most of them need involvement of the OS, for example to access the local file system or the network stack. Therefore it is difficult for a secure kernel to enforce effective and flexible access control between processes. Even if it is possible, with these functionalities, the trust of the secure kernel is difficult to maintain, because an important consideration to achieve its trustworthy status is to make it as simple and small as possible.

3. OVERVIEW OF SECUREBUS DESIGN

Figure 1 shows the architecture of a SecureBus-enhanced platform. On this platform, the hardware layer (comprising a TCG-compliant TPM and other necessary hardware such as LT-enabled CPU and chipset) provides the root of trust for TC. The secure kernel (SK) provides a protected runtime environment for SB.

SB is the middle layer between kernel space and user space. SB allocates isolated memory space for each process before the process starts to run. With SB, all interactions between a process and the OS are conducted through SB. Interactions between two isolated processes are monitored by a *reference monitor* in SB and are controlled according to pre-defined policies (see Section 5). Other related services can be in user space for the security management purpose, such as policy definition and administration.

One of SB's design goals is that it should be transparent to applications and OS, which enables most existing legacy software to run on commodity OS without changes. This requires that SB should provide the same interface as a normal OS does. When a process uses the interface, it is transparent to the process that now the interface is provided by SB instead of the OS. For each access request from one process to another, SB validates the access by querying pre-defined access control policies. If this access is allowable, SB forwards it to the OS silently. If this access is denied, an exception is returned to the requesting process. Thus SB works like a middleware such as Java Virtual Machine (JVM), which provides a transparent interface to applications and controls their accesses to underlying OS resources. However, since SB does not provide other complex functionalities such as a platform-independent runtime environment, its properties and behaviors can have high assurance and possibly even be formally verified.

Note that for platform management, there are applications and system services that run in user space without the involvement of SB. Typically, these applications, such as installing/updating system software and patching the system kernel, are "trusted" by the platform administrator such that they have administrative privileges.

Under this architecture, we now present our trust model and the primitive functions of SB.

3.1 The Trust Model

The integrity of SK is measured by TPM when the system starts. Also, SK is protected in memory space by hardware so that its integrity is guaranteed at runtime.

For local applications, before SB is started, SK measures SB's integrity and stores its hash value locally. In turn, when a program is

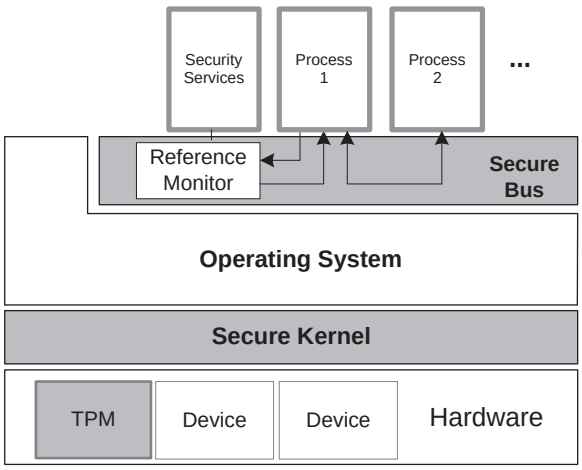

Figure 1: Platform architecture with SB

loaded, SB measures the integrity of the program (binary code) and allocates some separate memory space for it by utilizing functions of SK. The separation is enforced by SK during the entire running period of the application.

For remote attestations, a hash chain is constructed to establish the trust of SB and the upper level applications based on the root of trust provided by the hardware. Specifically, SK has a public-private key pair generated by TPM when the platform is initialized, where the public key is certified by the attestation identity key (AIK) of TPM. SK also generates a public-private key pair for SB, where the public key is certified by SK by signing with its private key and the private key is protected by SB with the sealed storage function of TPM. The key pair for SB is generated at the first time when SB is installed on the platform. For the attestation of a running process state, TPM signs a set of platform configuration register (PCR) values with its AIK key,[1] and SK signs the integrity value of SB with its private key, while SB signs the integrity value of the application code. These three signatures are then sent to the attestation challenger. The challenger verifies all these signatures and the public key certificates of AIK, SK, and SB, respectively. If all are valid and the integrity values match, the application is trusted.

Note that the "trust" of an application does not imply any extra privileges. The trust verification provides high assurance that it behaves as expected, but does not make any decision as to whether it is guaranteed to be safe. This is why access control mechanisms are needed to confine application activities, as illustrated in Section 5.

To support the remote attestation and sealed storage, a basic authentication infrastructure is needed to support our trust model. For simplicity we assume that necessary components for TC, such as privacy certificate authority (PCA) (to certify the AIKs of each platform), are available.

[1]We do not explicitly specify what PCR values are included in an attestation, since the required properties of a platform (including hardware, BIOS, and OS configurations) are application-specific.

3.2 Primitive Functions of SB

To provide transparent services to processes, SB implements an identical interface that a legacy OS provides to applications. When SB receives a system call from an application process, the call is checked by SB according to pre-defined policies based on the attributes of the caller and callee processes, the calling method, as well as possible context information (see Section 5 for more details). If the call is allowed, SB forwards it to the underlying OS. To provide a protected runtime environment for a process and enable trust verification by other components, SB provides the following primitive functions.

- SB allocates and maintains isolated memory space before launching a process, by utilizing the memory management functions provided by SK. This prevents interference between processes at runtime.
- SB provides integrity check and verification by measuring a process code's hash value and combining it with the hash value of any output data that it generates. This enables data and process authenticity verification on a local platform.
- SB enables process-based remote attestation by digitally signing the hash value of a process as well as its input. The remote attestation enables data and process authenticity verification across platforms.
- SB enforces flexible access control and information flow policies between processes on local platforms or between remote platforms based on the authenticity verification of data and processes.

Based on this trust model and primitive functions, in the following sections, we focus on the mechanisms of integrity measurement and authenticity verification in SB in Section 4 and access control through SB in Section 5.

4. PROCESS AND DATA AUTHENTICITY

The isolation provided by the hardware and SK can prevent malicious modification to the binary code of a running process. But protecting the integrity of the running code is not sufficient. Attacks can be mounted via other approaches, such as through input to a process. For example, a malicious entity can easily send or inject fake or erroneous data to a process and compromise its runtime integrity via buffer overflow. For a collaborative computing task like SETI@Home [2], a fraud peer can report fake results to the server without really performing the computation and get credits. Thus, it is essential to guarantee the authenticity of a process at runtime and the data it generates.

In our proposed architecture, the authenticity of a process and its generated data is achieved through a hash chain signed by SB. Figure 2 shows an example. Suppose process P_1 takes primitive input D_0 and generates output D_1, which is the input of process P_2. The output (D_2) of P_2 can be the input of another process, or it can be returned to P_1. Without loss of generality, we assume that interactions between P_1 and P_2 are on two different platforms, enhanced with SB_1 and SB_2, respectively. To ensure the authenticity of the process and its data, the following protocol is enforced.

1. P_1 takes primitive input D_0 and generates output D_1 and sends it to the local SB (SB_1). Note that SB_1 has verified

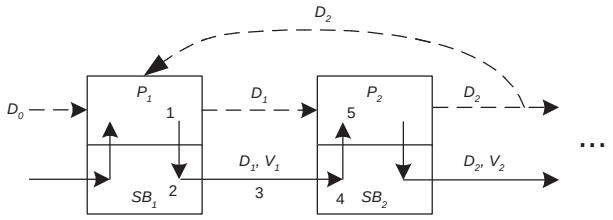

Figure 2: Integrity chain and verification

1	$P_1 \rightarrow SB_1$:	(D_0, D_1)
2	SB_1:	$V_1 = \{\mathcal{H}(\mathcal{H}(D_0)\|\|\mathcal{H}(P_1)\|\|\mathcal{H}(D_1))\}_{Kpr_1}$, where Kpr_1 is the private key of SB_1.
3	$SB_1 \rightarrow SB_2$:	$(D_1, V_1, \mathcal{H}(D_0), \mathcal{H}(P_1))$
4	SB_2:	verify the signature and integrity of $D_0\|\|P_1\|\|D_1$ with Kpb_1 (SB_1's public key).
5	$SB_2 \rightarrow P_2$:	D_1

Figure 3: Integrity measurements and verification

the source and integrity of D_0, either from local platform or from network resources.

2. SB_1 generates individual hash values for D_1, and concatenates with the hash value of D_0 and P_1 (measured when P_1 is launched), hashes the total and signs it with its private key. The result is V_1 as shown in Figure 2.

3. After SB_1 and SB_2 build a secure communication channel following remote attestation, SB_1 sends D_1 and V_1 to SB_2.[2]

4. SB_2 verifies the signature and the hash values, and makes decisions on the integrity and authenticity of P_1 and D_1.

5. Upon successful verification, SB_2 sends D_1 to P_2.

With the sealed signature capability of TC, SB can generate a valid digital signature only if it is loaded without modification, i.e., its integrity value matches what is sealed with its private key or any other key that protects the private key. Thus, the digital signature can be trusted by the applications on the local platform. Furthermore, with the remote attestation capability of underlying TC hardware, the trust of SBs on different platforms can be built, which enables the digital signature verification between applications across different platforms. Ideally, the remote attestation between SB_1 and SB_2 (required for step 3 in Figure 3) is a one-time operation between platforms whenever both of them are active can be reused.

As the above example demonstrates, from the process' viewpoint, only input/output data are transferred along the dashed lines shown in Figure 2. Thereby SB enforces the security mechanisms transparently to applications.

Note that SB does not detect attacks according to software vulnerabilities in the process code, such as buffer overflow with malicious inputs. Typically, "trusted computing cannot guarantee that software executed on a computer system is free of programming errors (vulnerabilities) that could be exploited" [22]. However, when the input of a process can be trusted (e.g., signed by a trusted party), SB can verify the execution status of the running code and the integrity of the output. For example, with a SB-enabled client, a SETI@Home server can verify the trust of the computed result from the client since the input of the client is generated by the server itself or trusted parties for task assignments.

The combined integrity verification of input/output data and process code can be used in many traditional communication mechanisms on a single platform or between platforms, such as pipe, signal, and remote procedure call (RPC). For process communications through shared memory and files, shared components also need to be protected by SB when loaded or created, and similar mechanisms can be used for secure communications between processes. Providing details of these mechanisms is beyond the scope of this paper.

5. ACCESS CONTROL ENFORCEMENT

The goal of access control through SB is to control the information flow between isolated processes. Information can flow during interactions between processes (e.g., call interfaces and return results), or accessing shared resources on the computing platform (e.g., read or write local files, network resources).[3] As aforementioned, with the runtime process integrity and the authenticity of process communications and input/output, we still cannot guarantee overall security. To ensure the overall security of a process, it is also essential to control information flow between processes according to application or organization specific policies. Beyond preserving a binary code's integrity with runtime space isolation and integrity verification with a hash chain, our architecture integrates application semantics into the integrity consideration, by introducing an application context-aware access control model enforced by SB.

Figure 4 sketches the access control architecture with SB, where the policy manager is launched whenever SB is loaded, with which the platform owner or system administrator define access control policies. The policy manager mainly consists of a policy decision point (PDP) and a policy database, of which the integrity can be verified by SB before any access control decision is made. Whenever a process needs to access another one (e.g., read data from or

[2]The remote attestation is either challenged by SB_1 or SB_2, or both depending on application and trust requirements.

[3]Note that we do not address the covert channel problem, which is generally considered to be beyond the scope of TC technologies.

write data to this process), the reference monitor evaluates the request by querying the policy manager, which in turn queries a policy database or another decision point (e.g., a higher level PDP in an organization). If the access is allowed, SB forwards the request to the destination process and returns the result to the requesting process. That is, SB acts as both a policy enforcement point (PEP) and a communication proxy for access control between applications. This makes it transparent to upper layer applications.

The additional advantage of this architecture is the separation of policy management and enforcement, which is a general requirement for modern complex computing systems. This feature makes our approach policy-neutral. That is, different types of security policies can be supported according to specific application requirements, such as role-based, domain/type-based, and history-based access control policies. In addition, the separation of policy management and enforcement makes SB small and simple, and enables parallel involvements of individuals.

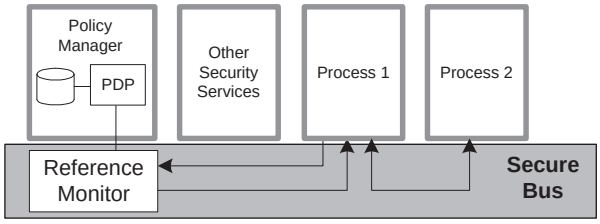

Figure 4: Access control architecture

5.1 Access Control Model

In our access control model, the process of an application is a subject, and resources are objects. Since a process can be accessed by another process, e.g., to build a connection to send or receive data, or to suspend/resume/kill the other, a process can be both a subject and an object.

An access control policy specifies whether an access can be allowed, by checking some context conditions about the requesting subject and the target object. Logically, a permission is defined as a triple (s, o, r), where s is the accessing subject, o is the object, and r is the access right. An access control decision is determined by security attributes of the subject and the object, such as security clearance/classification, types, etc., which are initially assigned by the security administrator, and can be updated during the lifetime of the system as the side-effect of the subjects' accesses. For a process, its attributes also include the context information of the user that launches the application, such as the user id, security classification, role, group, domain, etc. In this work we assume that only one user is involved in a process at a time.[4] On a single platform, the process ID can be used to identify the process. In a distributed computing environment, a globally unique process ID can be constructed through the hash of its code signed by SB. Access control policies can be specified and stored in a database or by XML files [21]. Also, there are several formal policy specification languages [9, 11, 17] that can be used. As we focus on high-level platform architecture and security enforcement mechanisms, we do not discuss how to specify policies in this paper.

As each process corresponds to a user, the authentication of the user is a prerequisite for access control, so as to obtain the security attributes of the user. Our architecture does not explicitly include a user authentication mechanism. Instead, SB only measures the integrity of processes, and forwards access requests to the PDP, which verifies the attributes of subjects and makes access control decisions according to specified security policies. However, existing user-based authentication mechanisms can be easily integrated into our architecture seamlessly. For example, in an enterprise environment, SB can accept the authentication ticket of a user provided by a trusted Kerberos server and determine his/her permissions based on group and domain names.

5.2 Enforcing Mandatory Access Control

To demonstrate the power of SB, we illustrate how mandatory access control (MAC) policies, which most commodity OS cannot enforce natively, can be enforced with SB. By assigning different levels of labels to computing entities, MAC controls one-way information flow for integrity or confidentiality. There are several approaches to support MAC. Among them, the Chinese Wall policy [10, 26] supports controlled information flow according to dynamic properties of the accessing subject or user, which can be used to enforce history-based MAC policies. We study how the Chinese Wall policy can be supported in our architecture as an example.

In the Chinese Wall policy, objects are categorized into mutually disjoint conflict-of-interest classes, and a user cannot access more than one object in a single class. The Chinese Wall policy can be described with a lattice-based access control policy as described in [26], where a user is assigned a label indicating the objects that s/he can access, and the set of all possible labels forms a lattice except that the topmost label is not assigned to any user. A user or a subject's security label can be updated according to his/her access history, which determines its further access permissions to other objects. This dynamic property makes it useful to express information flow control in collaborative computing systems. For example, a subject who participates in a collaborative project cannot write the data of the project to any file or container that is readable by a subject outside of the project. That is, once the subject joins a project, it cannot access the object of another project that conflicts with the one it joins.

A user in our Chinese Wall policy is a human being that obtains information by launching processes, which are represented as subjects of the user. A user can have multiple subjects, each of which is assigned with a label dominated by the label of the user in the lattice. Information flow between these subjects may or may not be allowed, depending on the relationship between their labels. Each user u has a maximum security label $\mathcal{L}_m(u)$ in a Chinese Wall lattice, which is pre-determined by the system administrator.

To enforce the policy between processes for different rights, a set of rules are defined for $read$, $write$, and $create$ rights as follows. Each rule specifies where a permission can be granted by checking the subject's and object's label relationship, and if necessary updates the subject label as the result of granting the permission.

(1). $(u, s, create) \Rightarrow \mathcal{L}(s) \leq \mathcal{L}_m(u)$, where $\mathcal{L}_m(u)$ and $\mathcal{L}(s)$ are the labels of user u and subject s, respectively. This pol-

[4] In reality there may be multiple users executing an application simultaneously, with each user having different security attributes. A simple policy may require that each user must satisfy the access control policy for the access request. We leave the details of multi-user access control for future work.

icy indicates that a user u can $create$ a subject (process) s and the subject's label is lower than or equal to the user's label. By "create" a process we mean the user invokes a program which the new process tuns.

(2). $(s, o, create) \Rightarrow \mathcal{L}(o) \leq \mathcal{L}(s)$, where s is subject process of a user, and o is be another process or other general object (file, directory, etc.) This policy specifies that a subject s can create a passive object (e.g., a file or directory) or invoke a process o, and the new object's label is lower than or equal to the subject's label.

(3). $(s, o, read) \Rightarrow \mathcal{L}(s) \geq \mathcal{L}(o)$. A subject s can $read$ an object o only if s's label dominates (higher than or equal to) o's label. This is referred as the read-down or simple-property of MAC.

(4). $(s, o, write) \Rightarrow \mathcal{L}(s) \leq \mathcal{L}(o)$. A subject s can $write$ an object o only if o's label dominates (higher than or equal to) s's label. This is referred as the write-up or star-property of MAC.

(5). If $\mathcal{L}(s)$ and $\mathcal{L}(o)$ are not *comparable* in the lattice, then $(s, o, read) \Rightarrow \big(\mathcal{L}(s)' = \mathcal{L}(s) \oplus \mathcal{L}(o)\big) \wedge \big(\mathcal{L}(s)' \leq \mathcal{L}_m(u)\big)$, where u is the user represented by s. This policy indicates that whenever a subject s wants to $read$ an object o and their labels are not comparable, then the label of the subject is updated to the least upper bound of their labels through the $\oplus$ operation, and this label must be lower than or equal to the label that its user can have, otherwise the access is denied. This is referred as the high-watermark property of MAC.

(6). If $\mathcal{L}(s_1)$ and $\mathcal{L}(s_2)$ are not *comparable* in the lattice, then $(s_1, s_2, write) \Rightarrow \big(\mathcal{L}(s_2)' = \mathcal{L}(s_1) \oplus \mathcal{L}(s_2)\big) \wedge \big(\mathcal{L}(s_2)' \leq \mathcal{L}_m(u_2)\big)$, where u_2 is the representing user of s_2. This policy indicates that whenever a subject process s_1 wants to $write$ an object (another subject process) s_2 and their labels are not comparable, then the label of s_2 is updated to the least upper bound of s_1 and s_2, and this new label must be less than or equal to the label that its user can have, otherwise the access is denied. This is another form of the high-watermark property. Note that s_1 and s_2 may or may not be the subjects of the same user.[5]

In the first two rules, the label of a created object is not specified. Two options can be considered here: the new object's label is determined at the discretionary of the user, or by some other organizational or administrative policies. In rule (5) and (6), a subject's label is updated as the least upper bound of the two labels, which results in that this subject can read from any object with the same label or any label that the subject's label dominates, but can only write to objects with higher or equal labels. If the least upper bound is higher than the user's assigned maximum label, the update cannot be performed and the access is denied. Since there is no policy to downgrade a subject's label, once a subject is assigned a higher level label, it cannot write to lower level objects. For example, a process with a label of its domain name originally can read and write within the domain. For the collaboration purpose it reads objects in another domain, and its label is updated to the compartment of both domain names as the result of the accessing. This prevents the process from writing the joined data into any object with a label of its original domain.

It is important to understand that a user may run the same program, such as a text editor, as a high level subject (process) or a low level subject. Even though both subjects run the same program on behalf of the same user, they obtain different privileges (e.g., to read from or write to other objects) due to their different security labels. Due to the dynamic transitions of a subject's label according to its access history, our model is *non-tranquil*.

[5]Note the asymmetry between rules 5 and 6. In both cases only the label of a subject can change. In rule 5 it is the label of the subject doing the read that changes, whereas in rule 6 it is the subject being written to whose label changes. Hence in 5 the target object o may be a passive object or an active subject, but in rule 6 the target s_2 must be an active subject and cannot be a passive object.

6. PROTOTYPE IMPLEMENTATION AND EVALUATION

To study the feasibility and performance overhead of our proposed architecture, we have implemented a prototype. In this section, we present the details of implementation and the evaluation results.

6.1 Prototype Overview

In our prototype, the isolation of running processes is achieved through application-level virtualization technology. Specifically, user-model Linux (UML) [13] is used to provide isolated runtime environments for individual processes. UML is a ported Linux version that can run in a Linux host's user space. A process running in a UML is a normal process in the host operating system, but is contained in UML by tracing and diverting all of its system calls to a user space kernel (UML kernel). Underlying hardware resources are virtualized by the host OS. Therefore a UML is a user space sandbox and existing programs can run in a UML without any changes. The memory space of processes in different UMLs on a single platform is strongly separated by a virtual machine monitor (VMM) running in the host OS.

Figure 5 shows the architecture of our prototype. The inter-process communication is implemented with Unix sockets. A UML process communicates with the host and external platforms with universal TUN/TAP driver [4], which provides packet reception and transmission for user space programs. The reference monitor is implemented as a user space daemon in the host OS. As the labels of objects like files and directories are *static*, they are stored with the object itself. Specifically, a directory is labelled if there is a SB_LABEL file in the directory, and the label name is the content of the file. Each file under this directory is assigned with the same label. For processes, their labels dynamically change. To improve the performance, the reference monitor maintains a table with tuples $(pid, label)$, where pid is a unique process identity.

Access control policies are stored in a file accessible to the reference monitor. Each policy entry includes the label of the requesting subject and the label of the target object, and the allowed action as the result of the access request. There are two types of actions, one is the generic rights of a system such as read and write, and the other is the update of the subject or the object label as the result of allowing an access. For an update action, the access control decision is made only after a successful update action, i.e., the corresponding label has been changed in the table. Also, the reference monitor maintains another file storing the maximum labels of users. Note that since all labels form a lattice, there is a finite number of labels and domination relationships between them.

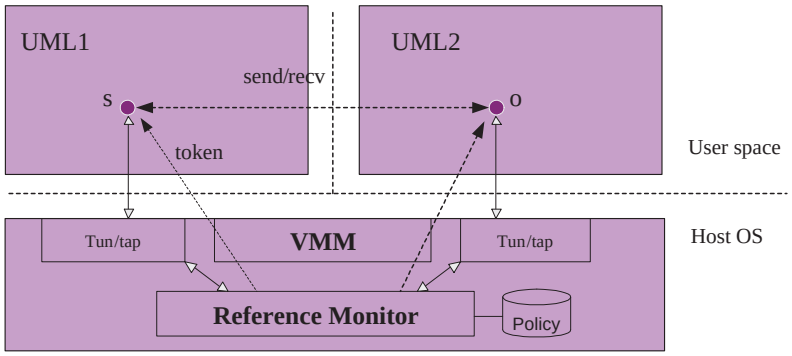

Figure 5: Platform architecture of SB prototype

Our prototype runs on a Pentium III 666MHz machine with 256 MB memory, Debian Linux with kernel 2.4.27. UML also uses a patched Linux kernel 2.4.27.

6.2 Implementation of Chinese Wall Policy

To enforce the Chinese Wall policy in our proposed architecture, the subject and object security labels must be identified by the reference monitor before allowing an access. Therefore we need to determine the *effective user* of a process. A user can create a process, which in turn can create other processes. Recursively, the effective user of a process is that of its *parent* subject (the user or the process from which it is created). As UML is a user space process in the host OS, we assume that for each process there is only one user involved. Therefore the effective user of a process in a UML is the effective user of the UML process in the host OS, which can be obtained with `getuid` in Linux. In Linux file systems, file permission bits provide discretionary access control (DAC), while we implement a parallel label-based MAC mechanism in our prototype. In a real system these two mechanisms can be enforced concurrently.

In our implementation, a security label is defined as a set of group names in the Linux system. According to the Chinese Wall policy, a user can be in only one group of a set of conflicting groups, and its maximum security label ($\mathcal{L}_m(u)$) is the set of all group names that it belongs to. The domination relationship between labels is the subset relation between group sets. When a user or a process creates another process (refer to rules 1 and 2 in Section 5.2), the security label of the new process can be set by the user or the process as a parameter of the invocation. By default, if no parameter is present, the new process is assigned with the same label as its parent and so recorded by the reference monitor.

The $read$ and $write$ permissions (refer to rules 3 to 6 in Section 5.2) are implemented with `send` and `recv` of TCP socket, which are used to send and receive messages through a socket connection between two processes. To support the method-level control, we use a *token* associated with each socket connection to specify allowed information flow directions. Logically, a token maps a socket connection to a right set of $\{read, write\}$, where a socket connection has a source (subject) IP/port and a destination (object) IP/port. For example, a token with $read$ indicates that the subject can receive messages from the object, and the object can send messages to the subject. All other permissions are denied.

The following describes the general procedure we implement for an access control request (s, o, r), where s is the subject process and o is the object process. Similar procedure happens when o is a static object such as file and directory.

1. Before s wants to connect to o (without loss of generality, we assume the subject always actively generates the request), it sends the request to the reference monitor with the target object's information.

2. The reference monitor obtains the security labels of s and o (recorded by SB when the processes are created), and queries the access control policies. A token is issued to specify allowed access permissions.

3. Before any of `send` and `recv` methods is called, the token is checked if the method can be activated, otherwise an access control exception is generated.

6.3 Performance Results

We measured the communication and access control overhead between two processes in the same host OS (without isolation enforced) and in two different UMLs isolated with VMM in the host OS, respectively. Under each case we measured the average overhead with and without access control between the subject and object processes. We measured the time for the access control decision check and the actual communication time. Since extra performance overhead is introduced with access requests from a subject process and the decision making from the reference monitor, denying and allowing an access result in the same performance overhead.[6] So an access is always allowed when the access control decision is checked in our performance study. Table 1 shows the average values of 50 measurements. Since the communication that we implemented in the prototype is through simple query-and-response messages between processes, the overhead introduced by the access control enforcement is comparable to the communication overhead. While the overhead significantly increases from the control within the same OS to that between UMLs in the same host OS, the access control enforcement overhead is still less than 2.0 ms (less than 1/3 of the total overhead), which is acceptable for most applications.

[6]Normally denying and allowing an access may result in different overhead if different number(s) of policies are evaluated. Here we only consider a small number of total policies such that the difference can be ignored.

	no isolation		isolation with VMM	
	no control	with control	no control	with control
access control (ms)	-	0.465	-	1.926
communication (ms)	0.372	0.369	4.484	4.551
total (ms)	0.372	0.834	4.484	6.477

Table 1: Access control and communication overhead

Strategies can be used to further reduce the access control overhead. For example, the reference monitor can cache a finite number of tokens it has issued. If the security labels of the communicating parties in a token have not updated upon the latest access, the token can be reused without querying the policy decision point (the policy file in our prototype). This is useful for real applications with complex access control policies and in distributed computing environments.

7. RELATED WORK

Considerable research has been conducted to provide application level security based on TC technologies. In [25], a security kernel is used beyond hardware to provide separated runtime space for the legacy operating system and "secure applications" (e.g., DRM applications). But communications between legacy applications and these secure applications are not supported, as this architecture is for multilateral security policies, e.g., platform owner's security and DRM policies. Similarly, a language-based virtual machine is proposed in [16] to provide trusted services, which is only for applications developed with a Java-like program language. It does not support communications between general processes and processes in a virtual machine.

So far only a few researchers have considered the simultaneous application-level security and flexible communication requirements. Besides those mentioned in Section 1, Terra [15] and sHype (secure Hypervisor) [24] use virtual machine monitor (VMM) as the trusted layer, which can support multiple legacy operating systems on a single platform. Access control is enforced in the VMM layer for resource sharing between upper VMs. The main difference from our approach is that in our approach the security enforcement is performed in a middle layer, which is above the main OS and below the applications. That is, our approach can provide finer-grained security services between applications by integrating application context information for access control.

Another line of work focuses on securing operating systems, such as Security-enhanced Linux (SELinux) [19], TrustedBSD [5], and Linux Security Modules (LSM) [29]. In these systems, the kernel is extended to include authorization modules and enforce access control policies. For example, in SELinux, security classes are defined for objects such as files, links, and processes, and accesses to objects from subjects are controlled by policies. Because of the complexity and the huge size of a general-purpose OS, it is considered that purely OS-based security enforcement cannot provide high assurance security services to applications, which is demonstrated by the increasing number of kernel-level rootkits in commodity OS's. These studies differ from our architecture in that our proposed trusted component (SB) is in the middle layer between the OS kernel and the user space processes, such that it is transparent to existing OS and applications. Furthermore, by leveraging trusted hardware, our architecture provides high assurance for the enforcement of policies.

Among these schemes, BIND [27] and KernelSec [23] are closest to ours. Although BIND [27] also uses the hash chain for integrity check of a process and its data, our approach is significantly different from that in BIND. First, with SB, both the hash values of the input/output and the process binary code are included in the signature that is sent to the downstream process. It is not only the integrity of the process code. This is practically necessary to verify that the output of a process is obtained with genuine input. For example, in collaborative computing systems like SETI@Home, a server needs to ensure that a peer does the computation with the input that the server assigns, but not something faked. In BIND, only the process code and its output data are verified, which cannot capture the integrity and authenticity of the input. Second, the integrity verification in our architecture is performed by SB on a platform, which makes this function transparent to applications since they always go through SB to communicate with each other. While in BIND, for the purpose of fine-grained attestation, the integrity measurements and verifications are based on some critical sections of a process, which calls the corresponding functions provided by BIND. So the integrity verification is performed by individual applications, whereby the security functions are not transparent to applications.

Similar to our approach, KernelSec [23] supports general security policies such as MAC and information flow control for applications. But as it is implemented at the OS kernel level without root of trust, KernelSec cannot provide high assurance of security enforcement. On the other side, SB supports more flexible application- and organization-specific access control policies with high assurance.

8. CONCLUSIONS

In this paper we propose a novel architecture for trusted computing. A trusted component called SecureBus is located between the main OS and applications to provide strong memory space isolation and secure communication for user applications. SB effectively preserves application integrity by attesting the integrity and authenticity of process and data, and enforcing flexible mandatory access control policies for information flow between applications, both of which are required by applications to defend software-based attacks. The major advantage of our architecture is that SB is transparent to both the underlying OS and applications, and can provide data authentication and flexible access control between processes simultaneously. In addition, our architecture enables the separation of security mechanisms from functionality in the design and development of systems and applications, which is convenient for legacy applications and OS's. We have implemented a prototype system to study its feasibility and the access control performance. The experimental results show that SB is effective.

In this work we have examined the application of SB through the Chinese Wall policy implemented using a lattice of security labels. The architecture is applicable to a much wider range of policies, details of which will be studied in future work.

9. ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their helpful comments. The work is partially supported by NSF grants CNS-0509061 and CNS-0621631, and by a grant from Intel.

10. REFERENCES

[1] *LaGrande Technology Preliminary Architecture Specification, http://www.intel.com/technology/security/downloads/PRELIM-LT-SPEC_D52212.htm*.

[2] SETI@Home, http://setiathome.ssl.berkeley.edu/.

[3] *TCG Specification Architecture Overview*. https://www.trustedcomputinggroup.org.

[4] Universial TUN/TAP driver. http://vtun.sourceforge.net/tun/.

[5] TrustedBSD: Adding trusted operating system features to FreeBSD. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, pages 15–28, Boston, MA, USA, June 28 2001.

[6] AMD platform for trustworthy computing. Microsoft WinHEC, http://www.microsoft.com/whdc/winhec/pres03.mspx, 2003.

[7] Technical introduction to next-generation secure computing base (NGSCB). Microsoft WinHEC, 2003.

[8] A. Baliga, L. Iftode, and X. Chen. Paladin: Automated detection and containment of rootkit attacks. Technical Report DCS-TR-593, Rutgers University, Department of Computer Science, 2006.

[9] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transaction on Information System Security*, 6(1):71–127, 2003.

[10] D. Brewer and M. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium On Research in Security and Privacy*, pages 206–214, Oakland, California, 1988.

[11] N. Damianou, N. Dulay, E. Lupu, , and M. Sloman. The ponder policy specification language. In *Proceedings of the Workshop on Policies for Distributed System s and Networks*, 2001.

[12] Department of Defense National Computer Security Center. *Trusted Database Interpretation of the Trusted Computer Systems Eval uation Criteria*, April 1991. NCSC-TG-021.

[13] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000.

[14] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercmputer Applications*, 11(2), 1997.

[15] 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*, pages 193–206, Bolton Landing, New York, USA, October 1922 2003.

[16] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation - a virtual machine directed approach to trusted computing. In *Proceedings of the Third virtual Machine Research and Technology Symposium*, pages 29–41, San Jose, CA, USA, May 6-7 2004. USENIX.

[17] S. Jajodia, P. Samarati, , and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium On Research in Security and Privacy*, pages 31–42, Oakland, CA, USA, 1997.

[18] J. F. Levine, J. B. Grizzard, and H. L. Owen. Detecting and categorizing kernel-level rootkits to aid future detection. *IEEE Security & Privacy*, 4(1):24–32, Jan.-Feb. 2006.

[19] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of USENIX Annual Technical Conference*, pages 29 – 42, June 25-30 2001.

[20] S. Muir, L. Peterson, M. Fiuczynski, J. Cappos, and J. Hartman. Proper: Privileged operations in a virtualised system environment. In *Proceedings of Usenix Annual Technical Conference*, 2005.

[21] OASIS XACML TC. *Core Specification: eXtensible Access Control Markup Language (XACML)*, 2005.

[22] R. Oppliger and R. Rytz. Does trusted computing remedy computer security problems? *IEEE Security & Privacy*, 3(2):16–19, 2005.

[23] M. Radhakrishnan and J. A. Solworth. Application support in the operating system kernel. In *Proceedings of ACM Symposium on InformAtion, Computer and Communications Security*, 2006.

[24] R.Sailer, T. Jaeger, E. Valdez, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a mac-based security architecture for the xen opensource hypervisor. Technical report, IBM Research Report RC23629, 2005.

[25] A. Sadeghi and C. Stuble. Taming trusted platforms by operating system design. In *Proceedings of the 4th International Workshop for Information Security Applications, LNCS 2908*, pages 286–302, Berlin, Germany, August 2003.

[26] R. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11), November 1993.

[27] E. Shi, A. Perrig, and L. Van Doorn. Bind: a fine-grained attestation service for secure distributed systems. In *Proceeedings of IEEE Symposium on Security and Privacy*, pages 154–168, Oakland, CA, USA, May 8-11 2005.

[28] Sean Smith. *Trusted Computing Platforms: Design and Applications*. Springer, 2005.

[29] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, 2002.